Apply Pod Security Standards at the Namespace Level

Note

This tutorial applies only for new clusters.

Pod Security Admission is an admission controller that applies Pod Security Standards when pods are created. It is a feature GA'ed in v1.25. In this tutorial, you will enforce the `baseline` Pod Security Standard, one namespace at a time.

You can also apply Pod Security Standards to multiple namespaces at once at the cluster level. For instructions, refer to Apply Pod Security Standards at the cluster level.

Before you begin

Install the following on your workstation:

- kind
- kubectl

Create cluster

1. Create a `kind` cluster as follows:

   ```
   kind create cluster --name psa-ns-level
   ```

   The output is similar to this:

   ```
   Creating cluster "psa-ns-level" ...
    ✓ Ensuring node image (kindest/node:v1.34.0) 🖼
    ✓ Preparing nodes 📦
    ✓ Writing configuration 📜
    ✓ Starting control-plane 🕹
    ✓ Installing CNI 🔌
    ✓ Installing StorageClass 💾
   Set kubectl context to "kind-psa-ns-level"
   You can now use your cluster with:

   kubectl cluster-info --context kind-psa-ns-level

   Not sure what to do next? 😅  Check out https://kind.sigs.k8s.io/docs/user/quick-start/
   ```

2. Set the kubectl context to the new cluster:

   ```
   kubectl cluster-info --context kind-psa-ns-level
   ```

   The output is similar to this:

   ```
   Kubernetes control plane is running at https://127.0.0.1:50996
   CoreDNS is running at https://127.0.0.1:50996/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

   To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
   ```

Create a namespace

Create a new namespace called `example`:

```
kubectl create ns example
```

The output is similar to this:

```
namespace/example created
```

Enable Pod Security Standards checking for that namespace

1. Enable Pod Security Standards on this namespace using labels supported by built-in Pod Security Admission. In this step you will configure a check to warn on Pods that don't meet the latest version of the *baseline* pod security standard.

   ```
   kubectl label --overwrite ns example \
      pod-security.kubernetes.io/warn=baseline \   pod-security.kubernetes.io/warn-version=latest
   ```

2. You can configure multiple pod security standard checks on any namespace, using labels. The following command will `enforce` the `baseline` Pod Security Standard, but `warn` and `audit` for `restricted` Pod Security Standards as per the latest version (default value)

   ```
   kubectl label --overwrite ns example \
     pod-security.kubernetes.io/enforce=baseline \  pod-security.kubernetes.io/enforce-version=latest \  pod-security.kubernetes
   ```

Verify the Pod Security Standard enforcement

1. Create a baseline Pod in the `example` namespace:

   ```
   kubectl apply -n example -f https://k8s.io/examples/security/example-baseline-pod.yaml
   ```

   The Pod does start OK; the output includes a warning. For example:

   ```
   Warning: would violate PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container "nginx" must set securit
   pod/nginx created
   ```

2. Create a baseline Pod in the `default` namespace:

   ```
   kubectl apply -n default -f https://k8s.io/examples/security/example-baseline-pod.yaml
   ```

   Output is similar to this:

   ```
   pod/nginx created
   ```

The Pod Security Standards enforcement and warning settings were applied only to the `example` namespace. You could create the same Pod in the `default` namespace with no warnings.

Clean up

Now delete the cluster which you created above by running the following command:

```
kind delete cluster --name psa-ns-level
```

What's next

- Run a shell script to perform all the preceding steps all at once.

  1. Create kind cluster
  2. Create new namespace
  3. Apply `baseline` Pod Security Standard in `enforce` mode while applying `restricted` Pod Security Standard also in `warn` and `audit` mode.
  4. Create a new pod with the following pod security standards applied

- Pod Security Admission
- Pod Security Standards
- Apply Pod Security Standards at the cluster level

Restrict a Container's Syscalls with seccomp

FEATURE STATE: `Kubernetes v1.19 [stable]`

Seccomp stands for secure computing mode and has been a feature of the Linux kernel since version 2.6.12. It can be used to sandbox the privileges of a process, restricting the calls it is able to make from userspace into the kernel. Kubernetes lets you automatically apply seccomp profiles loaded onto a node to your Pods and containers.

Identifying the privileges required for your workloads can be difficult. In this tutorial, you will go through how to load seccomp profiles into a local Kubernetes cluster, how to apply them to a Pod, and how you can begin to craft profiles that give only the necessary privileges to your container processes.

Objectives

- Learn how to load seccomp profiles on a node
- Learn how to apply a seccomp profile to a container
- Observe auditing of syscalls made by a container process
- Observe behavior when a missing profile is specified
- Observe a violation of a seccomp profile
- Learn how to create fine-grained seccomp profiles
- Learn how to apply a container runtime default seccomp profile

Before you begin

In order to complete all steps in this tutorial, you must install kind and kubectl.

The commands used in the tutorial assume that you are using Docker as your container runtime. (The cluster that `kind` creates may use a different container runtime internally). You could also use Podman but in that case, you would have to follow specific instructions in order to complete the tasks successfully.

This tutorial shows some examples that are still beta (since v1.25) and others that use only generally available seccomp functionality. You should make sure that your cluster is configured correctly for the version you are using.

The tutorial also uses the `curl` tool for downloading examples to your computer. You can adapt the steps to use a different tool if you prefer.

Note:

It is not possible to apply a seccomp profile to a container running with `privileged: true` set in the container's `securityContext`. Privileged containers always run as `Unconfined`.

Download example seccomp profiles

The contents of these profiles will be explored later on, but for now go ahead and download them into a directory named `profiles/` so that they can be loaded into the cluster.

- audit.json
- violation.json
- fine-grained.json

pods/security/seccomp/profiles/audit.json  Copy pods/security/seccomp/profiles/audit.json to clipboard

```
{
    "defaultAction": "SCMP_ACT_LOG"
}
```

pods/security/seccomp/profiles/violation.json  Copy pods/security/seccomp/profiles/violation.json to clipboard

```
{
    "defaultAction": "SCMP_ACT_ERRNO"
}
```

pods/security/seccomp/profiles/fine-grained.json  Copy pods/security/seccomp/profiles/fine-grained.json to clipboard

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "names": [
                "accept4",
                "epoll_wait",
                "pselect6",
                "futex",
                "madvise",
                "epoll_ctl",
                "getsockname",
                "setsockopt",
                "vfork",
                "mmap",
                "read",
                "write",
                "close",
                "arch_prctl",
                "sched_getaffinity",
                "munmap",
                "brk",
                "rt_sigaction",
                "rt_sigprocmask",
                "sigaltstack",
                "gettid",
                "clone",
                "bind",
                "socket",
                "openat",
                "readlinkat",
                "exit_group",
                "epoll_create1",
                "listen",
                "rt_sigreturn",
                "sched_yield",
                "clock_gettime",
                "connect",
                "dup2",
                "epoll_pwait",
                "execve",
                "exit",
                "fcntl",
                "getpid",
                "getuid",
                "ioctl",
                "mprotect",
                "nanosleep",
                "open",
                "poll",
                "recvfrom",
                "sendto",
                "set_tid_address",
                "setitimer",
                "writev",
                "fstatfs",
                "getdents64",
                "pipe2",
                "getrlimit"
            ],
            "action": "SCMP_ACT_ALLOW"
        }
    ]
}
```

Run these commands:

```
mkdir ./profiles
curl -L -o profiles/audit.json https://k8s.io/examples/pods/security/seccomp/profiles/audit.json
curl -L -o profiles/violation.json https://k8s.io/examples/pods/security/seccomp/profiles/violation.json
curl -L -o profiles/fine-grained.json https://k8s.io/examples/pods/security/seccomp/profiles/fine-grained.json
ls profiles
```

You should see three profiles listed at the end of the final step:

```
audit.json  fine-grained.json  violation.json
```

Create a local Kubernetes cluster with kind

For simplicity, kind can be used to create a single node cluster with the seccomp profiles loaded. Kind runs Kubernetes in Docker, so each node of the cluster is a container. This allows for files to be mounted in the filesystem of each container similar to loading files onto a node.

pods/security/seccomp/kind.yaml  Copy pods/security/seccomp/kind.yaml to clipboard

```
apiVersion: kind.x-k8s.io/v1alpha4
kind: Clusternodes:- role: control-plane  extraMounts:  - hostPath: "./profiles"    containerPath: "/var/lib/kubelet/seccomp/profi
```

Download that example kind configuration, and save it to a file named `kind.yaml`:

```
curl -L -O https://k8s.io/examples/pods/security/seccomp/kind.yaml
```

You can set a specific Kubernetes version by setting the node's container image. See Nodes within the kind documentation about configuration for more details on this. This tutorial assumes you are using Kubernetes v1.34.

As a beta feature, you can configure Kubernetes to use the profile that the container runtime prefers by default, rather than falling back to `Unconfined`. If you want to try that, see enable the use of `RuntimeDefault` as the default seccomp profile for all workloads before you continue.

Once you have a kind configuration in place, create the kind cluster with that configuration:

```
kind create cluster --config=kind.yaml
```

After the new Kubernetes cluster is ready, identify the Docker container running as the single node cluster:

```
docker ps
```

You should see output indicating that a container is running with name `kind-control-plane`. The output is similar to:

```
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS
6a96207fed4b        kindest/node:v1.18.2   "/usr/local/bin/entr…"   27 seconds ago      Up 24 seconds       127.0.0.1:42223->6443/t
```

If observing the filesystem of that container, you should see that the `profiles/` directory has been successfully loaded into the default seccomp path of the kubelet. Use `docker exec` to run a command in the Pod:

```
# Change 6a96207fed4b to the container ID you saw from "docker ps"
docker exec -it 6a96207fed4b ls /var/lib/kubelet/seccomp/profiles
```

```
audit.json  fine-grained.json  violation.json
```

You have verified that these seccomp profiles are available to the kubelet running within kind.

Create a Pod that uses the container runtime default seccomp profile

Most container runtimes provide a sane set of default syscalls that are allowed or not. You can adopt these defaults for your workload by setting the seccomp type in the security context of a pod or container to `RuntimeDefault`.

Note:

If you have the `seccompDefault` configuration enabled, then Pods use the `RuntimeDefault` seccomp profile whenever no other seccomp profile is specified. Otherwise, the default is `Unconfined`.

Here's a manifest for a Pod that requests the `RuntimeDefault` seccomp profile for all its containers:

pods/security/seccomp/ga/default-pod.yaml  Copy pods/security/seccomp/ga/default-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: default-pod  labels:    app: default-podspec:  securityContext:    seccompProfile:      type: RuntimeDefa
```

Create that Pod:

```
kubectl apply -f https://k8s.io/examples/pods/security/seccomp/ga/default-pod.yaml
```

```
kubectl get pod default-pod
```

The Pod should be showing as having started successfully:

```
NAME         READY   STATUS    RESTARTS   AGE
default-pod  1/1     Running   0          20s
```

Delete the Pod before moving to the next section:

```
kubectl delete pod default-pod --wait --now
```

Create a Pod with a seccomp profile for syscall auditing

To start off, apply the `audit.json` profile, which will log all syscalls of the process, to a new Pod.

Here's a manifest for that Pod:

pods/security/seccomp/ga/audit-pod.yaml Copy pods/security/seccomp/ga/audit-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: audit-pod  labels:    app: audit-podspec:  securityContext:    seccompProfile:      type: Localhost
```

Note:

Older versions of Kubernetes allowed you to configure seccomp behavior using annotations. Kubernetes 1.34 only supports using fields within `.spec.securityContext` to configure seccomp, and this tutorial explains that approach.

Create the Pod in the cluster:

```
kubectl apply -f https://k8s.io/examples/pods/security/seccomp/ga/audit-pod.yaml
```

This profile does not restrict any syscalls, so the Pod should start successfully.

```
kubectl get pod audit-pod
```

```
NAME        READY   STATUS    RESTARTS   AGE
audit-pod   1/1     Running   0          30s
```

In order to be able to interact with this endpoint exposed by this container, create a NodePort Service that allows access to the endpoint from inside the kind control plane container.

```
kubectl expose pod audit-pod --type NodePort --port 5678
```

Check what port the Service has been assigned on the node.

```
kubectl get service audit-pod
```

The output is similar to:

```
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
audit-pod   NodePort   10.111.36.142   <none>        5678:32373/TCP   72s
```

Now you can use `curl` to access that endpoint from inside the kind control plane container, at the port exposed by this Service. Use `docker exec` to run the `curl` command within the container belonging to that control plane container:

```
# Change 6a96207fed4b to the control plane container ID and 32373 to the port number you saw from "docker ps"
docker exec -it 6a96207fed4b curl localhost:32373
```

```
just made some syscalls!
```

You can see that the process is running, but what syscalls did it actually make? Because this Pod is running in a local cluster, you should be able to see those in `/var/log/syslog` on your local system. Open up a new terminal window and `tail` the output for calls from `http-echo`:

```
# The log path on your computer might be different from "/var/log/syslog"
tail -f /var/log/syslog | grep 'http-echo'
```

You should already see some logs of syscalls made by `http-echo`, and if you run `curl` again inside the control plane container you will see more output written to the log.

For example:

```
Jul  6 15:37:40 my-machine kernel: [369128.669452] audit: type=1326 audit(1594067860.484:14536): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:37:40 my-machine kernel: [369128.669453] audit: type=1326 audit(1594067860.484:14537): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:37:40 my-machine kernel: [369128.669455] audit: type=1326 audit(1594067860.484:14538): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:37:40 my-machine kernel: [369128.669456] audit: type=1326 audit(1594067860.484:14539): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:37:40 my-machine kernel: [369128.669517] audit: type=1326 audit(1594067860.484:14540): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:37:40 my-machine kernel: [369128.669519] audit: type=1326 audit(1594067860.484:14541): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:38:40 my-machine kernel: [369188.671648] audit: type=1326 audit(1594067920.488:14559): auid=4294967295 uid=0 gid=0 ses=4
Jul  6 15:38:40 my-machine kernel: [369188.671726] audit: type=1326 audit(1594067920.488:14560): auid=4294967295 uid=0 gid=0 ses=4
```

You can begin to understand the syscalls required by the `http-echo` process by looking at the `syscall=` entry on each line. While these are unlikely to encompass all syscalls it uses, it can serve as a basis for a seccomp profile for this container.

Delete the Service and the Pod before moving to the next section:

```
kubectl delete service audit-pod --wait
kubectl delete pod audit-pod --wait --now
```

Create a Pod with a seccomp profile that causes violation

For demonstration, apply a profile to the Pod that does not allow for any syscalls.

The manifest for this demonstration is:

pods/security/seccomp/ga/violation-pod.yaml Copy pods/security/seccomp/ga/violation-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: violation-pod  labels:    app: violation-podspec:  securityContext:    seccompProfile:      type: Localh
```

Attempt to create the Pod in the cluster:

```
kubectl apply -f https://k8s.io/examples/pods/security/seccomp/ga/violation-pod.yaml
```

The Pod creates, but there is an issue. If you check the status of the Pod, you should see that it failed to start.

```
kubectl get pod violation-pod
```

```
NAME            READY   STATUS             RESTARTS   AGE
violation-pod   0/1     CrashLoopBackOff   1          6s
```

As seen in the previous example, the `http-echo` process requires quite a few syscalls. Here seccomp has been instructed to error on any syscall by setting `"defaultAction": "SCMP_ACT_ERRNO"`. This is extremely secure, but removes the ability to do anything meaningful. What you really want is to give workloads only the privileges they need.

Delete the Pod before moving to the next section:

```
kubectl delete pod violation-pod --wait --now
```

Create a Pod with a seccomp profile that only allows necessary syscalls

If you take a look at the `fine-grained.json` profile, you will notice some of the syscalls seen in syslog of the first example where the profile set `"defaultAction": "SCMP_ACT_LOG"`. Now the profile is setting `"defaultAction": "SCMP_ACT_ERRNO"`, but explicitly allowing a set of syscalls in the `"action": "SCMP_ACT_ALLOW"` block. Ideally, the container will run successfully and you will see no messages sent to `syslog`.

The manifest for this example is:

pods/security/seccomp/ga/fine-pod.yaml  Copy pods/security/seccomp/ga/fine-pod.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: fine-pod  labels:    app: fine-podspec:  securityContext:    seccompProfile:      type: Localhost      l
```

Create the Pod in your cluster:

```
kubectl apply -f https://k8s.io/examples/pods/security/seccomp/ga/fine-pod.yaml
```

```
kubectl get pod fine-pod
```

The Pod should be showing as having started successfully:

```
NAME        READY   STATUS    RESTARTS   AGE
fine-pod    1/1     Running   0          30s
```

Open up a new terminal window and use `tail` to monitor for log entries that mention calls from `http-echo`:

```
# The log path on your computer might be different from "/var/log/syslog"
tail -f /var/log/syslog | grep 'http-echo'
```

Next, expose the Pod with a NodePort Service:

```
kubectl expose pod fine-pod --type NodePort --port 5678
```

Check what port the Service has been assigned on the node:

```
kubectl get service fine-pod
```

The output is similar to:

```
NAME        TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
fine-pod    NodePort   10.111.36.142   <none>        5678:32373/TCP   72s
```

Use `curl` to access that endpoint from inside the kind control plane container:

```
# Change 6a96207fed4b to the control plane container ID and 32373 to the port number you saw from "docker ps"
docker exec -it 6a96207fed4b curl localhost:32373
```

```
just made some syscalls!
```

You should see no output in the `syslog`. This is because the profile allowed all necessary syscalls and specified that an error should occur if one outside of the list is invoked. This is an ideal situation from a security perspective, but required some effort in analyzing the program. It would be nice if there was a simple way to get closer to this security without requiring as much effort.

Delete the Service and the Pod before moving to the next section:

```
kubectl delete service fine-pod --wait
kubectl delete pod fine-pod --wait --now
```

Enable the use of `RuntimeDefault` as the default seccomp profile for all workloads

FEATURE STATE: `Kubernetes v1.27 [stable]`

To use seccomp profile defaulting, you must run the kubelet with the `--seccomp-default` command line flag enabled for each node where you want to use it.

If enabled, the kubelet will use the `RuntimeDefault` seccomp profile by default, which is defined by the container runtime, instead of using the `Unconfined` (seccomp disabled) mode. The default profiles aim to provide a strong set of security defaults while preserving the functionality of the workload. It is possible that the default profiles differ between container runtimes and their release versions, for example when comparing those from CRI-O and containerd.

Note:

Enabling the feature will neither change the Kubernetes `securityContext.seccompProfile` API field nor add the deprecated annotations of the workload. This provides users the possibility to rollback anytime without actually changing the workload configuration. Tools like `crictl inspect` can be used to verify which seccomp profile is being used by a container.

Some workloads may require a lower amount of syscall restrictions than others. This means that they can fail during runtime even with the `RuntimeDefault` profile. To mitigate such a failure, you can:

- Run the workload explicitly as `Unconfined`.
- Disable the `SeccompDefault` feature for the nodes. Also making sure that workloads get scheduled on nodes where the feature is disabled.
- Create a custom seccomp profile for the workload.

If you were introducing this feature into production-like cluster, the Kubernetes project recommends that you enable this feature gate on a subset of your nodes and then test workload execution before rolling the change out cluster-wide.

You can find more detailed information about a possible upgrade and downgrade strategy in the related Kubernetes Enhancement Proposal (KEP): Enable seccomp by default.

Kubernetes 1.34 lets you configure the seccomp profile that applies when the spec for a Pod doesn't define a specific seccomp profile. However, you still need to enable this defaulting for each node where you would like to use it.

If you are running a Kubernetes 1.34 cluster and want to enable the feature, either run the kubelet with the `--seccomp-default` command line flag, or enable it through the kubelet configuration file. To enable the feature gate in kind, ensure that `kind` provides the minimum required Kubernetes version and enables the `SeccompDefault` feature in the kind configuration:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4nodes:  - role: control-plane    image: kindest/node:v1.28.0@sha256:9f3ff58f19dcf1a0611d11e8ac98
```

If the cluster is ready, then running a pod:

```
kubectl run --rm -it --restart=Never --image=alpine alpine -- sh
```

Should now have the default seccomp profile attached. This can be verified by using `docker exec` to run `crictl inspect` for the container on the kind worker:

```
docker exec -it kind-worker bash -c \
    'crictl inspect $(crictl ps --name=alpine -q) | jq .info.runtimeSpec.linux.seccomp'
```

