Tutorials

This section of the Kubernetes documentation contains tutorials. A tutorial shows how to accomplish a goal that is larger than a single task. Typically a tutorial has several sections, each of which has a sequence of steps. Before walking through each tutorial, you may want to bookmark the Standardized Glossary page for later references.

Basics

- Kubernetes Basics is an in-depth interactive tutorial that helps you understand the Kubernetes system and try out some basic Kubernetes features.
- Introduction to Kubernetes (edX)
- Hello Minikube

Configuration

- Example: Configuring a Java Microservice
- Configuring Redis Using a ConfigMap

Stateless Applications

- Exposing an External IP Address to Access an Application in a Cluster
- Example: Deploying PHP Guestbook application with Redis

Stateful Applications

- StatefulSet Basics
- Example: WordPress and MySQL with Persistent Volumes
- Example: Deploying Cassandra with Stateful Sets
- Running ZooKeeper, A CP Distributed System

Services

- Connecting Applications with Services
- Using Source IP

Security

- Apply Pod Security Standards at Cluster level
- Apply Pod Security Standards at Namespace level
- AppArmor

- seccomp

What's next

If you would like to write a tutorial, see Content Page Types for information about the tutorial page type.

1 - Hello Minikube

This tutorial shows you how to run a sample app on Kubernetes using minikube and Katacoda. Katacoda provides a free, in-browser Kubernetes environment.

> **Note:** You can also follow this tutorial if you've installed minikube locally. See minikube start for installation instructions.

Objectives

- Deploy a sample application to minikube.
- Run the app.
- View application logs.

Before you begin

This tutorial provides a container image that uses NGINX to echo back all the requests.

Create a minikube cluster

1. Click **Launch Terminal**.

   Launch Terminal

> **Note:** If you installed minikube locally, run `minikube start`. Before you run `minikube dashboard`, you should open a new terminal, start `minikube dashboard` there, and then switch back to the main terminal.

2. Open the Kubernetes dashboard in a browser:

   ```
   minikube dashboard
   ```

3. Katacoda environment only: At the top of the terminal pane, click the plus sign, and then click **Select port to view on Host 1**.
4. Katacoda environment only: Type `30000`, and then click **Display Port**.

> **Note:**

> The `dashboard` command enables the dashboard add-on and opens the proxy in the default web browser. You can create Kubernetes resources on the dashboard such as Deployment and Service.
>
> If you are running in an environment as root, see Open Dashboard with URL.
>
> By default, the dashboard is only accessible from within the internal Kubernetes virtual network. The `dashboard` command creates a temporary proxy to make the dashboard accessible from outside the Kubernetes virtual network.
>
> To stop the proxy, run `Ctrl+C` to exit the process. After the command exits, the dashboard remains running in the Kubernetes cluster. You can run the `dashboard` command again to create another proxy to access the dashboard.

Open Dashboard with URL

If you don't want to open a web browser, run the dashboard command with the `--url` flag to emit a URL:

```
minikube dashboard --url
```

Create a Deployment

A Kubernetes *Pod* is a group of one or more Containers, tied together for the purposes of administration and networking. The Pod in this tutorial has only one Container. A Kubernetes *Deployment* checks on the health of your Pod and restarts the Pod's Container if it terminates. Deployments are the recommended way to manage the creation and scaling of Pods.

1. Katacoda environment only: At the top of the terminal pane, click the plus sign, and then click **Open New Terminal**.
2. Use the `kubectl create` command to create a Deployment that manages a Pod. The Pod runs a Container based on the provided Docker image.

   ```
   kubectl create deployment hello-node --image=reg
   ```

3. View the Deployment:

```
kubectl get deployments
```

The output is similar to:

```
NAME         READY   UP-TO-DATE   AVAILABLE   AG
hello-node   1/1     1            1           1m
```

4. View the Pod:

```
kubectl get pods
```

The output is similar to:

```
NAME                          READY     STATUS
hello-node-5f76cf6ccf-br9b5   1/1       Running
```

5. View cluster events:

```
kubectl get events
```

6. View the `kubectl` configuration:

```
kubectl config view
```

> **Note:** For more information about `kubectl` commands, see the kubectl overview.

Create a Service

By default, the Pod is only accessible by its internal IP address within the Kubernetes cluster. To make the `hello-node` Container accessible from outside the Kubernetes virtual network, you have to expose the Pod as a Kubernetes *Service*.

1. Expose the Pod to the public internet using the `kubectl expose` command:

```
kubectl expose deployment hello-node --type=Load
```

The `--type=LoadBalancer` flag indicates that you want to expose your Service outside of the cluster.