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": ["SCMP_ARCH_X86_64", "SCMP_ARCH_X86", "SCMP_ARCH_X32"],
  "syscalls": [
    {
      "names": ["..."]
    }
  ]
}
```

What's next

You can learn more about Linux seccomp:

- A seccomp Overview
- Seccomp Security Profiles for Docker

Explore Termination Behavior for Pods And Their Endpoints

Once you connected your Application with Service following steps like those outlined in Connecting Applications with Services, you have a continuously running, replicated application, that is exposed on a network. This tutorial helps you look at the termination flow for Pods and to explore ways to implement graceful connection draining.

Termination process for Pods and their endpoints

There are often cases when you need to terminate a Pod - be it to upgrade or scale down. In order to improve application availability, it may be important to implement a proper active connections draining.

This tutorial explains the flow of Pod termination in connection with the corresponding endpoint state and removal by using a simple nginx web server to demonstrate the concept.

Example flow with endpoint termination

The following is the example flow described in the Termination of Pods document.

Let's say you have a Deployment containing a single `nginx` replica (say just for the sake of demonstration purposes) and a Service:

service/pod-with-graceful-termination.yaml Copy service/pod-with-graceful-termination.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deployment  labels:    app: nginxspec:  replicas: 1  selector:    matchLabels:      app: ng
```

service/explore-graceful-termination-nginx.yaml Copy service/explore-graceful-termination-nginx.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: nginx-servicespec:  selector:    app: nginx  ports:    - protocol: TCP      port: 80      targetPort
```

Now create the Deployment Pod and Service using the above files:

```
kubectl apply -f pod-with-graceful-termination.yaml
kubectl apply -f explore-graceful-termination-nginx.yaml
```

Once the Pod and Service are running, you can get the name of any associated EndpointSlices:

```
kubectl get endpointslice
```

The output is similar to this:

```
NAME                  ADDRESSTYPE   PORTS   ENDPOINTS                 AGE
nginx-service-6tjbr   IPv4          80      10.12.1.199,10.12.1.201   22m
```

You can see its status, and validate that there is one endpoint registered:

```
kubectl get endpointslices -o json -l kubernetes.io/service-name=nginx-service
```

The output is similar to this:

```
{
    "addressType": "IPv4",
    "apiVersion": "discovery.k8s.io/v1",
    "endpoints": [
        {
            "addresses": [
                "10.12.1.201"
            ],
            "conditions": {
                "ready": true,
                "serving": true,
                "terminating": false
```

Now let's terminate the Pod and validate that the Pod is being terminated respecting the graceful termination period configuration:

```
kubectl delete pod nginx-deployment-7768647bf9-b4b9s
```

All pods:

```
kubectl get pods
```

The output is similar to this:

```
NAME                                READY   STATUS        RESTARTS      AGE
nginx-deployment-7768647bf9-b4b9s   1/1     Terminating   0             4m1s
nginx-deployment-7768647bf9-rkxlw   1/1     Running       0             8s
```

You can see that the new pod got scheduled.

While the new endpoint is being created for the new Pod, the old endpoint is still around in the terminating state:

```
kubectl get endpointslice -o json nginx-service-6tjbr
```

The output is similar to this:

```
{
    "addressType": "IPv4",
    "apiVersion": "discovery.k8s.io/v1",
    "endpoints": [
        {
            "addresses": [
                "10.12.1.201"
            ],
            "conditions": {
                "ready": false,
                "serving": true,
                "terminating": true
            },
            "nodeName": "gke-main-default-pool-dca1511c-d17b",
            "targetRef": {
                "kind": "Pod",
                "name": "nginx-deployment-7768647bf9-b4b9s",
                "namespace": "default",
                "uid": "66fa831c-7eb2-407f-bd2c-f96dfe841478"
            },
            "zone": "us-central1-c"
        },
        {
            "addresses": [
                "10.12.1.202"
            ],
            "conditions": {
                "ready": true,
                "serving": true,
                "terminating": false
            },
            "nodeName": "gke-main-default-pool-dca1511c-d17b",
            "targetRef": {
```

```
            "kind": "Pod",
            "name": "nginx-deployment-7768647bf9-rkxlw",
            "namespace": "default",
            "uid": "722b1cbe-dcd7-4ed4-8928-4a4d0e2bbe35"
        },
        "zone": "us-central1-c"
```


This allows applications to communicate their state during termination and clients (such as load balancers) to implement connection draining functionality. These clients may detect terminating endpoints and implement a special logic for them.

In Kubernetes, endpoints that are terminating always have their `ready` status set as `false`. This needs to happen for backward compatibility, so existing load balancers will not use it for regular traffic. If traffic draining on terminating pod is needed, the actual readiness can be checked as a condition `serving`.

When Pod is deleted, the old endpoint will also be deleted.

What's next

- Learn how to Connect Applications with Services
- Learn more about Using a Service to Access an Application in a Cluster
- Learn more about Connecting a Front End to a Back End Using a Service
- Learn more about Creating an External Load Balancer

Stateless Applications

Exposing an External IP Address to Access an Application in a Cluster

Example: Deploying PHP Guestbook application with Redis

Services

Connecting Applications with Services

Using Source IP

Explore Termination Behavior for Pods And Their Endpoints

Security

Security is an important concern for most organizations and people who run Kubernetes clusters. You can find a basic security checklist elsewhere in the Kubernetes documentation.

To learn how to deploy and manage security aspects of Kubernetes, you can follow the tutorials in this section.

Apply Pod Security Standards at the Cluster Level

Apply Pod Security Standards at the Namespace Level

Restrict a Container's Access to Resources with AppArmor

Restrict a Container's Syscalls with seccomp

Restrict a Container's Access to Resources with AppArmor

FEATURE STATE: `Kubernetes v1.31 [stable]` (enabled by default: true)

This page shows you how to load AppArmor profiles on your nodes and enforce those profiles in Pods. To learn more about how Kubernetes can confine Pods using AppArmor, see Linux kernel security constraints for Pods and containers.

Objectives

- See an example of how to load a profile on a Node
- Learn how to enforce the profile on a Pod
- Learn how to check that the profile is loaded
- See what happens when a profile is violated
- See what happens when a profile cannot be loaded

Before you begin

AppArmor is an optional kernel module and Kubernetes feature, so verify it is supported on your Nodes before proceeding:

1. AppArmor kernel module is enabled -- For the Linux kernel to enforce an AppArmor profile, the AppArmor kernel module must be installed and enabled. Several distributions enable the module by default, such as Ubuntu and SUSE, and many others provide optional support. To check whether the module is enabled, check the `/sys/module/apparmor/parameters/enabled` file:

   ```
   cat /sys/module/apparmor/parameters/enabled
   Y
   ```

   The kubelet verifies that AppArmor is enabled on the host before admitting a pod with AppArmor explicitly configured.

2. Container runtime supports AppArmor -- All common Kubernetes-supported container runtimes should support AppArmor, including containerd and CRI-O. Please refer to the corresponding runtime documentation and verify that the cluster fulfills the requirements to use AppArmor.

3. Profile is loaded -- AppArmor is applied to a Pod by specifying an AppArmor profile that each container should be run with. If any of the specified profiles are not loaded in the kernel, the kubelet will reject the Pod. You can view which profiles are loaded on a node by checking the `/sys/kernel/security/apparmor/profiles` file. For example:

   ```
   ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/security/apparmor/profiles | sort"
   ```

   ```
   apparmor-test-deny-write (enforce)
   apparmor-test-audit-write (enforce)
   docker-default (enforce)
   k8s-nginx (enforce)
   ```

   For more details on loading profiles on nodes, see Setting up nodes with profiles.

Securing a Pod

Note:

Prior to Kubernetes v1.30, AppArmor was specified through annotations. Use the documentation version selector to view the documentation with this deprecated API.

AppArmor profiles can be specified at the pod level or container level. The container AppArmor profile takes precedence over the pod profile.

```
securityContext:
  appArmorProfile:
    type: <profile_type>
```

Where `<profile_type>` is one of:

- `RuntimeDefault` to use the runtime's default profile
- `Localhost` to use a profile loaded on the host (see below)
- `Unconfined` to run without AppArmor

See Specifying AppArmor Confinement for full details on the AppArmor profile API.

To verify that the profile was applied, you can check that the container's root process is running with the correct profile by examining its proc attr:

```
kubectl exec <pod_name> -- cat /proc/1/attr/current
```

The output should look something like this:

```
cri-containerd.apparmor.d (enforce)
```

Example

This example assumes you have already set up a cluster with AppArmor support.

First, load the profile you want to use onto your Nodes. This profile blocks all file write operations:

```
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>

  file,

  # Deny all file writes.
  deny /** w,
}
```

The profile needs to be loaded onto all nodes, since you don't know where the pod will be scheduled. For this example you can use SSH to install the profiles, but other approaches are discussed in Setting up nodes with profiles.

```
# This example assumes that node names match host names, and are reachable via SSH.
NODES=($( kubectl get node -o jsonpath='{.items[*].status.addresses[?(.type == "Hostname")].address}' ))

for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_parser -q <<EOF
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
  #include <abstractions/base>
```

```
  file,

  # Deny all file writes.
  deny /** w,
}
EOF'
done
```

Next, run a simple "Hello AppArmor" Pod with the deny-write profile:

pods/security/hello-apparmor.yaml  Copy pods/security/hello-apparmor.yaml to clipboard

```
apiVersion: v1
kind: Podmetadata:  name: hello-apparmorspec:  securityContext:    appArmorProfile:      type: Localhost      localhostProfile: k8
```

```
kubectl create -f hello-apparmor.yaml
```

You can verify that the container is actually running with that profile by checking `/proc/1/attr/current`:

```
kubectl exec hello-apparmor -- cat /proc/1/attr/current
```

The output should be:

```
k8s-apparmor-example-deny-write (enforce)
```

Finally, you can see what happens if you violate the profile by writing to a file:

```
kubectl exec hello-apparmor -- touch /tmp/test
```

```
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero exit code: Error executing in Docker Container: 1
```

To wrap up, see what happens if you try to specify a profile that hasn't been loaded:

```
kubectl create -f /dev/stdin <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor-2
spec:
  securityContext:
    appArmorProfile:
      type: Localhost
      localhostProfile: k8s-apparmor-example-allow-write
  containers:
  - name: hello
    image: busybox:1.28
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
EOF
```

```
pod/hello-apparmor-2 created
```

Although the Pod was created successfully, further examination will show that it is stuck in pending:

```
kubectl describe pod hello-apparmor-2
```

```
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/10.128.0.27
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kubernetes.io/hello=localhost/k8s-apparmor-example-allow-write
Status:        Pending
...
Events:
  Type     Reason     Age              From               Message
  ----     ------     ----             ----               -------
  Normal   Scheduled  10s              default-scheduler  Successfully assigned default/hello-apparmor to gke-test-default-pool-23
  Normal   Pulled     8s               kubelet            Successfully pulled image "busybox:1.28" in 370.157088ms (370.172701ms i
  Normal   Pulling    7s (x2 over 9s)  kubelet            Pulling image "busybox:1.28"
  Warning  Failed     7s (x2 over 8s)  kubelet            Error: failed to get container spec opts: failed to generate apparmor spe
  Normal   Pulled     7s               kubelet            Successfully pulled image "busybox:1.28" in 90.980331ms (91.005869ms inc
```

An Event provides the error message with the reason, the specific wording is runtime-dependent:

```
  Warning  Failed     7s (x2 over 8s)  kubelet            Error: failed to get container spec opts: failed to generate apparmor spe
```

Administration

Setting up Nodes with profiles

Kubernetes 1.34 does not provide any built-in mechanisms for loading AppArmor profiles onto Nodes. Profiles can be loaded through custom infrastructure or tools like the Kubernetes Security Profiles Operator.

The scheduler is not aware of which profiles are loaded onto which Node, so the full set of profiles must be loaded onto every Node. An alternative approach is to add a Node label for each profile (or class of profiles) on the Node, and use a node selector to ensure the Pod is run on a Node with the required profile.

Authoring Profiles

Getting AppArmor profiles specified correctly can be a tricky business. Fortunately there are some tools to help with that:

- `aa-genprof` and `aa-logprof` generate profile rules by monitoring an application's activity and logs, and admitting the actions it takes. Further instructions are provided by the AppArmor documentation.
- bane is an AppArmor profile generator for Docker that uses a simplified profile language.

To debug problems with AppArmor, you can check the system logs to see what, specifically, was denied. AppArmor logs verbose messages to `dmesg`, and errors can usually be found in the system logs or through `journalctl`. More information is provided in AppArmor failures.

Specifying AppArmor confinement

Caution:

Prior to Kubernetes v1.30, AppArmor was specified through annotations. Use the documentation version selector to view the documentation with this deprecated API.

AppArmor profile within security context

You can specify the `appArmorProfile` on either a container's `securityContext` or on a Pod's `securityContext`. If the profile is set at the pod level, it will be used as the default profile for all containers in the pod (including init, sidecar, and ephemeral containers). If both a pod & container AppArmor profile are set, the container's profile will be used.

An AppArmor profile has 2 fields:

`type` *(required)* - indicates which kind of AppArmor profile will be applied. Valid options are:

`Localhost`
: a profile pre-loaded on the node (specified by `localhostProfile`).

`RuntimeDefault`
: the container runtime's default profile.

`Unconfined`
: no AppArmor enforcement.

`localhostProfile` - The name of a profile loaded on the node that should be used. The profile must be preconfigured on the node to work. This option must be provided if and only if the `type` is `Localhost`.

What's next

Additional resources:

- Quick guide to the AppArmor profile language
- AppArmor core policy reference

Running ZooKeeper, A Distributed System Coordinator

This tutorial demonstrates running Apache Zookeeper on Kubernetes using StatefulSets, PodDisruptionBudgets, and PodAntiAffinity.

Before you begin

Before starting this tutorial, you should be familiar with the following Kubernetes concepts:

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- StatefulSets
- PodDisruptionBudgets
- PodAntiAffinity
- kubectl CLI

You must have a cluster with at least four nodes, and each node requires at least 2 CPUs and 4 GiB of memory. In this tutorial you will cordon and drain the cluster's nodes. **This means that the cluster will terminate and evict all Pods on its nodes, and the nodes will temporarily become unschedulable.** You should use a dedicated cluster for this tutorial, or you should ensure that the disruption you cause will not interfere with other tenants.

This tutorial assumes that you have configured your cluster to dynamically provision PersistentVolumes. If your cluster is not configured to do so, you will have to manually provision three 20 GiB volumes before starting this tutorial.

Objectives

After this tutorial, you will know the following.

- How to deploy a ZooKeeper ensemble using StatefulSet.
- How to consistently configure the ensemble.
- How to spread the deployment of ZooKeeper servers in the ensemble.
- How to use PodDisruptionBudgets to ensure service availability during planned maintenance.

ZooKeeper

Apache ZooKeeper is a distributed, open-source coordination service for distributed applications. ZooKeeper allows you to read, write, and observe updates to data. Data are organized in a file system like hierarchy and replicated to all ZooKeeper servers in the ensemble (a set of ZooKeeper servers). All operations on data are atomic and sequentially consistent. ZooKeeper ensures this by using the Zab consensus protocol to replicate a state machine across all servers in the ensemble.

The ensemble uses the Zab protocol to elect a leader, and the ensemble cannot write data until that election is complete. Once complete, the ensemble uses Zab to ensure that it replicates all writes to a quorum before it acknowledges and makes them visible to clients. Without respect to weighted quorums, a quorum is a majority component of the ensemble containing the current leader. For instance, if the ensemble has three servers, a component that contains the leader and one other server constitutes a quorum. If the ensemble can not achieve a quorum, the ensemble cannot write data.

ZooKeeper servers keep their entire state machine in memory, and write every mutation to a durable WAL (Write Ahead Log) on storage media. When a server crashes, it can recover its previous state by replaying the WAL. To prevent the WAL from growing without bound, ZooKeeper servers will periodically snapshot them in memory state to storage media. These snapshots can be loaded directly into memory, and all WAL entries that preceded the snapshot may be discarded.

Creating a ZooKeeper ensemble

The manifest below contains a Headless Service, a Service, a PodDisruptionBudget, and a StatefulSet.

application/zookeeper/zookeeper.yaml Copy application/zookeeper/zookeeper.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: zk-hs  labels:    app: zkspec:  ports:  - port: 2888    name: server  - port: 3888    name: leader-e
```

Open a terminal, and use the kubectl apply command to create the manifest.

```
kubectl apply -f https://k8s.io/examples/application/zookeeper/zookeeper.yaml
```

This creates the zk-hs Headless Service, the zk-cs Service, the zk-pdb PodDisruptionBudget, and the zk StatefulSet.

```
service/zk-hs created
service/zk-cs created
poddisruptionbudget.policy/zk-pdb created
statefulset.apps/zk created
```

Use kubectl get to watch the StatefulSet controller create the StatefulSet's Pods.

```
kubectl get pods -w -l app=zk
```

Once the zk-2 Pod is Running and Ready, use CTRL-C to terminate kubectl.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

The StatefulSet controller creates three Pods, and each Pod has a container with a ZooKeeper server.

Facilitating leader election

Because there is no terminating algorithm for electing a leader in an anonymous network, Zab requires explicit membership configuration to perform leader election. Each server in the ensemble needs to have a unique identifier, all servers need to know the global set of identifiers, and each identifier needs to be associated with a network address.

Use kubectl exec to get the hostnames of the Pods in the zk StatefulSet.

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname; done
```

The StatefulSet controller provides each Pod with a unique hostname based on its ordinal index. The hostnames take the form of <statefulset name>-<ordinal index>. Because the replicas field of the zk StatefulSet is set to 3, the Set's controller creates three Pods with their hostnames set to zk-0, zk-1, and zk-2.

```
zk-0
zk-1
zk-2
```

The servers in a ZooKeeper ensemble use natural numbers as unique identifiers, and store each server's identifier in a file called myid in the server's data directory.

To examine the contents of the myid file for each server use the following command.

```
for i in 0 1 2; do echo "myid zk-$i";kubectl exec zk-$i -- cat /var/lib/zookeeper/data/myid; done
```

Because the identifiers are natural numbers and the ordinal indices are non-negative integers, you can generate an identifier by adding 1 to the ordinal.

```
myid zk-0
1
myid zk-1
2
myid zk-2
3
```

To get the Fully Qualified Domain Name (FQDN) of each Pod in the `zk` StatefulSet use the following command.

```
for i in 0 1 2; do kubectl exec zk-$i -- hostname -f; done
```

The `zk-hs` Service creates a domain for all of the Pods, `zk-hs.default.svc.cluster.local`.

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

The A records in Kubernetes DNS resolve the FQDNs to the Pods' IP addresses. If Kubernetes reschedules the Pods, it will update the A records with the Pods' new IP addresses, but the A records names will not change.

ZooKeeper stores its application configuration in a file named `zoo.cfg`. Use `kubectl exec` to view the contents of the `zoo.cfg` file in the `zk-0` Pod.

```
kubectl exec zk-0 -- cat /opt/zookeeper/conf/zoo.cfg
```

In the `server.1`, `server.2`, and `server.3` properties at the bottom of the file, the `1`, `2`, and `3` correspond to the identifiers in the ZooKeeper servers' `myid` files. They are set to the FQDNs for the Pods in the `zk` StatefulSet.

```
clientPort=2181
dataDir=/var/lib/zookeeper/data
dataLogDir=/var/lib/zookeeper/log
tickTime=2000
initLimit=10
syncLimit=2000
maxClientCnxns=60
minSessionTimeout= 4000
maxSessionTimeout= 40000
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
server.1=zk-0.zk-hs.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:3888
```

Achieving consensus

Consensus protocols require that the identifiers of each participant be unique. No two participants in the Zab protocol should claim the same unique identifier. This is necessary to allow the processes in the system to agree on which processes have committed which data. If two Pods are launched with the same ordinal, two ZooKeeper servers would both identify themselves as the same server.

```
kubectl get pods -w -l app=zk
```

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

The A records for each Pod are entered when the Pod becomes Ready. Therefore, the FQDNs of the ZooKeeper servers will resolve to a single endpoint, and that endpoint will be the unique ZooKeeper server claiming the identity configured in its `myid` file.

```
zk-0.zk-hs.default.svc.cluster.local
zk-1.zk-hs.default.svc.cluster.local
zk-2.zk-hs.default.svc.cluster.local
```

This ensures that the `servers` properties in the ZooKeepers' `zoo.cfg` files represents a correctly configured ensemble.

```
server.1=zk-0.zk-hs.default.svc.cluster.local:2888:3888
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:3888
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:3888
```

When the servers use the Zab protocol to attempt to commit a value, they will either achieve consensus and commit the value (if leader election has succeeded and at least two of the Pods are Running and Ready), or they will fail to do so (if either of the conditions are not met). No state will arise where one server acknowledges a write on behalf of another.

Sanity testing the ensemble

The most basic sanity test is to write data to one ZooKeeper server and to read the data from another.

The command below executes the `zkCli.sh` script to write `world` to the path `/hello` on the `zk-0` Pod in the ensemble.

```
kubectl exec zk-0 -- zkCli.sh create /hello world
```

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
Created /hello
```

To get the data from the `zk-1` Pod use the following command.

```
kubectl exec zk-1 -- zkCli.sh get /hello
```

The data that you created on `zk-0` is available on all the servers in the ensemble.

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Providing durable storage

As mentioned in the ZooKeeper Basics section, ZooKeeper commits all entries to a durable WAL, and periodically writes snapshots in memory state, to storage media. Using WALs to provide durability is a common technique for applications that use consensus protocols to achieve a replicated state machine.

Use the `kubectl delete` command to delete the `zk` StatefulSet.

```
kubectl delete statefulset zk
```

```
statefulset.apps "zk" deleted
```

Watch the termination of the Pods in the StatefulSet.

```
kubectl get pods -w -l app=zk
```

When `zk-0` if fully terminated, use `CTRL-C` to terminate kubectl.

```
zk-2      1/1       Terminating   0         9m
zk-0      1/1       Terminating   0         11m
zk-1      1/1       Terminating   0         10m
zk-2      0/1       Terminating   0         9m
zk-2      0/1       Terminating   0         9m
zk-2      0/1       Terminating   0         9m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-1      0/1       Terminating   0         10m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
zk-0      0/1       Terminating   0         11m
```

Reapply the manifest in `zookeeper.yaml`.

```
kubectl apply -f https://k8s.io/examples/application/zookeeper/zookeeper.yaml
```

This creates the `zk` StatefulSet object, but the other API objects in the manifest are not modified because they already exist.

Watch the StatefulSet controller recreate the StatefulSet's Pods.

```
kubectl get pods -w -l app=zk
```

Once the `zk-2` Pod is Running and Ready, use `CTRL-C` to terminate kubectl.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Pending   0          0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         19s
zk-0      1/1       Running   0         40s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       ContainerCreating   0         0s
zk-1      0/1       Running   0         18s
zk-1      1/1       Running   0         40s
zk-2      0/1       Pending   0         0s
zk-2      0/1       Pending   0         0s
zk-2      0/1       ContainerCreating   0         0s
zk-2      0/1       Running   0         19s
zk-2      1/1       Running   0         40s
```

Use the command below to get the value you entered during the sanity test, from the `zk-2` Pod.

```
kubectl exec zk-2 zkCli.sh get /hello
```

Even though you terminated and recreated all of the Pods in the `zk` StatefulSet, the ensemble still serves the original value.

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

The `volumeClaimTemplates` field of the `zk` StatefulSet's `spec` specifies a PersistentVolume provisioned for each Pod.

```
volumeClaimTemplates:
  - metadata:
      name: datadir
      annotations:
        volume.alpha.kubernetes.io/storage-class: anything
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 20Gi
```

The `StatefulSet` controller generates a `PersistentVolumeClaim` for each Pod in the `StatefulSet`.

Use the following command to get the `StatefulSet`'s `PersistentVolumeClaims`.

```
kubectl get pvc -l app=zk
```

When the `StatefulSet` recreated its Pods, it remounts the Pods' PersistentVolumes.

```
NAME           STATUS    VOLUME                                     CAPACITY   ACCESSMODES   AGE
datadir-zk-0   Bound     pvc-bed742cd-bcb1-11e6-994f-42010a800002   20Gi       RWO           1h
datadir-zk-1   Bound     pvc-bedd27d2-bcb1-11e6-994f-42010a800002   20Gi       RWO           1h
datadir-zk-2   Bound     pvc-bee0817e-bcb1-11e6-994f-42010a800002   20Gi       RWO           1h
```

The `volumeMounts` section of the `StatefulSet`'s container `template` mounts the PersistentVolumes in the ZooKeeper servers' data directories.

```
volumeMounts:
- name: datadir  mountPath: /var/lib/zookeeper
```

When a Pod in the `zk` `StatefulSet` is (re)scheduled, it will always have the same `PersistentVolume` mounted to the ZooKeeper server's data directory. Even when the Pods are rescheduled, all the writes made to the ZooKeeper servers' WALs, and all their snapshots, remain durable.

Ensuring consistent configuration

As noted in the Facilitating Leader Election and Achieving Consensus sections, the servers in a ZooKeeper ensemble require consistent configuration to elect a leader and form a quorum. They also require consistent configuration of the Zab protocol in order for the protocol to work correctly over a network. In our example we achieve consistent configuration by embedding the configuration directly into the manifest.

Get the `zk` StatefulSet.

```
kubectl get sts zk -o yaml

…
command:
      - sh
      - -c
      - "start-zookeeper \
        --servers=3 \
        --data_dir=/var/lib/zookeeper/data \
        --data_log_dir=/var/lib/zookeeper/data/log \
        --conf_dir=/opt/zookeeper/conf \
        --client_port=2181 \
        --election_port=3888 \
        --server_port=2888 \
        --tick_time=2000 \
        --init_limit=10 \
        --sync_limit=5 \
        --heap=512M \
        --max_client_cnxns=60 \
        --snap_retain_count=3 \
        --purge_interval=12 \
        --max_session_timeout=40000 \
        --min_session_timeout=4000 \
        --log_level=INFO"
…
```

The command used to start the ZooKeeper servers passed the configuration as command line parameter. You can also use environment variables to pass configuration to the ensemble.

Configuring logging

One of the files generated by the `zkGenConfig.sh` script controls ZooKeeper's logging. ZooKeeper uses Log4j, and, by default, it uses a time and size based rolling file appender for its logging configuration.

Use the command below to get the logging configuration from one of Pods in the `zk StatefulSet`.

```
kubectl exec zk-0 cat /usr/etc/zookeeper/log4j.properties
```

The logging configuration below will cause the ZooKeeper process to write all of its logs to the standard output file stream.

```
zookeeper.root.logger=CONSOLE
zookeeper.console.threshold=INFO
log4j.rootLogger=${zookeeper.root.logger}
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=${zookeeper.console.threshold}
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%d{ISO8601} [myid:%X{myid}] - %-5p [%t:%C{1}@%L] - %m%n
```

This is the simplest possible way to safely log inside the container. Because the applications write logs to standard out, Kubernetes will handle log rotation for you. Kubernetes also implements a sane retention policy that ensures application logs written to standard out and standard error do not exhaust local storage media.

Use `kubectl logs` to retrieve the last 20 log lines from one of the Pods.

```
kubectl logs zk-0 --tail 20
```

You can view application logs written to standard out or standard error using `kubectl logs` and from the Kubernetes Dashboard.

```
2016-12-06 19:34:16,236 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:16,237 [myid:1] - INFO  [Thread-1136:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52740 (
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:26,156 [myid:1] - INFO  [Thread-1137:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52749 (
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:26,226 [myid:1] - INFO  [Thread-1138:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52750 (
2016-12-06 19:34:36,151 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:36,152 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:36,152 [myid:1] - INFO  [Thread-1139:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52760 (
2016-12-06 19:34:36,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:36,231 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:36,231 [myid:1] - INFO  [Thread-1140:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52761 (
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:46,149 [myid:1] - INFO  [Thread-1141:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52767 (
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted socket co
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok command f
2016-12-06 19:34:46,230 [myid:1] - INFO  [Thread-1142:NIOServerCnxn@1008] - Closed socket connection for client /127.0.0.1:52768 (
```

Kubernetes integrates with many logging solutions. You can choose a logging solution that best fits your cluster and applications. For cluster-level logging and aggregation, consider deploying a sidecar container to rotate and ship your logs.

Configuring a non-privileged user

The best practices to allow an application to run as a privileged user inside of a container are a matter of debate. If your organization requires that applications run as a non-privileged user you can use a SecurityContext to control the user that the entry point runs as.

The `zk StatefulSet`'s Pod `template` contains a `SecurityContext`.

```
securityContext:
  runAsUser: 1000
  fsGroup: 1000
```

In the Pods' containers, UID 1000 corresponds to the zookeeper user and GID 1000 corresponds to the zookeeper group.

Get the ZooKeeper process information from the `zk-0` Pod.

```
kubectl exec zk-0 -- ps -elf
```

As the `runAsUser` field of the `securityContext` object is set to 1000, instead of running as root, the ZooKeeper process runs as the zookeeper user.

```
F S UID        PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME TTY          TIME CMD
4 S zookeep+     1     0  0  80   0 -  1127 -      20:46 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.sh start-foreground
0 S zookeep+    27     1  0  80   0 - 1155556 -    20:46 ?        00:00:19 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -Dzookeeper.
```

By default, when the Pod's PersistentVolumes is mounted to the ZooKeeper server's data directory, it is only accessible by the root user. This configuration prevents the ZooKeeper process from writing to its WAL and storing its snapshots.

Use the command below to get the file permissions of the ZooKeeper data directory on the `zk-0` Pod.

```
kubectl exec -ti zk-0 -- ls -ld /var/lib/zookeeper/data
```

Because the `fsGroup` field of the `securityContext` object is set to 1000, the ownership of the Pods' PersistentVolumes is set to the zookeeper group, and the ZooKeeper process is able to read and write its data.

```
drwxr-sr-x 3 zookeeper zookeeper 4096 Dec  5 20:45 /var/lib/zookeeper/data
```

Managing the ZooKeeper process

The ZooKeeper documentation mentions that "You will want to have a supervisory process that manages each of your ZooKeeper server processes (JVM)." Utilizing a watchdog (supervisory process) to restart failed processes in a distributed system is a common pattern. When deploying an application in Kubernetes, rather than using an external utility as a supervisory process, you should use Kubernetes as the watchdog for your application.

Updating the ensemble

The `zk` `StatefulSet` is configured to use the `RollingUpdate` update strategy.

You can use `kubectl patch` to update the number of `cpus` allocated to the servers.

```
kubectl patch sts zk --type='json' -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/resources/requests/cpu", "valu
```

```
statefulset.apps/zk patched
```

Use `kubectl rollout status` to watch the status of the update.

```
kubectl rollout status sts/zk
```

```
waiting for statefulset rolling update to complete 0 pods at revision zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 1 pods at revision zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 2 pods at revision zk-5db4499664...
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
statefulset rolling update complete 3 pods at revision zk-5db4499664...
```

This terminates the Pods, one at a time, in reverse ordinal order, and recreates them with the new configuration. This ensures that quorum is maintained during a rolling update.

Use the `kubectl rollout history` command to view a history or previous configurations.

```
kubectl rollout history sts/zk
```

The output is similar to this:

```
statefulsets "zk"
REVISION
1
2
```

Use the `kubectl rollout undo` command to roll back the modification.

```
kubectl rollout undo sts/zk
```

The output is similar to this:

```
statefulset.apps/zk rolled back
```

Handling process failure

Restart Policies control how Kubernetes handles process failures for the entry point of the container in a Pod. For Pods in a `StatefulSet`, the only appropriate `RestartPolicy` is Always, and this is the default value. For stateful applications you should **never** override the default policy.

Use the following command to examine the process tree for the ZooKeeper server running in the `zk-0` Pod.

```
kubectl exec zk-0 -- ps -ef
```

The command used as the container's entry point has PID 1, and the ZooKeeper process, a child of the entry point, has PID 27.

```
UID        PID  PPID  C STIME TTY          TIME CMD
zookeep+     1     0  0 15:03 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.sh start-foreground
zookeep+    27     1  0 15:03 ?        00:00:03 /usr/lib/jvm/java-8-openjdk-amd64/bin/java -Dzookeeper.log.dir=/var/log/zookeeper
```

In another terminal watch the Pods in the `zk` `StatefulSet` with the following command.

```
kubectl get pod -w -l app=zk
```

In another terminal, terminate the ZooKeeper process in Pod `zk-0` with the following command.

```
kubectl exec zk-0 -- pkill java
```

The termination of the ZooKeeper process caused its parent process to terminate. Because the `RestartPolicy` of the container is Always, it restarted the parent process.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   0          21m
zk-1      1/1       Running   0          20m
zk-2      1/1       Running   0          19m
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Error     0          29m
zk-0      0/1       Running   1         29m
zk-0      1/1       Running   1         29m
```

If your application uses a script (such as `zkServer.sh`) to launch the process that implements the application's business logic, the script must terminate with the child process. This ensures that Kubernetes will restart the application's container when the process implementing the application's business logic fails.

Testing for liveness

Configuring your application to restart failed processes is not enough to keep a distributed system healthy. There are scenarios where a system's processes can be both alive and unresponsive, or otherwise unhealthy. You should use liveness probes to notify Kubernetes that your application's processes are unhealthy and it should restart them.

The Pod `template` for the `zk` `StatefulSet` specifies a liveness probe.

```
  livenessProbe:
    exec:
      command:
      - sh
      - -c
      - "zookeeper-ready 2181"
    initialDelaySeconds: 15
    timeoutSeconds: 5
```

The probe calls a bash script that uses the ZooKeeper `ruok` four letter word to test the server's health.

```
OK=$(echo ruok | nc 127.0.0.1 $1)
if [ "$OK" == "imok" ]; then
    exit 0
else
    exit 1
fi
```

In one terminal window, use the following command to watch the Pods in the `zk` StatefulSet.

```
kubectl get pod -w -l app=zk
```

In another window, using the following command to delete the `zookeeper-ready` script from the file system of Pod `zk-0`.

```
kubectl exec zk-0 -- rm /opt/zookeeper/bin/zookeeper-ready
```

When the liveness probe for the ZooKeeper process fails, Kubernetes will automatically restart the process for you, ensuring that unhealthy processes in the ensemble are restarted.

```
kubectl get pod -w -l app=zk
```

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   0          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS    RESTARTS   AGE
zk-0      0/1       Running   0          1h
zk-0      0/1       Running   1         1h
zk-0      1/1       Running   1         1h
```

Testing for readiness

Readiness is not the same as liveness. If a process is alive, it is scheduled and healthy. If a process is ready, it is able to process input. Liveness is a necessary, but not sufficient, condition for readiness. There are cases, particularly during initialization and termination, when a process can be alive but not ready.

If you specify a readiness probe, Kubernetes will ensure that your application's processes will not receive network traffic until their readiness checks pass.

For a ZooKeeper server, liveness implies readiness. Therefore, the readiness probe from the `zookeeper.yaml` manifest is identical to the liveness probe.

```
  readinessProbe:
    exec:
      command:
      - sh
      - -c
      - "zookeeper-ready 2181"
    initialDelaySeconds: 15
    timeoutSeconds: 5
```

Even though the liveness and readiness probes are identical, it is important to specify both. This ensures that only healthy servers in the ZooKeeper ensemble receive network traffic.

Tolerating Node failure

ZooKeeper needs a quorum of servers to successfully commit mutations to data. For a three server ensemble, two servers must be healthy for writes to succeed. In quorum based systems, members are deployed across failure domains to ensure availability. To avoid an outage, due to the loss of an individual machine, best practices preclude co-locating multiple instances of the application on the same machine.

By default, Kubernetes may co-locate Pods in a `StatefulSet` on the same node. For the three server ensemble you created, if two servers are on the same node, and that node fails, the clients of your ZooKeeper service will experience an outage until at least one of the Pods can be rescheduled.

You should always provision additional capacity to allow the processes of critical systems to be rescheduled in the event of node failures. If you do so, then the outage will only last until the Kubernetes scheduler reschedules one of the ZooKeeper servers. However, if you want your service to tolerate node failures with no downtime, you should set `podAntiAffinity`.

Use the command below to get the nodes for Pods in the `zk` `StatefulSet`.

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo ""; done
```

All of the Pods in the `zk` `StatefulSet` are deployed on different nodes.

```
kubernetes-node-cxpk
kubernetes-node-a5aq
kubernetes-node-2g2d
```

This is because the Pods in the `zk` `StatefulSet` have a `PodAntiAffinity` specified.

```
affinity:
  podAntiAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
            - key: "app"
              operator: In
              values:
                - zk
        topologyKey: "kubernetes.io/hostname"
```

The `requiredDuringSchedulingIgnoredDuringExecution` field tells the Kubernetes Scheduler that it should never co-locate two Pods which have `app` label as `zk` in the domain defined by the `topologyKey`. The `topologyKey` `kubernetes.io/hostname` indicates that the domain is an individual node. Using different rules, labels, and selectors, you can extend this technique to spread your ensemble across physical, network, and power failure domains.

Surviving maintenance

In this section you will cordon and drain nodes. If you are using this tutorial on a shared cluster, be sure that this will not adversely affect other tenants.

The previous section showed you how to spread your Pods across nodes to survive unplanned node failures, but you also need to plan for temporary node failures that occur due to planned maintenance.

Use this command to get the nodes in your cluster.

```
kubectl get nodes
```

This tutorial assumes a cluster with at least four nodes. If the cluster has more than four, use `kubectl cordon` to cordon all but four nodes. Constraining to four nodes will ensure Kubernetes encounters affinity and PodDisruptionBudget constraints when scheduling zookeeper Pods in the following maintenance simulation.

```
kubectl cordon <node-name>
```

Use this command to get the `zk-pdb` `PodDisruptionBudget`.

```
kubectl get pdb zk-pdb
```

The `max-unavailable` field indicates to Kubernetes that at most one Pod from `zk` `StatefulSet` can be unavailable at any time.

```
NAME      MIN-AVAILABLE   MAX-UNAVAILABLE   ALLOWED-DISRUPTIONS   AGE
zk-pdb    N/A             1                 1
```

In one terminal, use this command to watch the Pods in the `zk` `StatefulSet`.

```
kubectl get pods -w -l app=zk
```

In another terminal, use this command to get the nodes that the Pods are currently scheduled on.

```
for i in 0 1 2; do kubectl get pod zk-$i --template {{.spec.nodeName}}; echo ""; done
```

The output is similar to this:

```
kubernetes-node-pb41
kubernetes-node-ixsl
kubernetes-node-i4c4
```

Use `kubectl drain` to cordon and drain the node on which the `zk-0` Pod is scheduled.

```
kubectl drain $(kubectl get pod zk-0 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-emptydir-data
```

The output is similar to this:

```
node "kubernetes-node-pb41" cordoned

WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-p
pod "zk-0" deleted
node "kubernetes-node-pb41" drained
```

As there are four nodes in your cluster, `kubectl drain`, succeeds and the `zk-0` is rescheduled to another node.

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
```

Keep watching the `StatefulSet`'s Pods in the first terminal and drain the node on which `zk-1` is scheduled.

```
kubectl drain $(kubectl get pod zk-1 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-emptydir-data
```

The output is similar to this:

```
"kubernetes-node-ixsl" cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-
pod "zk-1" deleted
node "kubernetes-node-ixsl" drained
```

The `zk-1` Pod cannot be scheduled because the `zk` `StatefulSet` contains a `PodAntiAffinity` rule preventing co-location of the Pods, and as only two nodes are schedulable, the Pod will remain in a Pending state.

```
kubectl get pods -w -l app=zk
```

The output is similar to this:

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
zk-1      1/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
```

Continue to watch the Pods of the StatefulSet, and drain the node on which `zk-2` is scheduled.

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-emptydir-data
```

The output is similar to this:

```
node "kubernetes-node-i4c4" cordoned

WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-
WARNING: Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog; Deleting pods not managed by ReplicationController, Rep
There are pending pods when an error occurred: Cannot evict pod as it would violate the pod's disruption budget.
pod/zk-2
```

Use `CTRL-C` to terminate kubectl.

You cannot drain the third node because evicting `zk-2` would violate `zk-budget`. However, the node will remain cordoned.

Use `zkCli.sh` to retrieve the value you entered during the sanity test from `zk-0`.

```
kubectl exec zk-0 zkCli.sh get /hello
```

The service is still available because its `PodDisruptionBudget` is respected.