The application code inside the image `registry.k8s.io/echoserver` only listens on TCP port 8080. If you used `kubectl expose` to expose a different port, clients could not connect to that other port.

2. View the Service you created:

```
kubectl get services
```

The output is similar to:

```
NAME         TYPE           CLUSTER-IP      EXTE
hello-node   LoadBalancer   10.108.144.78   <pen
kubernetes   ClusterIP      10.96.0.1       <non
```

On cloud providers that support load balancers, an external IP address would be provisioned to access the Service. On minikube, the `LoadBalancer` type makes the Service accessible through the `minikube service` command.

3. Run the following command:

```
minikube service hello-node
```

4. Katacoda environment only: Click the plus sign, and then click **Select port to view on Host 1**.

5. Katacoda environment only: Note the 5-digit port number displayed opposite to `8080` in services output. This port number is randomly generated and it can be different for you. Type your number in the port number text box, then click Display Port. Using the example from earlier, you would type `30369` .

   This opens up a browser window that serves your app and shows the app's response.

Enable addons

The minikube tool includes a set of built-in addons that can be enabled, disabled and opened in the local Kubernetes environment.

1. List the currently supported addons:

```
minikube addons list
```

The output is similar to:

```
addon-manager: enabled
dashboard: enabled
default-storageclass: enabled
efk: disabled
freshpod: disabled
gvisor: disabled
helm-tiller: disabled
ingress: disabled
ingress-dns: disabled
logviewer: disabled
metrics-server: disabled
nvidia-driver-installer: disabled
nvidia-gpu-device-plugin: disabled
registry: disabled
registry-creds: disabled
storage-provisioner: enabled
storage-provisioner-gluster: disabled
```

2. Enable an addon, for example, `metrics-server` :

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

The output is similar to:

```
The 'metrics-server' addon is enabled
```

3. View the Pod and Service you created:

```
kubectl get pod,svc -n kube-system
```

The output is similar to:

```
NAME                                        READ
pod/coredns-5644d7b6d9-mh9ll                1/1
pod/coredns-5644d7b6d9-pqd2t                1/1
pod/metrics-server-67fb648c5                1/1
pod/etcd-minikube                           1/1
pod/influxdb-grafana-b29w8                  2/2
pod/kube-addon-manager-minikube             1/1
pod/kube-apiserver-minikube                 1/1
pod/kube-controller-manager-minikube        1/1
pod/kube-proxy-rnlps                        1/1
pod/kube-scheduler-minikube                 1/1
pod/storage-provisioner                     1/1

NAME                             TYPE        CLUST
service/metrics-server           ClusterIP   10.96
service/kube-dns                 ClusterIP   10.96
service/monitoring-grafana       NodePort    10.99
service/monitoring-influxdb      ClusterIP   10.11
```

4. Disable `metrics-server`:

   ```
   minikube addons disable metrics-server
   ```

   The output is similar to:

   ```
   metrics-server was successfully disabled
   ```

Clean up

Now you can clean up the resources you created in your cluster:

```
kubectl delete service hello-node
kubectl delete deployment hello-node
```

Optionally, stop the Minikube virtual machine (VM):

```
minikube stop
```

Optionally, delete the Minikube VM:

```
minikube delete
```

What's next

- Learn more about Deployment objects.
- Learn more about Deploying applications.
- Learn more about Service objects.

2 - Learn Kubernetes Basics

Kubernetes Basics

This tutorial provides a walkthrough of the basics of the Kubernetes cluster orchestration system. Each module contains some background information on major Kubernetes features and concepts, and includes an interactive online tutorial. These interactive tutorials let you manage a simple cluster and its containerized applications for yourself.

Using the interactive tutorials, you can learn to:

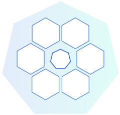
- Deploy a containerized application on a cluster.
- Scale the deployment.
- Update the containerized application with a new software version.
- Debug the containerized application.

The tutorials use Katacoda to run a virtual terminal in your web browser that runs Minikube, a small-scale local deployment of Kubernetes that can run anywhere. There's no need to install any software or configure anything; each interactive tutorial runs directly out of your web browser itself.

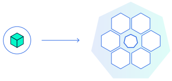
What can Kubernetes do for you?

With modern web services, users expect applications to be available 24/7, and developers expect to deploy new versions of those applications several times a day. Containerization helps package software to serve these goals, enabling applications to be released and updated without downtime. Kubernetes helps you make sure those containerized applications run where and when you want, and helps them find the resources and tools they need to work. Kubernetes is a production-ready, open source platform designed with Google's accumulated experience in container orchestration, combined with best-of-breed ideas from the community.