```
WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x200000002
ctime = Wed Dec 07 00:08:59 UTC 2016
mZxid = 0x200000002
mtime = Wed Dec 07 00:08:59 UTC 2016
pZxid = 0x200000002
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Use [kubectl uncordon] to uncordon the first node.

```
kubectl uncordon kubernetes-node-pb41
```

The output is similar to this:

```
node "kubernetes-node-pb41" uncordoned
```

`zk-1` is rescheduled on this node. Wait until `zk-1` is Running and Ready.

```
kubectl get pods -w -l app=zk
```

The output is similar to this:

```
NAME      READY     STATUS    RESTARTS   AGE
zk-0      1/1       Running   2          1h
```

```
zk-1      1/1       Running   0          1h
zk-2      1/1       Running   0          1h
NAME      READY     STATUS        RESTARTS   AGE
zk-0      1/1       Terminating   2          2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Terminating   2         2h
zk-0      0/1       Pending   0         0s
zk-0      0/1       Pending   0         0s
zk-0      0/1       ContainerCreating   0         0s
zk-0      0/1       Running   0         51s
zk-0      1/1       Running   0         1m
zk-1      1/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Terminating   0         2h
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         0s
zk-1      0/1       Pending   0         12m
zk-1      0/1       ContainerCreating   0         12m
zk-1      0/1       Running   0         13m
zk-1      1/1       Running   0         13m
```

Attempt to drain the node on which `zk-2` is scheduled.

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeName}}) --ignore-daemonsets --force --delete-emptydir-data
```

The output is similar to this:

```
node "kubernetes-node-i4c4" already cordoned
WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-kubernetes-node-
pod "heapster-v1.2.0-2604621511-wht1r" deleted
pod "zk-2" deleted
node "kubernetes-node-i4c4" drained
```

This time `kubectl drain` succeeds.

Uncordon the second node to allow `zk-2` to be rescheduled.

```
kubectl uncordon kubernetes-node-ixsl
```

The output is similar to this:

```
node "kubernetes-node-ixsl" uncordoned
```

You can use `kubectl drain` in conjunction with `PodDisruptionBudgets` to ensure that your services remain available during maintenance. If drain is used to cordon nodes and evict pods prior to taking the node offline for maintenance, services that express a disruption budget will have that budget respected. You should always allocate additional capacity for critical services so that their Pods can be immediately rescheduled.

Cleaning up

- Use `kubectl uncordon` to uncordon all the nodes in your cluster.
- You must delete the persistent storage media for the PersistentVolumes used in this tutorial. Follow the necessary steps, based on your environment, storage configuration, and provisioning method, to ensure that all storage is reclaimed.

StatefulSet Basics

This tutorial provides an introduction to managing applications with StatefulSets. It demonstrates how to create, delete, scale, and update the Pods of StatefulSets.

Before you begin

Before you begin this tutorial, you should familiarize yourself with the following Kubernetes concepts:

- Pods
- Cluster DNS
- Headless Services
- PersistentVolumes
- The kubectl command line tool

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

You should configure `kubectl` to use a context that uses the `default` namespace. If you are using an existing cluster, make sure that it's OK to use that cluster's default namespace to practice. Ideally, practice in a cluster that doesn't run any real workloads.

It's also useful to read the concept page about StatefulSets.

Note:

This tutorial assumes that your cluster is configured to dynamically provision PersistentVolumes. You'll also need to have a default StorageClass. If your cluster is not configured to provision storage dynamically, you will have to manually provision two 1 GiB volumes prior to starting this tutorial and set up your cluster so that those PersistentVolumes map to the PersistentVolumeClaim templates that the StatefulSet defines.

Objectives

StatefulSets are intended to be used with stateful applications and distributed systems. However, the administration of stateful applications and distributed systems on Kubernetes is a broad, complex topic. In order to demonstrate the basic features of a StatefulSet, and not to conflate the former topic with the latter, you will deploy a simple web application using a StatefulSet.

After this tutorial, you will be familiar with the following.

- How to create a StatefulSet
- How a StatefulSet manages its Pods
- How to delete a StatefulSet
- How to scale a StatefulSet
- How to update a StatefulSet's Pods

Creating a StatefulSet

Begin by creating a StatefulSet (and the Service that it relies upon) using the example below. It is similar to the example presented in the StatefulSets concept. It creates a headless Service, `nginx`, to publish the IP addresses of Pods in the StatefulSet, `web`.

application/web/web.yaml Copy application/web/web.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: nginx  labels:    app: nginxspec:  ports:  - port: 80    name: web  clusterIP: None  selector:    ap
```

You will need to use at least two terminal windows. In the first terminal, use `kubectl get` to watch the creation of the StatefulSet's Pods.

```
# use this terminal to run commands that specify --watch
# end this watch when you are asked to start a new watch
kubectl get pods --watch -l app=nginx
```

In the second terminal, use `kubectl apply` to create the headless Service and StatefulSet:

```
kubectl apply -f https://k8s.io/examples/application/web/web.yaml
```

```
service/nginx created
statefulset.apps/web created
```

The command above creates two Pods, each running an NGINX webserver. Get the `nginx` Service...

```
kubectl get service nginx
```

```
NAME      TYPE         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     ClusterIP    None         <none>        80/TCP    12s
```

...then get the `web` StatefulSet, to verify that both were created successfully:

```
kubectl get statefulset web
```

```
NAME   READY   AGE
web    2/2     37s
```

Ordered Pod creation

A StatefulSet defaults to creating its Pods in a strict order.

For a StatefulSet with *n* replicas, when Pods are being deployed, they are created sequentially, ordered from *{0..n-1}*. Examine the output of the `kubectl get` command in the first terminal. Eventually, the output will look like the example below.

```
# Do not start a new watch;
# this should already be running
kubectl get pods --watch -l app=nginx
```

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0          0s
web-0     0/1       ContainerCreating   0          0s
web-0     1/1       Running   0          19s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0          0s
web-1     1/1       Running   0          18s
```

Notice that the `web-1` Pod is not launched until the `web-0` Pod is *Running* (see Pod Phase) and *Ready* (see `type` in Pod Conditions).

Later in this tutorial you will practice parallel startup.

Note:

To configure the integer ordinal assigned to each Pod in a StatefulSet, see Start ordinal.

Pods in a StatefulSet

Pods in a StatefulSet have a unique ordinal index and a stable network identity.

Examining the Pod's ordinal index

Get the StatefulSet's Pods:

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

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

As mentioned in the StatefulSets concept, the Pods in a StatefulSet have a sticky, unique identity. This identity is based on a unique ordinal index that is assigned to each Pod by the StatefulSet controller.
The Pods' names take the form `<statefulset name>-<ordinal index>`. Since the `web` StatefulSet has two replicas, it creates two Pods, `web-0` and `web-1`.

Using stable network identities

Each Pod has a stable hostname based on its ordinal index. Use `kubectl exec` to execute the `hostname` command in each Pod:

```
for i in 0 1; do kubectl exec "web-$i" -- sh -c 'hostname'; done
```

```
web-0
web-1
```

Use `kubectl run` to execute a container that provides the `nslookup` command from the `dnsutils` package. Using `nslookup` on the Pods' hostnames, you can examine their in-cluster DNS addresses:

```
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
```

which starts a new shell. In that new shell, run:

```
# Run this in the dns-test container shell
nslookup web-0.nginx
```

The output is similar to:

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.6

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.6
```

(and now exit the container shell: `exit`)

The CNAME of the headless service points to SRV records (one for each Pod that is Running and Ready). The SRV records point to A record entries that contain the Pods' IP addresses.

In one terminal, watch the StatefulSet's Pods:

```
# Start a new watch
# End this watch when you've seen that the delete is finished
kubectl get pod --watch -l app=nginx
```

In a second terminal, use `kubectl delete` to delete all the Pods in the StatefulSet:

```
kubectl delete pod -l app=nginx
```

```
pod "web-0" deleted
pod "web-1" deleted
```

Wait for the StatefulSet to restart them, and for both Pods to transition to Running and Ready:

```
# This should already be running
kubectl get pod --watch -l app=nginx
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0          34s
```

Use `kubectl exec` and `kubectl run` to view the Pods' hostnames and in-cluster DNS entries. First, view the Pods' hostnames:

```
for i in 0 1; do kubectl exec web-$i -- sh -c 'hostname'; done
```

```
web-0
web-1
```

then, run:

```
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
```

which starts a new shell.
In that new shell, run:

```
# Run this in the dns-test container shell
nslookup web-0.nginx
```

The output is similar to:

```
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.7

nslookup web-1.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-1.nginx
Address 1: 10.244.2.8
```

(and now exit the container shell: `exit`)

The Pods' ordinals, hostnames, SRV records, and A record names have not changed, but the IP addresses associated with the Pods may have changed. In the cluster used for this tutorial, they have. This is why it is important not to configure other applications to connect to Pods in a StatefulSet by the IP address of a particular Pod (it is OK to connect to Pods by resolving their hostname).

Discovery for specific Pods in a StatefulSet

If you need to find and connect to the active members of a StatefulSet, you should query the CNAME of the headless Service (`nginx.default.svc.cluster.local`). The SRV records associated with the CNAME will contain only the Pods in the StatefulSet that are Running and Ready.

If your application already implements connection logic that tests for liveness and readiness, you can use the SRV records of the Pods (`web-0.nginx.default.svc.cluster.local`, `web-1.nginx.default.svc.cluster.local`), as they are stable, and your application will be able to discover the Pods' addresses when they transition to Running and Ready.

If your application wants to find any healthy Pod in a StatefulSet, and therefore does not need to track each specific Pod, you could also connect to the IP address of a `type: ClusterIP` Service, backed by the Pods in that StatefulSet. You can use the same Service that tracks the StatefulSet (specified in the `serviceName` of the StatefulSet) or a separate Service that selects the right set of Pods.

Writing to stable storage

Get the PersistentVolumeClaims for `web-0` and `web-1`:

```
kubectl get pvc -l app=nginx
```

The output is similar to:

```
NAME        STATUS    VOLUME                                     CAPACITY   ACCESSMODES   AGE
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi        RWO           48s
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002   1Gi        RWO           48s
```

The StatefulSet controller created two PersistentVolumeClaims that are bound to two PersistentVolumes.

As the cluster used in this tutorial is configured to dynamically provision PersistentVolumes, the PersistentVolumes were created and bound automatically.

The NGINX webserver, by default, serves an index file from `/usr/share/nginx/html/index.html`. The `volumeMounts` field in the StatefulSet's `spec` ensures that the `/usr/share/nginx/html` directory is backed by a PersistentVolume.

Write the Pods' hostnames to their `index.html` files and verify that the NGINX webservers serve the hostnames:

```
for i in 0 1; do kubectl exec "web-$i" -- sh -c 'echo "$(hostname)" > /usr/share/nginx/html/index.html'; done

for i in 0 1; do kubectl exec -i -t "web-$i" -- curl http://localhost/; done

web-0
web-1
```

Note:

If you instead see **403 Forbidden** responses for the above curl command, you will need to fix the permissions of the directory mounted by the `volumeMounts` (due to a bug when using hostPath volumes), by running:

```
for i in 0 1; do kubectl exec web-$i -- chmod 755 /usr/share/nginx/html; done
```

before retrying the `curl` command above.

In one terminal, watch the StatefulSet's Pods:

```
# End this watch when you've reached the end of the section.
# At the start of "Scaling a StatefulSet" you'll start a new watch.
kubectl get pod --watch -l app=nginx
```

In a second terminal, delete all of the StatefulSet's Pods:

```
kubectl delete pod -l app=nginx
```

```
pod "web-0" deleted
pod "web-1" deleted
```

Examine the output of the `kubectl get` command in the first terminal, and wait for all of the Pods to transition to Running and Ready.

```
# This should already be running
kubectl get pod --watch -l app=nginx
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     0/1       ContainerCreating   0          0s
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2s
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0          34s
```

Verify the web servers continue to serve their hostnames:

```
for i in 0 1; do kubectl exec -i -t "web-$i" -- curl http://localhost/; done
```

```
web-0
web-1
```

Even though `web-0` and `web-1` were rescheduled, they continue to serve their hostnames because the PersistentVolumes associated with their PersistentVolumeClaims are remounted to their `volumeMounts`. No matter what node `web-0`and `web-1` are scheduled on, their PersistentVolumes will be mounted to the appropriate mount points.

Scaling a StatefulSet

Scaling a StatefulSet refers to increasing or decreasing the number of replicas (horizontal scaling). This is accomplished by updating the `replicas` field. You can use either `kubectl scale` or `kubectl patch` to scale a StatefulSet.

Scaling up

Scaling up means adding more replicas. Provided that your app is able to distribute work across the StatefulSet, the new larger set of Pods can perform more of that work.

In one terminal window, watch the Pods in the StatefulSet:

```
# If you already have a watch running, you can continue using that.
# Otherwise, start one.
# End this watch when there are 5 healthy Pods for the StatefulSet
kubectl get pods --watch -l app=nginx
```

In another terminal window, use `kubectl scale` to scale the number of replicas to 5:

```
kubectl scale sts web --replicas=5
```

```
statefulset.apps/web scaled
```

Examine the output of the `kubectl get` command in the first terminal, and wait for the three additional Pods to transition to Running and Ready.

```
# This should already be running
kubectl get pod --watch -l app=nginx
```

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          2h
web-1     1/1       Running   0          2h
NAME      READY     STATUS    RESTARTS   AGE
web-2     0/1       Pending   0          0s
web-2     0/1       Pending   0          0s
web-2     0/1       ContainerCreating   0         0s
web-2     1/1       Running   0          19s
web-3     0/1       Pending   0          0s
web-3     0/1       Pending   0          0s
web-3     0/1       ContainerCreating   0         0s
web-3     1/1       Running   0          18s
web-4     0/1       Pending   0          0s
web-4     0/1       Pending   0          0s
web-4     0/1       ContainerCreating   0         0s
web-4     1/1       Running   0          19s
```

The StatefulSet controller scaled the number of replicas. As with StatefulSet creation, the StatefulSet controller created each Pod sequentially with respect to its ordinal index, and it waited for each Pod's predecessor to be Running and Ready before launching the subsequent Pod.

Scaling down

Scaling down means reducing the number of replicas. For example, you might do this because the level of traffic to a service has decreased, and at the current scale there are idle resources.

In one terminal, watch the StatefulSet's Pods:

```
# End this watch when there are only 3 Pods for the StatefulSet
kubectl get pod --watch -l app=nginx
```

In another terminal, use `kubectl patch` to scale the StatefulSet back down to three replicas:

```
kubectl patch sts web -p '{"spec":{"replicas":3}}'
```

```
statefulset.apps/web patched
```

Wait for `web-4` and `web-3` to transition to Terminating.

```
# This should already be running
kubectl get pods --watch -l app=nginx
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          3h
web-1     1/1       Running             0          3h
web-2     1/1       Running             0          55s
web-3     1/1       Running             0          36s
web-4     0/1       ContainerCreating   0          18s
NAME      READY     STATUS    RESTARTS   AGE
web-4     1/1       Running   0          19s
web-4     1/1       Terminating   0          24s
web-4     1/1       Terminating   0          24s
web-3     1/1       Terminating   0          42s
web-3     1/1       Terminating   0          42s
```

Ordered Pod termination

The control plane deleted one Pod at a time, in reverse order with respect to its ordinal index, and it waited for each Pod to be completely shut down before deleting the next one.

Get the StatefulSet's PersistentVolumeClaims:

```
kubectl get pvc -l app=nginx
```

```
NAME        STATUS    VOLUME                                     CAPACITY   ACCESSMODES   AGE
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi        RWO           13h
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002   1Gi        RWO           13h
www-web-2   Bound     pvc-e1125b27-b508-11e6-932f-42010a800002   1Gi        RWO           13h
www-web-3   Bound     pvc-e1176df6-b508-11e6-932f-42010a800002   1Gi        RWO           13h
www-web-4   Bound     pvc-e11bb5f8-b508-11e6-932f-42010a800002   1Gi        RWO           13h
```

There are still five PersistentVolumeClaims and five PersistentVolumes. When exploring a Pod's stable storage, you saw that the PersistentVolumes mounted to the Pods of a StatefulSet are not deleted when the StatefulSet's Pods are deleted. This is still true when Pod deletion is caused by scaling the StatefulSet down.

Updating StatefulSets

The StatefulSet controller supports automated updates. The strategy used is determined by the `spec.updateStrategy` field of the StatefulSet API object. This feature can be used to upgrade the container images, resource requests and/or limits, labels, and annotations of the Pods in a StatefulSet.

There are two valid update strategies, `RollingUpdate` (the default) and `OnDelete`.

RollingUpdate

The `RollingUpdate` update strategy will update all Pods in a StatefulSet, in reverse ordinal order, while respecting the StatefulSet guarantees.

You can split updates to a StatefulSet that uses the `RollingUpdate` strategy into *partitions*, by specifying `.spec.updateStrategy.rollingUpdate.partition`. You'll practice that later in this tutorial.

First, try a simple rolling update.

In one terminal window, patch the `web` StatefulSet to change the container image again:

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/image", "value":"regis
```

```
statefulset.apps/web patched
```

In another terminal, watch the Pods in the StatefulSet:

```
# End this watch when the rollout is complete
#
# If you're not sure, leave it running one more minute
kubectl get pod -l app=nginx --watch
```

The output is similar to:

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          7m
web-1     1/1       Running   0          7m
web-2     1/1       Running   0          8m
web-2     1/1       Terminating   0          8m
web-2     1/1       Terminating   0          8m
web-2     0/1       Terminating   0          8m
web-2     0/1       Terminating   0          8m
web-2     0/1       Terminating   0          8m
web-2     0/1       Terminating   0          8m
web-2     0/1       Pending   0         0s
web-2     0/1       Pending   0         0s
web-2     0/1       ContainerCreating   0         0s
web-2     1/1       Running   0         19s
```

```
web-1     1/1       Terminating   0          8m
web-1     0/1       Terminating   0          8m
web-1     0/1       Terminating   0          8m
web-1     0/1       Terminating   0          8m
web-1     0/1       Pending   0          0s
web-1     0/1       Pending   0          0s
web-1     0/1       ContainerCreating   0          0s
web-1     1/1       Running   0          6s
web-0     1/1       Terminating   0          7m
web-0     1/1       Terminating   0          7m
web-0     0/1       Terminating   0          7m
web-0     0/1       Terminating   0          7m
web-0     0/1       Terminating   0          7m
web-0     0/1       Terminating   0          7m
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0          0s
web-0     0/1       ContainerCreating   0          0s
web-0     1/1       Running   0          10s
```

The Pods in the StatefulSet are updated in reverse ordinal order. The StatefulSet controller terminates each Pod, and waits for it to transition to Running and Ready prior to updating the next Pod. Note that, even though the StatefulSet controller will not proceed to update the next Pod until its ordinal successor is Running and Ready, it will restore any Pod that fails during the update to that Pod's existing version.

Pods that have already received the update will be restored to the updated version, and Pods that have not yet received the update will be restored to the previous version. In this way, the controller attempts to continue to keep the application healthy and the update consistent in the presence of intermittent failures.

Get the Pods to view their container images:

```
for p in 0 1 2; do kubectl get pod "web-$p" --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
```

```
registry.k8s.io/nginx-slim:0.24
registry.k8s.io/nginx-slim:0.24
registry.k8s.io/nginx-slim:0.24
```

All the Pods in the StatefulSet are now running the previous container image.

Note:

You can also use `kubectl rollout status sts/<name>` to view the status of a rolling update to a StatefulSet

Staging an update

You can split updates to a StatefulSet that uses the `RollingUpdate` strategy into *partitions*, by specifying `.spec.updateStrategy.rollingUpdate.partition`.

For more context, you can read Partitioned rolling updates in the StatefulSet concept page.

You can stage an update to a StatefulSet by using the `partition` field within `.spec.updateStrategy.rollingUpdate`. For this update, you will keep the existing Pods in the StatefulSet unchanged whilst you change the pod template for the StatefulSet. Then you - or, outside of a tutorial, some external automation - can trigger that prepared update.

First, patch the `web` StatefulSet to add a partition to the `updateStrategy` field:

```
# The value of "partition" determines which ordinals a change applies to
# Make sure to use a number bigger than the last ordinal for the
# StatefulSet
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate","rollingUpdate":{"partition":3}}}}'
```

```
statefulset.apps/web patched
```

Patch the StatefulSet again to change the container image that this StatefulSet uses:

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/template/spec/containers/0/image", "value":"regis
```

```
statefulset.apps/web patched
```

Delete a Pod in the StatefulSet:

```
kubectl delete pod web-2
```

```
pod "web-2" deleted
```

Wait for the replacement `web-2` Pod to be Running and Ready:

```
# End the watch when you see that web-2 is healthy
kubectl get pod -l app=nginx --watch
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          4m
web-1     1/1       Running             0          4m
web-2     0/1       ContainerCreating   0          11s
web-2     1/1       Running   0         18s
```

Get the Pod's container image:

```
kubectl get pod web-2 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'
```

```
registry.k8s.io/nginx-slim:0.24
```

Notice that, even though the update strategy is `RollingUpdate` the StatefulSet restored the Pod with the original container image. This is because the ordinal of the Pod is less than the `partition` specified by the `updateStrategy`.

Rolling out a canary

You're now going to try a canary rollout of that staged change.

You can roll out a canary (to test the modified template) by decrementing the `partition` you specified above.

Patch the StatefulSet to decrement the partition:

```
# The value of "partition" should match the highest existing ordinal for
# the StatefulSet
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate","rollingUpdate":{"partition":2}}}}'
```

```
statefulset.apps/web patched
```

The control plane triggers replacement for `web-2` (implemented by a graceful **delete** followed by creating a new Pod once the deletion is complete). Wait for the new `web-2` Pod to be Running and Ready.

```
# This should already be running
kubectl get pod -l app=nginx --watch
```

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          4m
web-1     1/1       Running             0          4m
web-2     0/1       ContainerCreating   0          11s
web-2     1/1       Running   0         18s
```

Get the Pod's container:

```
kubectl get pod web-2 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'
```

```
registry.k8s.io/nginx-slim:0.21
```

When you changed the `partition`, the StatefulSet controller automatically updated the `web-2` Pod because the Pod's ordinal was greater than or equal to the `partition`.

Delete the `web-1` Pod:

```
kubectl delete pod web-1
```

```
pod "web-1" deleted
```

Wait for the `web-1` Pod to be Running and Ready.

```
# This should already be running
kubectl get pod -l app=nginx --watch
```

The output is similar to:

```
NAME      READY     STATUS        RESTARTS   AGE
web-0     1/1       Running       0          6m
web-1     0/1       Terminating   0          6m
web-2     1/1       Running       0          2m
web-1     0/1       Terminating   0         6m
web-1     0/1       Terminating   0         6m
web-1     0/1       Terminating   0         6m
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       ContainerCreating   0         0s
web-1     1/1       Running   0         18s
```

Get the `web-1` Pod's container image:

```
kubectl get pod web-1 --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'
```

```
registry.k8s.io/nginx-slim:0.24
```

`web-1` was restored to its original configuration because the Pod's ordinal was less than the partition. When a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet's `.spec.template` is updated. If a Pod that has an ordinal less than the partition is deleted or otherwise terminated, it will be restored to its original configuration.

Phased roll outs

You can perform a phased roll out (e.g. a linear, geometric, or exponential roll out) using a partitioned rolling update in a similar manner to how you rolled out a canary. To perform a phased roll out, set the `partition` to the ordinal at which you want the controller to pause the update.

The partition is currently set to `2`. Set the partition to `0`:

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"RollingUpdate","rollingUpdate":{"partition":0}}}}'
```

```
statefulset.apps/web patched
```

Wait for all of the Pods in the StatefulSet to become Running and Ready.

```
# This should already be running
kubectl get pod -l app=nginx --watch
```

The output is similar to:

```
NAME      READY     STATUS              RESTARTS   AGE
web-0     1/1       Running             0          3m
web-1     0/1       ContainerCreating   0          11s
web-2     1/1       Running             0          2m
web-1     1/1       Running   0         18s
web-0     1/1       Terminating   0         3m
web-0     1/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Terminating   0         3m
web-0     0/1       Pending   0         0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         3s
```

Get the container image details for the Pods in the StatefulSet:

```
for p in 0 1 2; do kubectl get pod "web-$p" --template '{{range $i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
```

```
registry.k8s.io/nginx-slim:0.21
registry.k8s.io/nginx-slim:0.21
registry.k8s.io/nginx-slim:0.21
```

By moving the `partition` to `0`, you allowed the StatefulSet to continue the update process.

OnDelete

You select this update strategy for a StatefulSet by setting the `.spec.template.updateStrategy.type` to `OnDelete`.

Patch the `web` StatefulSet to use the `OnDelete` update strategy:

```
kubectl patch statefulset web -p '{"spec":{"updateStrategy":{"type":"OnDelete", "rollingUpdate": null}}}'
```

```
statefulset.apps/web patched
```

When you select this update strategy, the StatefulSet controller does not automatically update Pods when a modification is made to the StatefulSet's `.spec.template` field. You need to manage the rollout yourself - either manually, or using separate automation.

Deleting StatefulSets

StatefulSet supports both *non-cascading* and *cascading* deletion. In a non-cascading **delete**, the StatefulSet's Pods are not deleted when the StatefulSet is deleted. In a cascading **delete**, both the StatefulSet and its Pods are deleted.

Read Use Cascading Deletion in a Cluster to learn about cascading deletion generally.

Non-cascading delete

In one terminal window, watch the Pods in the StatefulSet.

```
# End this watch when there are no Pods for the StatefulSet
kubectl get pods --watch -l app=nginx
```

Use `kubectl delete` to delete the StatefulSet. Make sure to supply the `--cascade=orphan` parameter to the command. This parameter tells Kubernetes to only delete the StatefulSet, and to **not** delete any of its Pods.

```
kubectl delete statefulset web --cascade=orphan
```

```
statefulset.apps "web" deleted
```

Get the Pods, to examine their status:

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

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          6m
web-1     1/1       Running   0          7m
web-2     1/1       Running   0          5m
```

Even though `web` has been deleted, all of the Pods are still Running and Ready. Delete `web-0`:

```
kubectl delete pod web-0
```

```
pod "web-0" deleted
```

Get the StatefulSet's Pods:

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

```
NAME      READY     STATUS    RESTARTS   AGE
web-1     1/1       Running   0          10m
web-2     1/1       Running   0          7m
```

As the `web` StatefulSet has been deleted, `web-0` has not been relaunched.

In one terminal, watch the StatefulSet's Pods.

```
# Leave this watch running until the next time you start a watch
kubectl get pods --watch -l app=nginx
```

In a second terminal, recreate the StatefulSet. Note that, unless you deleted the `nginx` Service (which you should not have), you will see an error indicating that the Service already exists.

```
kubectl apply -f https://k8s.io/examples/application/web/web.yaml
```

```
statefulset.apps/web created
service/nginx unchanged
```

Ignore the error. It only indicates that an attempt was made to create the *nginx* headless Service even though that Service already exists.

Examine the output of the `kubectl get` command running in the first terminal.

```
# This should already be running
kubectl get pods --watch -l app=nginx
```

```
NAME      READY     STATUS    RESTARTS   AGE
web-1     1/1       Running   0          16m
web-2     1/1       Running   0          2m
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         18s
web-2     1/1       Terminating   0          3m
web-2     0/1       Terminating   0          3m
web-2     0/1       Terminating   0          3m
web-2     0/1       Terminating   0          3m
```

When the `web` StatefulSet was recreated, it first relaunched `web-0`. Since `web-1` was already Running and Ready, when `web-0` transitioned to Running and Ready, it adopted this Pod. Since you recreated the StatefulSet with `replicas` equal to 2, once `web-0` had been recreated, and once `web-1` had been determined to already be Running and Ready, `web-2` was terminated.

Now take another look at the contents of the `index.html` file served by the Pods' webservers:

```
for i in 0 1; do kubectl exec -i -t "web-$i" -- curl http://localhost/; done
```

```
web-0
web-1
```

Even though you deleted both the StatefulSet and the `web-0` Pod, it still serves the hostname originally entered into its `index.html` file. This is because the StatefulSet never deletes the PersistentVolumes associated with a Pod. When you recreated the StatefulSet and it relaunched `web-0`, its original PersistentVolume was remounted.

Cascading delete

In one terminal window, watch the Pods in the StatefulSet.

```
# Leave this running until the next page section
kubectl get pods --watch -l app=nginx
```

In another terminal, delete the StatefulSet again. This time, omit the `--cascade=orphan` parameter.

```
kubectl delete statefulset web
```

```
statefulset.apps "web" deleted
```

Examine the output of the `kubectl get` command running in the first terminal, and wait for all of the Pods to transition to Terminating.

```
# This should already be running
kubectl get pods --watch -l app=nginx
```

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     1/1       Running   0          11m
web-1     1/1       Running   0          27m
NAME      READY     STATUS        RESTARTS   AGE
web-0     1/1       Terminating   0          12m
web-1     1/1       Terminating   0          29m
web-0     0/1       Terminating   0          12m
web-0     0/1       Terminating   0          12m
web-0     0/1       Terminating   0          12m
web-1     0/1       Terminating   0          29m
web-1     0/1       Terminating   0          29m
web-1     0/1       Terminating   0          29m
```

As you saw in the Scaling Down section, the Pods are terminated one at a time, with respect to the reverse order of their ordinal indices. Before terminating a Pod, the StatefulSet controller waits for the Pod's successor to be completely terminated.

Note:

Although a cascading delete removes a StatefulSet together with its Pods, the cascade does **not** delete the headless Service associated with the StatefulSet. You must delete the `nginx` Service manually.

```
kubectl delete service nginx
```

```
service "nginx" deleted
```

Recreate the StatefulSet and headless Service one more time:

```
kubectl apply -f https://k8s.io/examples/application/web/web.yaml
```

```
service/nginx created
statefulset.apps/web created
```

When all of the StatefulSet's Pods transition to Running and Ready, retrieve the contents of their `index.html` files:

```
for i in 0 1; do kubectl exec -i -t "web-$i" -- curl http://localhost/; done
```

```
web-0
web-1
```

Even though you completely deleted the StatefulSet, and all of its Pods, the Pods are recreated with their PersistentVolumes mounted, and `web-0` and `web-1` continue to serve their hostnames.

Finally, delete the `nginx` Service...

```
kubectl delete service nginx
```

```
service "nginx" deleted
```

...and the `web` StatefulSet:

```
kubectl delete statefulset web
```

```
statefulset "web" deleted
```

Pod management policy

For some distributed systems, the StatefulSet ordering guarantees are unnecessary and/or undesirable. These systems require only uniqueness and identity.

You can specify a Pod management policy to avoid this strict ordering; either `OrderedReady` (the default), or `Parallel`.

OrderedReady Pod management

`OrderedReady` pod management is the default for StatefulSets. It tells the StatefulSet controller to respect the ordering guarantees demonstrated above.

Use this when your application requires or expects that changes, such as rolling out a new version of your application, happen in the strict order of the ordinal (pod number) that the StatefulSet provides. In other words, if you have Pods `app-0`, `app-1` and `app-2`, Kubernetes will update `app-0` first and check it. Once the checks are good, Kubernetes updates `app-1` and finally `app-2`.

If you added two more Pods, Kubernetes would set up `app-3` and wait for that to become healthy before deploying `app-4`.

Because this is the default setting, you've already practised using it.

Parallel Pod management

The alternative, `Parallel` pod management, tells the StatefulSet controller to launch or terminate all Pods in parallel, and not to wait for Pods to become `Running` and `Ready` or completely terminated prior to launching or terminating another Pod.

The `Parallel` pod management option only affects the behavior for scaling operations. Updates are not affected; Kubernetes still rolls out changes in order. For this tutorial, the application is very simple: a webserver that tells you its hostname (because this is a StatefulSet, the hostname for each Pod is different and predictable).

application/web/web-parallel.yaml  Copy application/web/web-parallel.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: nginx  labels:    app: nginxspec:  ports:  - port: 80    name: web  clusterIP: None  selector:    ap
```

This manifest is identical to the one you downloaded above except that the `.spec.podManagementPolicy` of the `web` StatefulSet is set to `Parallel`.

In one terminal, watch the Pods in the StatefulSet.

```
# Leave this watch running until the end of the section
kubectl get pod -l app=nginx --watch
```

In another terminal, reconfigure the StatefulSet for `Parallel` Pod management:

```
kubectl apply -f https://k8s.io/examples/application/web/web-parallel.yaml
```

```
service/nginx updated
statefulset.apps/web updated
```

Keep the terminal open where you're running the watch. In another terminal window, scale the StatefulSet:

```
kubectl scale statefulset/web --replicas=5
```

```
statefulset.apps/web scaled
```

Examine the output of the terminal where the `kubectl get` command is running. It may look something like

```
web-3     0/1       Pending   0         0s
web-3     0/1       Pending   0         0s
web-3     0/1       Pending   0         7s
web-3     0/1       ContainerCreating   0         7s
web-2     0/1       Pending   0         0s
web-4     0/1       Pending   0         0s
web-2     1/1       Running   0         8s
web-4     0/1       ContainerCreating   0         4s
```

```
web-3     1/1       Running   0         26s
web-4     1/1       Running   0         2s
```

The StatefulSet launched three new Pods, and it did not wait for the first to become Running and Ready prior to launching the second and third Pods.

This approach is useful if your workload has a stateful element, or needs Pods to be able to identify each other with predictable naming, and especially if you sometimes need to provide a lot more capacity quickly. If this simple web service for the tutorial suddenly got an extra 1,000,000 requests per minute then you would want to run some more Pods - but you also would not want to wait for each new Pod to launch. Starting the extra Pods in parallel cuts the time between requesting the extra capacity and having it available for use.

Cleaning up

You should have two terminals open, ready for you to run `kubectl` commands as part of cleanup.

```
kubectl delete sts web
# sts is an abbreviation for statefulset
```

You can watch `kubectl get` to see those Pods being deleted.

```
# end the watch when you've seen what you need to
kubectl get pod -l app=nginx --watch
```

```
web-3     1/1       Terminating   0         9m
web-2     1/1       Terminating   0         9m
web-3     1/1       Terminating   0         9m
web-2     1/1       Terminating   0         9m
web-1     1/1       Terminating   0         44m
web-0     1/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-3     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-1     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-2     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-2     0/1       Terminating   0         9m
web-1     0/1       Terminating   0         44m
web-1     0/1       Terminating   0         44m
web-1     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-0     0/1       Terminating   0         44m
web-3     0/1       Terminating   0         9m
web-3     0/1       Terminating   0         9m
web-3     0/1       Terminating   0         9m
```

During deletion, a StatefulSet removes all Pods concurrently; it does not wait for a Pod's ordinal successor to terminate prior to deleting that Pod.

Close the terminal where the `kubectl get` command is running and delete the `nginx` Service:

```
kubectl delete svc nginx
```

Delete the persistent storage media for the PersistentVolumes used in this tutorial.

```
kubectl get pvc
```

```
NAME        STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
www-web-0   Bound     pvc-2bf00408-d366-4a12-bad0-1869c65d0bee   1Gi        RWO            standard       25m
www-web-1   Bound     pvc-ba3bfe9c-413e-4b95-a2c0-3ea8a54dbab4   1Gi        RWO            standard       24m
www-web-2   Bound     pvc-cba6cfa6-3a47-486b-a138-db5930207eaf   1Gi        RWO            standard       15m
www-web-3   Bound     pvc-0c04d7f0-787a-4977-8da3-d9d3a6d8d752   1Gi        RWO            standard       15m
www-web-4   Bound     pvc-b2c73489-e70b-4a4e-9ec1-9eab439aa43e   1Gi        RWO            standard       14m
```

```
kubectl get pv
```

```
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS    CLAIM               STORAGECLASS   l
pvc-0c04d7f0-787a-4977-8da3-d9d3a6d8d752   1Gi        RWO            Delete           Bound     default/www-web-3   standard
pvc-2bf00408-d366-4a12-bad0-1869c65d0bee   1Gi        RWO            Delete           Bound     default/www-web-0   standard
pvc-b2c73489-e70b-4a4e-9ec1-9eab439aa43e   1Gi        RWO            Delete           Bound     default/www-web-4   standard
pvc-ba3bfe9c-413e-4b95-a2c0-3ea8a54dbab4   1Gi        RWO            Delete           Bound     default/www-web-1   standard
pvc-cba6cfa6-3a47-486b-a138-db5930207eaf   1Gi        RWO            Delete           Bound     default/www-web-2   standard
```

```
kubectl delete pvc www-web-0 www-web-1 www-web-2 www-web-3 www-web-4
```

```
persistentvolumeclaim "www-web-0" deleted
persistentvolumeclaim "www-web-1" deleted
persistentvolumeclaim "www-web-2" deleted
persistentvolumeclaim "www-web-3" deleted
persistentvolumeclaim "www-web-4" deleted
```

```
kubectl get pvc
```

```
No resources found in default namespace.
```

Note:

You also need to delete the persistent storage media for the PersistentVolumes used in this tutorial. Follow the necessary steps, based on your environment, storage configuration, and provisioning method, to ensure that all storage is reclaimed.

Connecting Applications with Services

The Kubernetes model for connecting containers

Now that you have a continuously running, replicated application you can expose it on a network.

Kubernetes assumes that pods can communicate with other pods, regardless of which host they land on. Kubernetes gives every pod its own cluster-private IP address, so you do not need to explicitly create links between pods or map container ports to host ports. This means that containers within a Pod can all reach each other's ports on localhost, and all pods in a cluster can see each other without NAT. The rest of this document elaborates on how you can run reliable services on such a networking model.

This tutorial uses a simple nginx web server to demonstrate the concept.

Exposing pods to the cluster

We did this in a previous example, but let's do it once again and focus on the networking perspective. Create an nginx Pod, and note that it has a container port specification:

service/networking/run-my-nginx.yaml Copy service/networking/run-my-nginx.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: my-nginxspec:  selector:    matchLabels:      run: my-nginx  replicas: 2  template:    metadata:
```

This makes it accessible from any node in your cluster. Check the nodes the Pod is running on:

```
kubectl apply -f ./run-my-nginx.yaml
kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS   AGE       IP            NODE
my-nginx-3800858182-jr4a2   1/1       Running   0          13s       10.244.3.4    kubernetes-minion-905m
my-nginx-3800858182-kna2y   1/1       Running   0          13s       10.244.2.5    kubernetes-minion-ljyd
```

Check your pods' IPs:

```
kubectl get pods -l run=my-nginx -o custom-columns=POD_IP:.status.podIPs
    POD_IP
    [map[ip:10.244.3.4]]
    [map[ip:10.244.2.5]]
```

You should be able to ssh into any node in your cluster and use a tool such as `curl` to make queries against both IPs. Note that the containers are *not* using port 80 on the node, nor are there any special NAT rules to route traffic to the pod. This means you can run multiple nginx pods on the same node all using the same `containerPort`, and access them from any other pod or node in your cluster using the assigned IP address for the pod. If you want to arrange for a specific port on the host Node to be forwarded to backing Pods, you can - but the networking model should mean that you do not need to do so.

You can read more about the Kubernetes Networking Model if you're curious.

Creating a Service

So we have pods running nginx in a flat, cluster wide, address space. In theory, you could talk to these pods directly, but what happens when a node dies? The pods die with it, and the ReplicaSet inside the Deployment will create new ones, with different IPs. This is the problem a Service solves.

A Kubernetes Service is an abstraction which defines a logical set of Pods running somewhere in your cluster, that all provide the same functionality. When created, each Service is assigned a unique IP address (also called clusterIP). This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the Service, and know that communication to the Service will be automatically load-balanced out to some pod that is a member of the Service.

You can create a Service for your 2 nginx replicas with `kubectl expose`:

```
kubectl expose deployment/my-nginx
```

```
service/my-nginx exposed
```

This is equivalent to `kubectl apply -f` in the following yaml:

service/networking/nginx-svc.yaml Copy service/networking/nginx-svc.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-nginx  labels:    run: my-nginxspec:  ports:  - port: 80    protocol: TCP  selector:    run: my-n
```

This specification will create a Service which targets TCP port 80 on any Pod with the `run: my-nginx` label, and expose it on an abstracted Service port (`targetPort`: is the port the container accepts traffic on, `port`: is the abstracted Service port, which can be any port other pods use to access the Service). View Service API object to see the list of supported fields in service definition. Check your Service:

```
kubectl get svc my-nginx
```

```
NAME       TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
my-nginx   ClusterIP   10.0.162.149   <none>        80/TCP    21s
```

As mentioned previously, a Service is backed by a group of Pods. These Pods are exposed through EndpointSlices. The Service's selector will be evaluated continuously and the results will be POSTed to an EndpointSlice that is connected to the Service using labels. When a Pod dies, it is automatically removed from the EndpointSlices that contain it as an endpoint. New Pods that match the Service's selector will automatically get added to an EndpointSlice for that Service. Check the endpoints, and note that the IPs are the same as the Pods created in the first step:

```
kubectl describe svc my-nginx
```

```
Name:                my-nginx
Namespace:           default
Labels:              run=my-nginx
```

```
Annotations:         <none>
Selector:            run=my-nginx
Type:                ClusterIP
IP Family Policy:    SingleStack
IP Families:         IPv4
IP:                  10.0.162.149
IPs:                 10.0.162.149
Port:                <unset> 80/TCP
TargetPort:          80/TCP
Endpoints:           10.244.2.5:80,10.244.3.4:80
Session Affinity:    None
Events:              <none>
```

```
kubectl get endpointslices -l kubernetes.io/service-name=my-nginx
```

```
NAME             ADDRESSTYPE   PORTS   ENDPOINTS               AGE
my-nginx-7vzhx   IPv4          80      10.244.2.5,10.244.3.4   21s
```

You should now be able to curl the nginx Service on `<CLUSTER-IP>:<PORT>` from any node in your cluster. Note that the Service IP is completely virtual, it never hits the wire. If you're curious about how this works you can read more about the service proxy.

Accessing the Service

Kubernetes supports 2 primary modes of finding a Service - environment variables and DNS. The former works out of the box while the latter requires the CoreDNS cluster addon.

Note:

If the service environment variables are not desired (because possible clashing with expected program ones, too many variables to process, only using DNS, etc) you can disable this mode by setting the `enableServiceLinks` flag to `false` on the pod spec.

Environment Variables

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service. This introduces an ordering problem. To see why, inspect the environment of your running nginx Pods (your Pod name will be different):

```
kubectl exec my-nginx-3800858182-jr4a2 -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_HOST=10.0.0.1
KUBERNETES_SERVICE_PORT=443
KUBERNETES_SERVICE_PORT_HTTPS=443
```

Note there's no mention of your Service. This is because you created the replicas before the Service. Another disadvantage of doing this is that the scheduler might put both Pods on the same machine, which will take your entire Service down if it dies. We can do this the right way by killing the 2 Pods and waiting for the Deployment to recreate them. This time the Service exists *before* the replicas. This will give you scheduler-level Service spreading of your Pods (provided all your nodes have equal capacity), as well as the right environment variables:

```
kubectl scale deployment my-nginx --replicas=0; kubectl scale deployment my-nginx --replicas=2;

kubectl get pods -l run=my-nginx -o wide
```

```
NAME                        READY     STATUS    RESTARTS   AGE     IP            NODE
my-nginx-3800858182-e9ihh   1/1       Running   0          5s      10.244.2.7    kubernetes-minion-ljyd
my-nginx-3800858182-j4rm4   1/1       Running   0          5s      10.244.3.8    kubernetes-minion-905m
```

You may notice that the pods have different names, since they are killed and recreated.

```
kubectl exec my-nginx-3800858182-e9ihh -- printenv | grep SERVICE
```

```
KUBERNETES_SERVICE_PORT=443
MY_NGINX_SERVICE_HOST=10.0.162.149
KUBERNETES_SERVICE_HOST=10.0.0.1
MY_NGINX_SERVICE_PORT=80
KUBERNETES_SERVICE_PORT_HTTPS=443
```

DNS

Kubernetes offers a DNS cluster addon Service that automatically assigns dns names to other Services. You can check if it's running on your cluster:

```
kubectl get services kube-dns --namespace=kube-system
```

```
NAME       TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)         AGE
kube-dns   ClusterIP   10.0.0.10    <none>        53/UDP,53/TCP   8m
```

The rest of this section will assume you have a Service with a long lived IP (my-nginx), and a DNS server that has assigned a name to that IP. Here we use the CoreDNS cluster addon (application name `kube-dns`), so you can talk to the Service from any pod in your cluster using standard methods (e.g. `gethostbyname()`). If CoreDNS isn't running, you can enable it referring to the CoreDNS README or Installing CoreDNS. Let's run another curl application to test this:

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty --rm
```

```
Waiting for pod default/curl-131556218-9fnch to be running, status is Pending, pod ready: false
Hit enter for command prompt
```

Then, hit enter and run `nslookup my-nginx`:

```
[ root@curl-131556218-9fnch:/ ]$ nslookup my-nginx
Server:    10.0.0.10
Address 1: 10.0.0.10
```

```
Name:      my-nginx
Address 1: 10.0.162.149
```

Securing the Service

Till now we have only accessed the nginx server from within the cluster. Before exposing the Service to the internet, you want to make sure the communication channel is secure. For this, you will need:

- Self signed certificates for https (unless you already have an identity certificate)
- An nginx server configured to use the certificates
- A secret that makes the certificates accessible to pods

You can acquire all these from the nginx https example. This requires having go and make tools installed. If you don't want to install those, then follow the manual steps later. In short:

```
make keys KEY=/tmp/nginx.key CERT=/tmp/nginx.crt
kubectl create secret tls nginxsecret --key /tmp/nginx.key --cert /tmp/nginx.crt
```

```
secret/nginxsecret created
```

```
kubectl get secrets
```

```
NAME                  TYPE                                  DATA      AGE
nginxsecret           kubernetes.io/tls                     2         1m
```

And also the configmap:

```
kubectl create configmap nginxconfigmap --from-file=default.conf
```

You can find an example for `default.conf` in the Kubernetes examples project repo.

```
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

```
NAME             DATA   AGE
nginxconfigmap   1      114s
```

You can view the details of the `nginxconfigmap` ConfigMap using the following command:

```
kubectl describe configmap  nginxconfigmap
```

The output is similar to:

```
Name:         nginxconfigmap
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data====default.conf:----server {        listen 80 default_server;        listen [::]:80 default_server ipv6only=on;        listen
```

Following are the manual steps to follow in case you run into problems running make (on windows for example):

```
# Create a public private key pair
openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /d/tmp/nginx.key -out /d/tmp/nginx.crt -subj "/CN=my-nginx/O=my-nginx"
# Convert the keys to base64 encoding
cat /d/tmp/nginx.crt | base64
cat /d/tmp/nginx.key | base64
```

Use the output from the previous commands to create a yaml file as follows. The base64 encoded value should all be on a single line.

```
apiVersion: "v1"
kind: "Secret"metadata:  name: "nginxsecret"  namespace: "default"type: kubernetes.io/tlsdata: # NOTE: Replace the following value
```

Now create the secrets using the file:

```
kubectl apply -f nginxsecrets.yaml
kubectl get secrets
```

```
NAME                  TYPE                                  DATA      AGE
nginxsecret           kubernetes.io/tls                     2         1m
```

Now modify your nginx replicas to start an https server using the certificate in the secret, and the Service, to expose both ports (80 and 443):

service/networking/nginx-secure-app.yaml  Copy service/networking/nginx-secure-app.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: my-nginx  labels:    run: my-nginxspec:  type: NodePort  ports:  - port: 8080    targetPort: 80    p
```

Noteworthy points about the nginx-secure-app manifest:

- It contains both Deployment and Service specification in the same file.
- The nginx server serves HTTP traffic on port 80 and HTTPS traffic on 443, and nginx Service exposes both ports.
- Each container has access to the keys through a volume mounted at `/etc/nginx/ssl`. This is set up *before* the nginx server is started.

```
kubectl delete deployments,svc my-nginx; kubectl create -f ./nginx-secure-app.yaml
```

At this point you can reach the nginx server from any node.

```
kubectl get pods -l run=my-nginx -o custom-columns=POD_IP:.status.podIPs
    POD_IP
```

```
    [map[ip:10.244.3.5]]

node $ curl -k https://10.244.3.5
...
<h1>Welcome to nginx!</h1>
```

Note how we supplied the `-k` parameter to curl in the last step, this is because we don't know anything about the pods running nginx at certificate generation time, so we have to tell curl to ignore the CName mismatch. By creating a Service we linked the CName used in the certificate with the actual DNS name used by pods during Service lookup. Let's test this from a pod (the same secret is being reused for simplicity, the pod only needs nginx.crt to access the Service):

service/networking/curlpod.yaml  Copy service/networking/curlpod.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: curl-deploymentspec:  selector:    matchLabels:      app: curlpod  replicas: 1  template:    meta
```

```
kubectl apply -f ./curlpod.yaml
kubectl get pods -l app=curlpod

NAME                                READY     STATUS    RESTARTS   AGE
curl-deployment-1515033274-1410r   1/1       Running   0          1m

kubectl exec curl-deployment-1515033274-1410r -- curl https://my-nginx --cacert /etc/nginx/ssl/tls.crt
...
<title>Welcome to nginx!</title>
...
```

Exposing the Service

For some parts of your applications you may want to expose a Service onto an external IP address. Kubernetes supports two ways of doing this: NodePorts and LoadBalancers. The Service created in the last section already used `NodePort`, so your nginx HTTPS replica is ready to serve traffic on the internet if your node has a public IP.

```
kubectl get svc my-nginx -o yaml | grep nodePort -C 5
  uid: 07191fb3-f61a-11e5-8ae5-42010af00002
spec:
  clusterIP: 10.0.162.149
  ports:
  - name: http
    nodePort: 31704
    port: 8080
    protocol: TCP
    targetPort: 80
  - name: https
    nodePort: 32453
    port: 443
    protocol: TCP
    targetPort: 443
  selector:
    run: my-nginx

kubectl get nodes -o yaml | grep ExternalIP -C 1
    - address: 104.197.41.11
      type: ExternalIP
    allocatable:
--
    - address: 23.251.152.56
      type: ExternalIP
    allocatable:
...

$ curl https://<EXTERNAL-IP>:<NODE-PORT> -k
...
<h1>Welcome to nginx!</h1>
```

Let's now recreate the Service to use a cloud load balancer. Change the `Type` of `my-nginx` Service from `NodePort` to `LoadBalancer`:

```
kubectl edit svc my-nginx
kubectl get svc my-nginx

NAME       TYPE           CLUSTER-IP     EXTERNAL-IP        PORT(S)               AGE
my-nginx   LoadBalancer   10.0.162.149     xx.xxx.xxx.xxx     8080:30163/TCP        21s

curl https://<EXTERNAL-IP> -k
...
<title>Welcome to nginx!</title>
```

The IP address in the `EXTERNAL-IP` column is the one that is available on the public internet. The `CLUSTER-IP` is only available inside your cluster/private cloud network.

Note that on AWS, type `LoadBalancer` creates an ELB, which uses a (long) hostname, not an IP. It's too long to fit in the standard `kubectl get svc` output, in fact, so you'll need to do `kubectl describe service my-nginx` to see it. You'll see something like this:

```
kubectl describe service my-nginx
...
LoadBalancer Ingress:   a320587ffd19711e5a37606cf4a74574-1142138393.us-east-1.elb.amazonaws.com
...
```

What's next

- Learn more about Using a Service to Access an Application in a Cluster

- Learn more about Connecting a Front End to a Back End Using a Service
- Learn more about Creating an External Load Balancer

Example: Deploying WordPress and MySQL with Persistent Volumes

This tutorial shows you how to deploy a WordPress site and a MySQL database using Minikube. Both applications use PersistentVolumes and PersistentVolumeClaims to store data.

A PersistentVolume (PV) is a piece of storage in the cluster that has been manually provisioned by an administrator, or dynamically provisioned by Kubernetes using a StorageClass. A PersistentVolumeClaim (PVC) is a request for storage by a user that can be fulfilled by a PV. PersistentVolumes and PersistentVolumeClaims are independent from Pod lifecycles and preserve data through restarting, rescheduling, and even deleting Pods.

Warning:

This deployment is not suitable for production use cases, as it uses single instance WordPress and MySQL Pods. Consider using WordPress Helm Chart to deploy WordPress in production.

Note:

The files provided in this tutorial are using GA Deployment APIs and are specific to kubernetes version 1.9 and later. If you wish to use this tutorial with an earlier version of Kubernetes, please update the API version appropriately, or reference earlier versions of this tutorial.

Objectives

- Create PersistentVolumeClaims and PersistentVolumes
- Create a `kustomization.yaml` with
  - a Secret generator
  - MySQL resource configs
  - WordPress resource configs
- Apply the kustomization directory by `kubectl apply -k ./`
- Clean up

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To check the version, enter `kubectl version`.

The example shown on this page works with `kubectl` 1.27 and above.

Download the following configuration files:

1. mysql-deployment.yaml
2. wordpress-deployment.yaml

Create PersistentVolumeClaims and PersistentVolumes

MySQL and Wordpress each require a PersistentVolume to store data. Their PersistentVolumeClaims will be created at the deployment step.

Many cluster environments have a default StorageClass installed. When a StorageClass is not specified in the PersistentVolumeClaim, the cluster's default StorageClass is used instead.

When a PersistentVolumeClaim is created, a PersistentVolume is dynamically provisioned based on the StorageClass configuration.

Warning:

In local clusters, the default StorageClass uses the `hostPath` provisioner. `hostPath` volumes are only suitable for development and testing. With `hostPath` volumes, your data lives in `/tmp` on the node the Pod is scheduled onto and does not move between nodes. If a Pod dies and gets scheduled to another node in the cluster, or the node is rebooted, the data is lost.

Note:

If you are bringing up a cluster that needs to use the `hostPath` provisioner, the `--enable-hostpath-provisioner` flag must be set in the `controller-manager` component.

Note:

If you have a Kubernetes cluster running on Google Kubernetes Engine, please follow this guide.

Create a kustomization.yaml

Add a Secret generator

A Secret is an object that stores a piece of sensitive data like a password or key. Since 1.14, `kubectl` supports the management of Kubernetes objects using a kustomization file. You can create a Secret by generators in `kustomization.yaml`.

Add a Secret generator in `kustomization.yaml` from the following command. You will need to replace `YOUR_PASSWORD` with the password you want to use.

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: mysql-pass
  literals:
  - password=YOUR_PASSWORD
EOF
```

Add resource configs for MySQL and WordPress

The following manifest describes a single-instance MySQL Deployment. The MySQL container mounts the PersistentVolume at /var/lib/mysql. The `MYSQL_ROOT_PASSWORD` environment variable sets the database password from the Secret.

application/wordpress/mysql-deployment.yaml  Copy application/wordpress/mysql-deployment.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: wordpress-mysql  labels:    app: wordpressspec:  ports:    - port: 3306  selector:    app: wordpress
```

The following manifest describes a single-instance WordPress Deployment. The WordPress container mounts the PersistentVolume at `/var/www/html` for website data files. The `WORDPRESS_DB_HOST` environment variable sets the name of the MySQL Service defined above, and WordPress will access the database by Service. The `WORDPRESS_DB_PASSWORD` environment variable sets the database password from the Secret kustomize generated.

application/wordpress/wordpress-deployment.yaml  Copy application/wordpress/wordpress-deployment.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  name: wordpress  labels:    app: wordpressspec:  ports:    - port: 80  selector:    app: wordpress    tier
```

1. Download the MySQL deployment configuration file.

   ```
   curl -LO https://k8s.io/examples/application/wordpress/mysql-deployment.yaml
   ```

2. Download the WordPress configuration file.

   ```
   curl -LO https://k8s.io/examples/application/wordpress/wordpress-deployment.yaml
   ```

3. Add them to `kustomization.yaml` file.

   ```
   cat <<EOF >>./kustomization.yaml
   resources:
     - mysql-deployment.yaml
     - wordpress-deployment.yaml
   EOF
   ```

Apply and Verify

The `kustomization.yaml` contains all the resources for deploying a WordPress site and a MySQL database. You can apply the directory by

```
kubectl apply -k ./
```

Now you can verify that all objects exist.

1. Verify that the Secret exists by running the following command:

   ```
   kubectl get secrets
   ```

   The response should be like this:

   ```
   NAME                    TYPE                                  DATA   AGE
   mysql-pass-c57bb4t7mf   Opaque                                1      9s
   ```

2. Verify that a PersistentVolume got dynamically provisioned.

   ```
   kubectl get pvc
   ```

   Note:

   It can take up to a few minutes for the PVs to be provisioned and bound.

   The response should be like this:

   ```
   NAME             STATUS    VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS       AGE
   mysql-pv-claim   Bound     pvc-8cbd7b2e-4044-11e9-b2bb-42010a800002   20Gi       RWO            standard           77s
   wp-pv-claim      Bound     pvc-8cd0df54-4044-11e9-b2bb-42010a800002   20Gi       RWO            standard           77s
   ```

3. Verify that the Pod is running by running the following command:

   ```
   kubectl get pods
   ```

Note:

It can take up to a few minutes for the Pod's Status to be `RUNNING`.

The response should be like this:

```
NAME                                READY     STATUS    RESTARTS   AGE
wordpress-mysql-1894417608-x5dzt    1/1       Running   0          40s
```

4. Verify that the Service is running by running the following command:

   ```
   kubectl get services wordpress
   ```

   The response should be like this:

   ```
   NAME        TYPE            CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
   wordpress   LoadBalancer    10.0.0.89    <pending>     80:32406/TCP   4m
   ```

   Note:

   Minikube can only expose Services through `NodePort`. The EXTERNAL-IP is always pending.

5. Run the following command to get the IP Address for the WordPress Service:

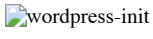
   ```
   minikube service wordpress --url
   ```

   The response should be like this:

   ```
   http://1.2.3.4:32406
   ```

6. Copy the IP address, and load the page in your browser to view your site.

   You should see the WordPress set up page similar to the following screenshot.

   
   wordpress-init

   Warning:

   Do not leave your WordPress installation on this page. If another user finds it, they can set up a website on your instance and use it to serve malicious content.

   Either install WordPress by creating a username and password or delete your instance.

Cleaning up

1. Run the following command to delete your Secret, Deployments, Services and PersistentVolumeClaims:

   ```
   kubectl delete -k ./
   ```

What's next

- Learn more about Introspection and Debugging
- Learn more about Jobs
- Learn more about Port Forwarding
- Learn how to Get a Shell to a Container

Example: Deploying PHP Guestbook application with Redis

This tutorial shows you how to build and deploy a simple *(not production ready)*, multi-tier web application using Kubernetes and Docker. This example consists of the following components:

- A single-instance Redis to store guestbook entries
- Multiple web frontend instances

Objectives

- Start up a Redis leader.
- Start up two Redis followers.
- Start up the guestbook frontend.
- Expose and view the Frontend Service.
- Clean up.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

Your Kubernetes server must be at or later than version v1.14.

To check the version, enter `kubectl version`.

Start up the Redis Database

The guestbook application uses Redis to store its data.

Creating the Redis Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica Redis Pod.

application/guestbook/redis-leader-deployment.yaml Copy application/guestbook/redis-leader-deployment.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: apps/v1kind: Deploymentmetadata:  name: redis-leader  labels:    app: redis    role: leader    tier: backendspec:  rep
```

1. Launch a terminal window in the directory you downloaded the manifest files.
2. Apply the Redis Deployment from the `redis-leader-deployment.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-leader-deployment.yaml
   ```

3. Query the list of Pods to verify that the Redis Pod is running:

   ```
   kubectl get pods
   ```

   The response should be similar to this:

   ```
   NAME                           READY   STATUS    RESTARTS   AGE
   redis-leader-fb76b4755-xjr2n   1/1     Running   0          13s
   ```

4. Run the following command to view the logs from the Redis leader Pod:

   ```
   kubectl logs -f deployment/redis-leader
   ```

Creating the Redis leader Service

The guestbook application needs to communicate to the Redis to write its data. You need to apply a Service to proxy the traffic to the Redis Pod. A Service defines a policy to access the Pods.

application/guestbook/redis-leader-service.yaml Copy application/guestbook/redis-leader-service.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: v1kind: Servicemetadata:  name: redis-leader  labels:    app: redis    role: leader    tier: backendspec:  ports:  - p
```

1. Apply the Redis Service from the following `redis-leader-service.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-leader-service.yaml
   ```

2. Query the list of Services to verify that the Redis Service is running:

   ```
   kubectl get service
   ```

   The response should be similar to this:

   ```
   NAME           TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
   kubernetes     ClusterIP   10.0.0.1     <none>        443/TCP    1m
   redis-leader   ClusterIP   10.103.78.24 <none>        6379/TCP   16s
   ```

Note:

This manifest file creates a Service named `redis-leader` with a set of labels that match the labels previously defined, so the Service routes network traffic to the Redis Pod.

Set up Redis followers

Although the Redis leader is a single Pod, you can make it highly available and meet traffic demands by adding a few Redis followers, or replicas.

application/guestbook/redis-follower-deployment.yaml Copy application/guestbook/redis-follower-deployment.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: apps/v1kind: Deploymentmetadata:  name: redis-follower  labels:    app: redis    role: follower    tier: backendspec:
```

1. Apply the Redis Deployment from the following `redis-follower-deployment.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-follower-deployment.yaml
   ```

2. Verify that the two Redis follower replicas are running by querying the list of Pods:

```
kubectl get pods
```

The response should be similar to this:

```
NAME                              READY   STATUS    RESTARTS   AGE
redis-follower-dddfbdcc9-82sfr    1/1     Running   0          37s
redis-follower-dddfbdcc9-qrt5k    1/1     Running   0          38s
redis-leader-fb76b4755-xjr2n      1/1     Running   0          11m
```

Creating the Redis follower service

The guestbook application needs to communicate with the Redis followers to read data. To make the Redis followers discoverable, you must set up another Service.

application/guestbook/redis-follower-service.yaml  Copy application/guestbook/redis-follower-service.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: v1kind: Servicemetadata:  name: redis-follower  labels:    app: redis    role: follower    tier: backendspec:  ports:
```

1. Apply the Redis Service from the following `redis-follower-service.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/redis-follower-service.yaml
   ```

2. Query the list of Services to verify that the Redis Service is running:

   ```
   kubectl get service
   ```

   The response should be similar to this:

   ```
   NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
   kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP    3d19h
   redis-follower   ClusterIP   10.110.162.42   <none>        6379/TCP   9s
   redis-leader     ClusterIP   10.103.78.24    <none>        6379/TCP   6m10s
   ```

Note:

This manifest file creates a Service named `redis-follower` with a set of labels that match the labels previously defined, so the Service routes network traffic to the Redis Pod.

Set up and Expose the Guestbook Frontend

Now that you have the Redis storage of your guestbook up and running, start the guestbook web servers. Like the Redis followers, the frontend is deployed using a Kubernetes Deployment.

The guestbook app uses a PHP frontend. It is configured to communicate with either the Redis follower or leader Services, depending on whether the request is a read or a write. The frontend exposes a JSON interface, and serves a jQuery-Ajax-based UX.

Creating the Guestbook Frontend Deployment

application/guestbook/frontend-deployment.yaml  Copy application/guestbook/frontend-deployment.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: apps/v1kind: Deploymentmetadata:  name: frontendspec:  replicas: 3  selector:    matchLabels:        app: guestbook
```

1. Apply the frontend Deployment from the `frontend-deployment.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-deployment.yaml
   ```

2. Query the list of Pods to verify that the three frontend replicas are running:

   ```
   kubectl get pods -l app=guestbook -l tier=frontend
   ```

   The response should be similar to this:

   ```
   NAME                        READY   STATUS    RESTARTS   AGE
   frontend-85595f5bf9-5tqhb   1/1     Running   0          47s
   frontend-85595f5bf9-qbzwm   1/1     Running   0          47s
   frontend-85595f5bf9-zchwc   1/1     Running   0          47s
   ```

Creating the Frontend Service

The `Redis` Services you applied is only accessible within the Kubernetes cluster because the default type for a Service is ClusterIP. `ClusterIP` provides a single IP address for the set of Pods the Service is pointing to. This IP address is accessible only within the cluster.

If you want guests to be able to access your guestbook, you must configure the frontend Service to be externally visible, so a client can request the Service from outside the Kubernetes cluster. However a Kubernetes user can use `kubectl port-forward` to access the service even though it uses a `ClusterIP`.

Note:

Some cloud providers, like Google Compute Engine or Google Kubernetes Engine, support external load balancers. If your cloud provider supports load balancers and you want to use it, uncomment `type: LoadBalancer`.
application/guestbook/frontend-service.yaml  Copy application/guestbook/frontend-service.yaml to clipboard

```
# SOURCE: https://cloud.google.com/kubernetes-engine/docs/tutorials/guestbook
apiVersion: v1kind: Servicemetadata:  name: frontend  labels:    app: guestbook    tier: frontendspec:  # if your cluster supports
```

1. Apply the frontend Service from the `frontend-service.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/guestbook/frontend-service.yaml
   ```

2. Query the list of Services to verify that the frontend Service is running:

   ```
   kubectl get services
   ```

   The response should be similar to this:

   ```
   NAME             TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)    AGE
   frontend         ClusterIP   10.97.28.230    <none>        80/TCP     19s
   kubernetes       ClusterIP   10.96.0.1       <none>        443/TCP    3d19h
   redis-follower   ClusterIP   10.110.162.42   <none>        6379/TCP   5m48s
   redis-leader     ClusterIP   10.103.78.24    <none>        6379/TCP   11m
   ```

Viewing the Frontend Service via `kubectl port-forward`

1. Run the following command to forward port `8080` on your local machine to port `80` on the service.

   ```
   kubectl port-forward svc/frontend 8080:80
   ```

   The response should be similar to this:

   ```
   Forwarding from 127.0.0.1:8080 -> 80
   Forwarding from [::1]:8080 -> 80
   ```

2. load the page http://localhost:8080 in your browser to view your guestbook.

Viewing the Frontend Service via `LoadBalancer`

If you deployed the `frontend-service.yaml` manifest with type: `LoadBalancer` you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

   ```
   kubectl get service frontend
   ```

   The response should be similar to this:

   ```
   NAME       TYPE           CLUSTER-IP      EXTERNAL-IP        PORT(S)        AGE
   frontend   LoadBalancer   10.51.242.136   109.197.92.229     80:32372/TCP   1m
   ```

2. Copy the external IP address, and load the page in your browser to view your guestbook.

Note:

Try adding some guestbook entries by typing in a message, and clicking Submit. The message you typed appears in the frontend. This message indicates that data is successfully added to Redis through the Services you created earlier.

Scale the Web Frontend

You can scale up or down as needed because your servers are defined as a Service that uses a Deployment controller.

1. Run the following command to scale up the number of frontend Pods:

   ```
   kubectl scale deployment frontend --replicas=5
   ```

2. Query the list of Pods to verify the number of frontend Pods running:

   ```
   kubectl get pods
   ```

   The response should look similar to this:

   ```
   NAME                             READY   STATUS    RESTARTS   AGE
   frontend-85595f5bf9-5df5m        1/1     Running   0          83s
   frontend-85595f5bf9-7zmg5        1/1     Running   0          83s
   frontend-85595f5bf9-cpskg        1/1     Running   0          15m
   frontend-85595f5bf9-l2l54        1/1     Running   0          14m
   frontend-85595f5bf9-l9c8z        1/1     Running   0          14m
   redis-follower-dddfbdcc9-82sfr   1/1     Running   0          97m
   redis-follower-dddfbdcc9-qrt5k   1/1     Running   0          97m
   redis-leader-fb76b4755-xjr2n     1/1     Running   0          108m
   ```

3. Run the following command to scale down the number of frontend Pods:

   ```
   kubectl scale deployment frontend --replicas=2
   ```

4. Query the list of Pods to verify the number of frontend Pods running:

   ```
   kubectl get pods
   ```

   The response should look similar to this:

   ```
   NAME                             READY   STATUS    RESTARTS   AGE
   frontend-85595f5bf9-cpskg        1/1     Running   0          16m
   frontend-85595f5bf9-l9c8z        1/1     Running   0          15m
   redis-follower-dddfbdcc9-82sfr   1/1     Running   0          98m
   ```

```
redis-follower-dddfbdcc9-qrt5k   1/1     Running   0          98m
redis-leader-fb76b4755-xjr2n     1/1     Running   0          109m
```

Cleaning up

Deleting the Deployments and Services also deletes any running Pods. Use labels to delete multiple resources with one command.

1. Run the following commands to delete all Pods, Deployments, and Services.

   ```
   kubectl delete deployment -l app=redis
   kubectl delete service -l app=redis
   kubectl delete deployment frontend
   kubectl delete service frontend
   ```

   The response should look similar to this:

   ```
   deployment.apps "redis-follower" deleted
   deployment.apps "redis-leader" deleted
   deployment.apps "frontend" deleted
   service "frontend" deleted
   ```

2. Query the list of Pods to verify that no Pods are running:

   ```
   kubectl get pods
   ```

   The response should look similar to this:

   ```
   No resources found in default namespace.
   ```

What's next

- Complete the Kubernetes Basics Interactive Tutorials
- Use Kubernetes to create a blog using Persistent Volumes for MySQL and Wordpress
- Read more about connecting applications with services
- Read more about using labels effectively

Stateful Applications

StatefulSet Basics

Example: Deploying WordPress and MySQL with Persistent Volumes

Example: Deploying Cassandra with a StatefulSet

Running ZooKeeper, A Distributed System Coordinator

Using Source IP

Applications running in a Kubernetes cluster find and communicate with each other, and the outside world, through the Service abstraction. This document explains what happens to the source IP of packets sent to different types of Services, and how you can toggle this behavior according to your needs.

Before you begin

Terminology

This document makes use of the following terms:

NAT
: Network address translation

Source NAT
: Replacing the source IP on a packet; in this page, that usually means replacing with the IP address of a node.

Destination NAT
: Replacing the destination IP on a packet; in this page, that usually means replacing with the IP address of a Pod

VIP
: A virtual IP address, such as the one assigned to every Service in Kubernetes

kube-proxy
: A network daemon that orchestrates Service VIP management on every node

Prerequisites

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

- iximiuz Labs

- Killercoda
- KodeKloud
- Play with Kubernetes

The examples use a small nginx webserver that echoes back the source IP of requests it receives through an HTTP header. You can create it as follows:

Note:

The image in the following command only runs on AMD64 architectures.

```
kubectl create deployment source-ip-app --image=registry.k8s.io/echoserver:1.10
```

The output is:

```
deployment.apps/source-ip-app created
```

Objectives

- Expose a simple application through various types of Services
- Understand how each Service type handles source IP NAT
- Understand the tradeoffs involved in preserving source IP

Source IP for Services with `Type=ClusterIP`

Packets sent to ClusterIP from within the cluster are never source NAT'd if you're running kube-proxy in iptables mode, (the default). You can query the kube-proxy mode by fetching `http://localhost:10249/proxyMode` on the node where kube-proxy is running.

```
kubectl get nodes
```

The output is similar to this:

```
NAME                          STATUS     ROLES    AGE     VERSION
kubernetes-node-6jst   Ready      <none>   2h      v1.13.0
kubernetes-node-cx31   Ready      <none>   2h      v1.13.0
kubernetes-node-jj1t   Ready      <none>   2h      v1.13.0
```

Get the proxy mode on one of the nodes (kube-proxy listens on port 10249):

```
# Run this in a shell on the node you want to query.
curl http://localhost:10249/proxyMode
```

The output is:

```
iptables
```

You can test source IP preservation by creating a Service over the source IP app:

```
kubectl expose deployment source-ip-app --name=clusterip --port=80 --target-port=8080
```

The output is:

```
service/clusterip exposed
```

```
kubectl get svc clusterip
```

The output is similar to:

```
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
clusterip    ClusterIP   10.0.170.92   <none>        80/TCP    51s
```

And hitting the `ClusterIP` from a pod in the same cluster:

```
kubectl run busybox -it --image=busybox:1.28 --restart=Never --rm
```

The output is similar to this:

```
Waiting for pod default/busybox to be running, status is Pending, pod ready: false
If you don't see a command prompt, try pressing enter.
```

You can then run a command inside that Pod:

```
# Run this inside the terminal from "kubectl run"
ip addr
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc noqueue
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.8/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::188a:84ff:feb0:26a5/64 scope link
       valid_lft forever preferred_lft forever
```

…then use `wget` to query the local webserver

```
# Replace "10.0.170.92" with the IPv4 address of the Service named "clusterip"
wget -qO - 10.0.170.92

CLIENT VALUES:
client_address=10.244.3.8
command=GET
...
```

The `client_address` is always the client pod's IP address, whether the client pod and server pod are in the same node or in different nodes.

Source IP for Services with `Type=NodePort`

Packets sent to Services with `Type=NodePort` are source NAT'd by default. You can test this by creating a `NodePort` Service:

```
kubectl expose deployment source-ip-app --name=nodeport --port=80 --target-port=8080 --type=NodePort
```

The output is:

```
service/nodeport exposed
```

```
NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].nodePort}" services nodeport)
NODES=$(kubectl get nodes -o jsonpath='{ $.items[*].status.addresses[?(@.type=="InternalIP")].address }')
```

If you're running on a cloud provider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

```
for node in $NODES; do curl -s $node:$NODEPORT | grep -i client_address; done
```

The output is similar to:

```
client_address=10.180.1.1
client_address=10.240.0.5
client_address=10.240.0.3
```

Note that these are not the correct client IPs, they're cluster internal IPs. This is what happens:

- Client sends packet to `node2:nodePort`
- `node2` replaces the source IP address (SNAT) in the packet with its own IP address
- `node2` replaces the destination IP on the packet with the pod IP
- packet is routed to node 1, and then to the endpoint
- the pod's reply is routed back to node2
- the pod's reply is sent back to the client

Visually:

Figure. Source IP Type=NodePort using SNAT

To avoid this, Kubernetes has a feature to preserve the client source IP. If you set `service.spec.externalTrafficPolicy` to the value `Local`, kube-proxy only proxies proxy requests to local endpoints, and does not forward traffic to other nodes. This approach preserves the original source IP address. If there are no local endpoints, packets sent to the node are dropped, so you can rely on the correct source-ip in any packet processing rules you might apply a packet that make it through to the endpoint.

Set the `service.spec.externalTrafficPolicy` field as follows:

```
kubectl patch svc nodeport -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

The output is:

```
service/nodeport patched
```

Now, re-run the test:

```
for node in $NODES; do curl --connect-timeout 1 -s $node:$NODEPORT | grep -i client_address; done
```

The output is similar to:

```
client_address=198.51.100.79
```

Note that you only got one reply, with the *right* client IP, from the one node on which the endpoint pod is running.

This is what happens:

- client sends packet to `node2:nodePort`, which doesn't have any endpoints
- packet is dropped
- client sends packet to `node1:nodePort`, which *does* have endpoints
- node1 routes packet to endpoint with the correct source IP

Visually:

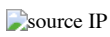


Figure. Source IP Type=NodePort preserves client source IP address

Source IP for Services with `Type=LoadBalancer`

Packets sent to Services with `Type=LoadBalancer` are source NAT'd by default, because all schedulable Kubernetes nodes in the `Ready` state are eligible for load-balanced traffic. So if packets arrive at a node without an endpoint, the system proxies it to a node *with* an endpoint, replacing the source IP on the packet with the IP of the node (as described in the previous section).

You can test this by exposing the source-ip-app through a load balancer:

```
kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-port=8080 --type=LoadBalancer
```

The output is:

```
service/loadbalancer exposed
```

Print out the IP addresses of the Service:

```
kubectl get svc loadbalancer
```

The output is similar to this:

```
NAME           TYPE           CLUSTER-IP    EXTERNAL-IP       PORT(S)   AGE
loadbalancer   LoadBalancer   10.0.65.118   203.0.113.140     80/TCP    5m
```

Next, send a request to this Service's external-ip:

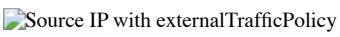
```
curl 203.0.113.140
```

The output is similar to this:

```
CLIENT VALUES:
client_address=10.240.0.5
...
```

However, if you're running on Google Kubernetes Engine/GCE, setting the same `service.spec.externalTrafficPolicy` field to `Local` forces nodes *without* Service endpoints to remove themselves from the list of nodes eligible for loadbalanced traffic by deliberately failing health checks.

Visually:


Source IP with externalTrafficPolicy

You can test this by setting the annotation:

```
kubectl patch svc loadbalancer -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

You should immediately see the `service.spec.healthCheckNodePort` field allocated by Kubernetes:

```
kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort
```

The output is similar to this:

```
  healthCheckNodePort: 32122
```

The `service.spec.healthCheckNodePort` field points to a port on every node serving the health check at `/healthz`. You can test this:

```
kubectl get pod -o wide -l app=source-ip-app
```

The output is similar to this:

```
NAME                              READY     STATUS    RESTARTS   AGE       IP             NODE
source-ip-app-826191075-qehz4   1/1       Running   0          20h       10.180.1.136   kubernetes-node-6jst
```

Use `curl` to fetch the `/healthz` endpoint on various nodes:

```
# Run this locally on a node you choose
curl localhost:32122/healthz
```

```
1 Service Endpoints found
```

On a different node you might get a different result:

```
# Run this locally on a node you choose
curl localhost:32122/healthz
```

```
No Service Endpoints Found
```

A controller running on the control plane is responsible for allocating the cloud load balancer. The same controller also allocates HTTP health checks pointing to this port/path on each node. Wait about 10 seconds for the 2 nodes without endpoints to fail health checks, then use `curl` to query the IPv4 address of the load balancer:

```
curl 203.0.113.140
```

The output is similar to this:

```
CLIENT VALUES:
client_address=198.51.100.79
...
```

Cross-platform support

Only some cloud providers offer support for source IP preservation through Services with `Type=LoadBalancer`. The cloud provider you're running on might fulfill the request for a loadbalancer in a few different ways:

1. With a proxy that terminates the client connection and opens a new connection to your nodes/endpoints. In such cases the source IP will always be that of the cloud LB, not that of the client.

2. With a packet forwarder, such that requests from the client sent to the loadbalancer VIP end up at the node with the source IP of the client, not an intermediate proxy.

Load balancers in the first category must use an agreed upon protocol between the loadbalancer and backend to communicate the true client IP such as the HTTP Forwarded or X-FORWARDED-FOR headers, or the proxy protocol. Load balancers in the second category can leverage the feature described above by creating an HTTP health check pointing at the port stored in the `service.spec.healthCheckNodePort` field on the Service.

Cleaning up

Delete the Services:

```
kubectl delete svc -l app=source-ip-app
```

Delete the Deployment, ReplicaSet and Pod:

```
kubectl delete deployment source-ip-app
```

What's next

- Learn more about connecting applications via services
- Read how to Create an External Load Balancer

Apply Pod Security Standards at the Cluster Level

Note

This tutorial applies only for new clusters.

Pod Security is an admission controller that carries out checks against the Kubernetes Pod Security Standards when new pods are created. It is a feature GA'ed in v1.25. This tutorial shows you how to enforce the `baseline` Pod Security Standard at the cluster level which applies a standard configuration to all namespaces in a cluster.

To apply Pod Security Standards to specific namespaces, refer to Apply Pod Security Standards at the namespace level.

If you are running a version of Kubernetes other than v1.34, check the documentation for that version.

Before you begin

Install the following on your workstation:

- kind
- kubectl

This tutorial demonstrates what you can configure for a Kubernetes cluster that you fully control. If you are learning how to configure Pod Security Admission for a managed cluster where you are not able to configure the control plane, read Apply Pod Security Standards at the namespace level.

Choose the right Pod Security Standard to apply

Pod Security Admission lets you apply built-in Pod Security Standards with the following modes: `enforce`, `audit`, and `warn`.

To gather information that helps you to choose the Pod Security Standards that are most appropriate for your configuration, do the following:

1. Create a cluster with no Pod Security Standards applied:

   ```
   kind create cluster --name psa-wo-cluster-pss
   ```

   The output is similar to:

   ```
   Creating cluster "psa-wo-cluster-pss" ...
   ✓ Ensuring node image (kindest/node:v1.34.0) 🖼
   ✓ Preparing nodes 📦
   ✓ Writing configuration 📜
   ✓ Starting control-plane 🕹️
   ✓ Installing CNI 🔌
   ✓ Installing StorageClass 💾
   Set kubectl context to "kind-psa-wo-cluster-pss"
   You can now use your cluster with:

   kubectl cluster-info --context kind-psa-wo-cluster-pss

   Thanks for using kind! 😊
   ```

2. Set the kubectl context to the new cluster:

   ```
   kubectl cluster-info --context kind-psa-wo-cluster-pss
   ```

   The output is similar to this:

   ```
   Kubernetes control plane is running at https://127.0.0.1:61350

   CoreDNS is running at https://127.0.0.1:61350/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

   To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
   ```

3. Get a list of namespaces in the cluster:

   ```
   kubectl get ns
   ```

   The output is similar to this:

   ```
   NAME                 STATUS   AGE
   default              Active   9m30s
   kube-node-lease      Active   9m32s
   kube-public          Active   9m32s
   kube-system          Active   9m32s
   local-path-storage   Active   9m26s
   ```

4. Use `--dry-run=server` to understand what happens when different Pod Security Standards are applied:

   1. Privileged

      ```
      kubectl label --dry-run=server --overwrite ns --all \
      pod-security.kubernetes.io/enforce=privileged
      ```

      The output is similar to:

      ```
      namespace/default labeled
      namespace/kube-node-lease labeled
      namespace/kube-public labeled
      namespace/kube-system labeled
      namespace/local-path-storage labeled
      ```

   2. Baseline

      ```
      kubectl label --dry-run=server --overwrite ns --all \
      pod-security.kubernetes.io/enforce=baseline
      ```

      The output is similar to:

      ```
      namespace/default labeled
      namespace/kube-node-lease labeled
      namespace/kube-public labeled
      Warning: existing pods in namespace "kube-system" violate the new PodSecurity enforce level "baseline:latest"
      Warning: etcd-psa-wo-cluster-pss-control-plane (and 3 other pods): host namespaces, hostPath volumes
      Warning: kindnet-vzj42: non-default capabilities, host namespaces, hostPath volumes
      Warning: kube-proxy-m6hwf: host namespaces, hostPath volumes, privileged
      namespace/kube-system labeled
      namespace/local-path-storage labeled
      ```

   3. Restricted

      ```
      kubectl label --dry-run=server --overwrite ns --all \
      pod-security.kubernetes.io/enforce=restricted
      ```

      The output is similar to:

      ```
      namespace/default labeled
      namespace/kube-node-lease labeled
      namespace/kube-public labeled
      Warning: existing pods in namespace "kube-system" violate the new PodSecurity enforce level "restricted:latest"
      Warning: coredns-7bb9c7b568-hsptc (and 1 other pod): unrestricted capabilities, runAsNonRoot != true, seccompProfile
      Warning: etcd-psa-wo-cluster-pss-control-plane (and 3 other pods): host namespaces, hostPath volumes, allowPrivilegeEsca
      Warning: kindnet-vzj42: non-default capabilities, host namespaces, hostPath volumes, allowPrivilegeEscalation != false,
      Warning: kube-proxy-m6hwf: host namespaces, hostPath volumes, privileged, allowPrivilegeEscalation != false, unrestricte
      namespace/kube-system labeled
      Warning: existing pods in namespace "local-path-storage" violate the new PodSecurity enforce level "restricted:latest"
      Warning: local-path-provisioner-d6d9f7ffc-lw9lh: allowPrivilegeEscalation != false, unrestricted capabilities, runAsNonR
      namespace/local-path-storage labeled
      ```

From the previous output, you'll notice that applying the `privileged` Pod Security Standard shows no warnings for any namespaces. However, `baseline` and `restricted` standards both have warnings, specifically in the `kube-system` namespace.

Set modes, versions and standards

In this section, you apply the following Pod Security Standards to the `latest` version:

- `baseline` standard in `enforce` mode.
- `restricted` standard in `warn` and `audit` mode.

The `baseline` Pod Security Standard provides a convenient middle ground that allows keeping the exemption list short and prevents known privilege escalations.

Additionally, to prevent pods from failing in `kube-system`, you'll exempt the namespace from having Pod Security Standards applied.

When you implement Pod Security Admission in your own environment, consider the following:

1. Based on the risk posture applied to a cluster, a stricter Pod Security Standard like `restricted` might be a better choice.

2. Exempting the `kube-system` namespace allows pods to run as `privileged` in this namespace. For real world use, the Kubernetes project strongly recommends that you apply strict RBAC policies that limit access to `kube-system`, following the principle of least privilege. To implement the preceding standards, do the following:

3. Create a configuration file that can be consumed by the Pod Security Admission Controller to implement these Pod Security Standards:

   ```
   mkdir -p /tmp/pss
   cat <<EOF > /tmp/pss/cluster-level-pss.yaml
   apiVersion: apiserver.config.k8s.io/v1
   kind: AdmissionConfiguration
   plugins:
   - name: PodSecurity
     configuration:
       apiVersion: pod-security.admission.config.k8s.io/v1
       kind: PodSecurityConfiguration
       defaults:
         enforce: "baseline"
         enforce-version: "latest"
         audit: "restricted"
         audit-version: "latest"
         warn: "restricted"
         warn-version: "latest"
       exemptions:
         usernames: []
         runtimeClasses: []
         namespaces: [kube-system]
   EOF
   ```

   Note:

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

4. Configure the API server to consume this file during cluster creation:

   ```
   cat <<EOF > /tmp/pss/cluster-config.yaml
   kind: Cluster
   apiVersion: kind.x-k8s.io/v1alpha4
   nodes:
   - role: control-plane
     kubeadmConfigPatches:
     - |
       kind: ClusterConfiguration
       apiServer:
           extraArgs:
             admission-control-config-file: /etc/config/cluster-level-pss.yaml
           extraVolumes:
             - name: accf
               hostPath: /etc/config
               mountPath: /etc/config
               readOnly: false
               pathType: "DirectoryOrCreate"
     extraMounts:
     - hostPath: /tmp/pss
       containerPath: /etc/config
       # optional: if set, the mount is read-only.
       # default false
       readOnly: false
       # optional: if set, the mount needs SELinux relabeling.
       # default false
       selinuxRelabel: false
       # optional: set propagation mode (None, HostToContainer or Bidirectional)
       # see https://kubernetes.io/docs/concepts/storage/volumes/#mount-propagation
       # default None
       propagation: None
   EOF
   ```

   Note:

   If you use Docker Desktop with *kind* on macOS, you can add `/tmp` as a Shared Directory under the menu item **Preferences > Resources > File Sharing**.

5. Create a cluster that uses Pod Security Admission to apply these Pod Security Standards:

   ```
   kind create cluster --name psa-with-cluster-pss --config /tmp/pss/cluster-config.yaml
   ```

   The output is similar to this:

   ```
   Creating cluster "psa-with-cluster-pss" ...
    ✓ Ensuring node image (kindest/node:v1.34.0) 🖼
    ✓ Preparing nodes 📦
    ✓ Writing configuration 📜
    ✓ Starting control-plane 🕹️
    ✓ Installing CNI 🔌
   ```

```
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-psa-with-cluster-pss"
You can now use your cluster with:

kubectl cluster-info --context kind-psa-with-cluster-pss

Have a question, bug, or feature request? Let us know! https://kind.sigs.k8s.io/#community 🙂
```

6. Point kubectl to the cluster:

   ```
   kubectl cluster-info --context kind-psa-with-cluster-pss
   ```

   The output is similar to this:

   ```
   Kubernetes control plane is running at https://127.0.0.1:63855
   CoreDNS is running at https://127.0.0.1:63855/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

   To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
   ```

7. Create a Pod in the default namespace:

   security/example-baseline-pod.yaml Copy security/example-baseline-pod.yaml to clipboard

   ```
   apiVersion: v1

   kind: Podmetadata:  name: nginxspec:  containers:    - image: nginx      name: nginx      ports:        - containerPort: 80


   kubectl apply -f https://k8s.io/examples/security/example-baseline-pod.yaml
   ```

   The pod is started normally, but the output includes a warning:

   ```
   Warning: would violate PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container "nginx" must set securit
   pod/nginx created
   ```

Clean up

Now delete the clusters which you created above by running the following command:

```
kind delete cluster --name psa-with-cluster-pss
```

```
kind delete cluster --name psa-wo-cluster-pss
```

What's next

- Run a shell script to perform all the preceding steps at once:
  1. Create a Pod Security Standards based cluster level Configuration
  2. Create a file to let API server consume this configuration
  3. Create a cluster that creates an API server with this configuration
  4. Set kubectl context to this new cluster
  5. Create a minimal pod yaml file
  6. Apply this file to create a Pod in the new cluster
- Pod Security Admission
- Pod Security Standards
- Apply Pod Security Standards at the namespace level

Exposing an External IP Address to Access an Application in a Cluster

This page shows how to create a Kubernetes Service object that exposes an external IP address.

Before you begin

- Install kubectl.
- Use a cloud provider like Google Kubernetes Engine or Amazon Web Services to create a Kubernetes cluster. This tutorial creates an external load balancer, which requires a cloud provider.
- Configure `kubectl` to communicate with your Kubernetes API server. For instructions, see the documentation for your cloud provider.

Objectives

- Run five instances of a Hello World application.
- Create a Service object that exposes an external IP address.
- Use the Service object to access the running application.

Creating a service for an application running in five pods

1. Run a Hello World application in your cluster:

   service/load-balancer-example.yaml Copy service/load-balancer-example.yaml to clipboard

   ```
   apiVersion: apps/v1
   ```

```
kind: Deploymentmetadata:  labels:    app.kubernetes.io/name: load-balancer-example  name: hello-worldspec:  replicas: 5  sel
```

```
kubectl apply -f https://k8s.io/examples/service/load-balancer-example.yaml
```

The preceding command creates a Deployment and an associated ReplicaSet. The ReplicaSet has five Pods each of which runs the Hello World application.

2. Display information about the Deployment:

```
kubectl get deployments hello-world
kubectl describe deployments hello-world
```

3. Display information about your ReplicaSet objects:

```
kubectl get replicasets
kubectl describe replicasets
```

4. Create a Service object that exposes the deployment:

```
kubectl expose deployment hello-world --type=LoadBalancer --name=my-service
```

5. Display information about the Service:

```
kubectl get services my-service
```

The output is similar to:

```
NAME         TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)    AGE
my-service   LoadBalancer   10.3.245.137   104.198.205.71   8080/TCP   54s
```

Note:

The `type=LoadBalancer` service is backed by external cloud providers, which is not covered in this example. Please refer to setting `type: LoadBalancer` for your Service for the details.

Note:

If the external IP address is shown as <pending>, wait for a minute and enter the same command again.

6. Display detailed information about the Service:

```
kubectl describe services my-service
```

The output is similar to:

```
Name:           my-service
Namespace:      default
Labels:         app.kubernetes.io/name=load-balancer-example
Annotations:    <none>
Selector:       app.kubernetes.io/name=load-balancer-example
Type:           LoadBalancer
IP:             10.3.245.137
LoadBalancer Ingress:   104.198.205.71
Port:           <unset> 8080/TCP
NodePort:       <unset> 32377/TCP
Endpoints:      10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...
Session Affinity:   None
Events:         <none>
```

Make a note of the external IP address (`LoadBalancer Ingress`) exposed by your service. In this example, the external IP address is 104.198.205.71. Also note the value of `Port` and `NodePort`. In this example, the `Port` is 8080 and the `NodePort` is 32377.

7. In the preceding output, you can see that the service has several endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more. These are internal addresses of the pods that are running the Hello World application. To verify these are pod addresses, enter this command:

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

The output is similar to:

```
NAME                         ...  IP         NODE
hello-world-2895499144-1jaz9 ...  10.0.1.6   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-2e5uh ...  10.0.1.8   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-9m4h1 ...  10.0.0.6   gke-cluster-1-default-pool-e0b8d269-5v7a
hello-world-2895499144-o4z13 ...  10.0.1.7   gke-cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-segjf ...  10.0.2.5   gke-cluster-1-default-pool-e0b8d269-cpuc
```

8. Use the external IP address (`LoadBalancer Ingress`) to access the Hello World application:

```
curl http://<external-ip>:<port>
```

where `<external-ip>` is the external IP address (`LoadBalancer Ingress`) of your Service, and `<port>` is the value of `Port` in your Service description. If you are using minikube, typing `minikube service my-service` will automatically open the Hello World application in a browser.

The response to a successful request is a hello message:

```
Hello, world!
Version: 2.0.0
Hostname: 0bd46b45f32f
```

Cleaning up

To delete the Service, enter this command:

```
kubectl delete services my-service
```

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

```
kubectl delete deployment hello-world
```

What's next

Learn more about connecting applications with services.

Example: Deploying Cassandra with a StatefulSet

This tutorial shows you how to run Apache Cassandra on Kubernetes. Cassandra, a database, needs persistent storage to provide data durability (application *state*). In this example, a custom Cassandra seed provider lets the database discover new Cassandra instances as they join the Cassandra cluster.

StatefulSets make it easier to deploy stateful applications into your Kubernetes cluster. For more information on the features used in this tutorial, see StatefulSet.

Note:

Cassandra and Kubernetes both use the term *node* to mean a member of a cluster. In this tutorial, the Pods that belong to the StatefulSet are Cassandra nodes and are members of the Cassandra cluster (called a *ring*). When those Pods run in your Kubernetes cluster, the Kubernetes control plane schedules those Pods onto Kubernetes Nodes.

When a Cassandra node starts, it uses a *seed list* to bootstrap discovery of other nodes in the ring. This tutorial deploys a custom Cassandra seed provider that lets the database discover new Cassandra Pods as they appear inside your Kubernetes cluster.

Objectives

- Create and validate a Cassandra headless Service.
- Use a StatefulSet to create a Cassandra ring.
- Validate the StatefulSet.
- Modify the StatefulSet.
- Delete the StatefulSet and its Pods.

Before you begin

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

- iximiuz Labs
- Killercoda
- KodeKloud
- Play with Kubernetes

To complete this tutorial, you should already have a basic familiarity with Pods, Services, and StatefulSets.

Additional Minikube setup instructions

Caution:

Minikube defaults to 2048MB of memory and 2 CPU. Running Minikube with the default resource configuration results in insufficient resource errors during this tutorial. To avoid these errors, start Minikube with the following settings:

```
minikube start --memory 5120 --cpus=4
```

Creating a headless Service for Cassandra

In Kubernetes, a Service describes a set of Pods that perform the same task.

The following Service is used for DNS lookups between Cassandra Pods and clients within your cluster:

application/cassandra/cassandra-service.yaml  Copy application/cassandra/cassandra-service.yaml to clipboard

```
apiVersion: v1
kind: Servicemetadata:  labels:    app: cassandra  name: cassandraspec:  clusterIP: None  ports:  - port: 9042  selector:    app: 
```

Create a Service to track all Cassandra StatefulSet members from the `cassandra-service.yaml` file:

```
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-service.yaml
```

Validating (optional)

Get the Cassandra Service.

```
kubectl get svc cassandra
```

The response is

```
NAME        TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
cassandra   ClusterIP   None         <none>        9042/TCP   45s
```

If you don't see a Service named `cassandra`, that means creation failed. Read Debug Services for help troubleshooting common issues.

Using a StatefulSet to create a Cassandra ring

The StatefulSet manifest, included below, creates a Cassandra ring that consists of three Pods.

Note:

This example uses the default provisioner for Minikube. Please update the following StatefulSet for the cloud you are working with.
application/cassandra/cassandra-statefulset.yaml Copy application/cassandra/cassandra-statefulset.yaml to clipboard

```
apiVersion: apps/v1
kind: StatefulSetmetadata:  name: cassandra  labels:    app: cassandraspec:  serviceName: cassandra  replicas: 3  selector:    mat
```

Create the Cassandra StatefulSet from the `cassandra-statefulset.yaml` file:

```
# Use this if you are able to apply cassandra-statefulset.yaml unmodified
kubectl apply -f https://k8s.io/examples/application/cassandra/cassandra-statefulset.yaml
```

If you need to modify `cassandra-statefulset.yaml` to suit your cluster, download https://k8s.io/examples/application/cassandra/cassandra-statefulset.yaml and then apply that manifest, from the folder you saved the modified version into:

```
# Use this if you needed to modify cassandra-statefulset.yaml locally
kubectl apply -f cassandra-statefulset.yaml
```

Validating the Cassandra StatefulSet

1. Get the Cassandra StatefulSet:

   ```
   kubectl get statefulset cassandra
   ```

   The response should be similar to:

   ```
   NAME        DESIRED   CURRENT   AGE
   cassandra   3         0         13s
   ```

   The `StatefulSet` resource deploys Pods sequentially.

2. Get the Pods to see the ordered creation status:

   ```
   kubectl get pods -l="app=cassandra"
   ```

   The response should be similar to:

   ```
   NAME          READY     STATUS              RESTARTS   AGE
   cassandra-0   1/1       Running             0          1m
   cassandra-1   0/1       ContainerCreating   0          8s
   ```

   It can take several minutes for all three Pods to deploy. Once they are deployed, the same command returns output similar to:

   ```
   NAME          READY     STATUS    RESTARTS   AGE
   cassandra-0   1/1       Running   0          10m
   cassandra-1   1/1       Running   0          9m
   cassandra-2   1/1       Running   0          8m
   ```

3. Run the Cassandra nodetool inside the first Pod, to display the status of the ring.

   ```
   kubectl exec -it cassandra-0 -- nodetool status
   ```

   The response should look something like:

   ```
   Datacenter: DC1-K8Demo
   ======================
   Status=Up/Down
   |/ State=Normal/Leaving/Joining/Moving
   --  Address     Load       Tokens       Owns (effective)  Host ID                               Rack
   UN  172.17.0.5  83.57 KiB  32           74.0%             e2dd09e6-d9d3-477e-96c5-45094c08db0f  Rack1-K8Demo
   UN  172.17.0.4  101.04 KiB  32           58.8%             f89d6835-3a42-4419-92b3-0e62cae1479c  Rack1-K8Demo
   UN  172.17.0.6  84.74 KiB  32           67.1%             a6a1e8c2-3dc5-4417-b1a0-26507af2aaad  Rack1-K8Demo
   ```

Modifying the Cassandra StatefulSet

Use `kubectl edit` to modify the size of a Cassandra StatefulSet.

1. Run the following command:

   ```
   kubectl edit statefulset cassandra
   ```

   This command opens an editor in your terminal. The line you need to change is the `replicas` field. The following sample is an excerpt of the StatefulSet file:

   ```
   # Please edit the object below. Lines beginning with a '#' will be ignored,
   # and an empty file will abort the edit. If an error occurs while saving this file will be# reopened with the relevant failur
   ```

2. Change the number of replicas to 4, and then save the manifest.

   The StatefulSet now scales to run with 4 Pods.

3. Get the Cassandra StatefulSet to verify your change:

   ```
   kubectl get statefulset cassandra
   ```

   The response should be similar to:

   ```
   NAME        DESIRED   CURRENT   AGE
   cassandra   4         4         36m
   ```

Cleaning up

Deleting or scaling a StatefulSet down does not delete the volumes associated with the StatefulSet. This setting is for your safety because your data is more valuable than automatically purging all related StatefulSet resources.

Warning:

Depending on the storage class and reclaim policy, deleting the *PersistentVolumeClaims* may cause the associated volumes to also be deleted. Never assume you'll be able to access data if its volume claims are deleted.

1. Run the following commands (chained together into a single command) to delete everything in the Cassandra StatefulSet:

   ```
   grace=$(kubectl get pod cassandra-0 -o=jsonpath='{.spec.terminationGracePeriodSeconds}') \
     && kubectl delete statefulset -l app=cassandra \  && echo "Sleeping ${grace} seconds" 1>&2 \  && sleep $grace \  && kubectl
   ```

2. Run the following command to delete the Service you set up for Cassandra:

   ```
   kubectl delete service -l app=cassandra
   ```

Cassandra container environment variables

The Pods in this tutorial use the `gcr.io/google-samples/cassandra:v13` image from Google's container registry. The Docker image above is based on debian-base and includes OpenJDK 8.

This image includes a standard Cassandra installation from the Apache Debian repo. By using environment variables you can change values that are inserted into `cassandra.yaml`.

Environment variable	Default value
CASSANDRA_CLUSTER_NAME	'Test Cluster'
CASSANDRA_NUM_TOKENS	32
CASSANDRA_RPC_ADDRESS	0.0.0.0

What's next

- Learn how to Scale a StatefulSet.
- Learn more about the *KubernetesSeedProvider*
- See more custom Seed Provider Configurations