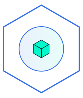
Kubernetes Basics Modules



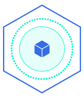
1. Create a Kubernetes cluster



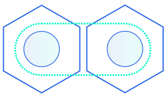
2. Deploy an app



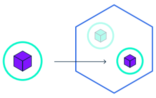
3. Explore your app



4. Expose your app publicly



5. Scale up your app



6. Update your app

2.1 - Create a Cluster

Learn about Kubernetes cluster and create a simple cluster using Minikube.

2.1.1 - Using Minikube to Create a Cluster

Objectives

- Learn what a Kubernetes cluster is.
- Learn what Minikube is.
- Start a Kubernetes cluster using an online terminal.

Kubernetes Clusters

Kubernetes coordinates a highly available cluster of computers that are connected to work as a single unit. The abstractions in Kubernetes allow you to deploy containerized applications to a cluster without tying them specifically to individual machines. To make use of this new model of deployment, applications need to be packaged in a way that decouples them from individual hosts: they need to be containerized. Containerized applications are more flexible and available than in past deployment models, where applications were installed directly onto specific machines as packages deeply integrated into the host. **Kubernetes automates the distribution and scheduling of application containers across a cluster in a more efficient way.** Kubernetes is an open-source platform and is production-ready.

A Kubernetes cluster consists of two types of resources:

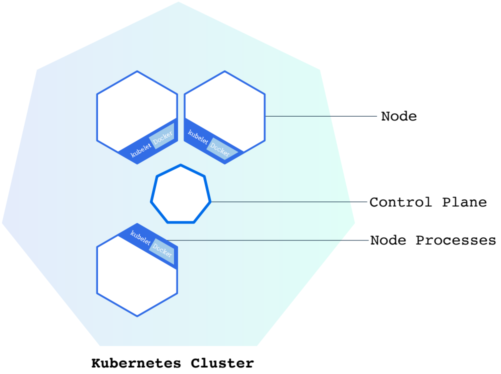
- The **Control Plane** coordinates the cluster
- **Nodes** are the workers that run applications

Summary:

- Kubernetes cluster
- Minikube

Kubernetes is a production-grade, open-source platform that orchestrates the placement (scheduling) and execution of application containers within and across computer clusters.

Cluster Diagram



The Control Plane is responsible for managing the cluster. The Control Plane coordinates all activities in your cluster, such as scheduling applications, maintaining applications' desired state, scaling applications, and rolling out new updates.

A node is a VM or a physical computer that serves as a worker machine in a Kubernetes cluster. Each node has a Kubelet, which is an agent for managing the node and communicating with the Kubernetes control plane. The node should also have tools for handling container operations, such as containerd or Docker. A Kubernetes cluster that handles production traffic should have a minimum of three nodes because if one node goes down, both an etcd member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.

Control Planes manage the cluster and the nodes that are used to host the running applications.

When you deploy applications on Kubernetes, you tell the control plane to start the application containers. The control plane schedules the containers to run on the cluster's nodes. **The nodes communicate with the control plane using the Kubernetes API**, which the control plane exposes. End users can also use the Kubernetes API directly to interact with the cluster.

A Kubernetes cluster can be deployed on either physical or virtual machines. To get started with Kubernetes development, you can use Minikube. Minikube is a lightweight Kubernetes implementation that creates a VM on your local machine and deploys a simple cluster containing only one node. Minikube is available for Linux, macOS, and Windows systems. The Minikube CLI provides basic bootstrapping operations for working with your cluster, including start, stop, status, and delete. For this tutorial, however, you'll use a provided online terminal with Minikube pre-installed.

Now that you know what Kubernetes is, let's go to the online tutorial and start our first cluster!

Start Interactive Tutorial ›

2.1.2 - Interactive Tutorial - Creating a Cluster

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 1 - Create a Kubernetes cluster

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Home Continue to Module 2 >

2.2 - Deploy an App

2.2.1 - Using kubectl to Create a Deployment

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 1 - Create a Kubernetes cluster

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Objectives

- Learn about application Deployments.
- Deploy your first app on Kubernetes with kubectl.

Kubernetes Deployments

Once you have a running Kubernetes cluster, you can deploy your containerized applications on top of it. To do so, you create a Kubernetes **Deployment** configuration. The Deployment instructs Kubernetes how to create and update instances of your application. Once you've created a Deployment, the Kubernetes control plane schedules the application instances included in that Deployment to run on individual Nodes in the cluster.

Summary:

- Deployments
- Kubectl

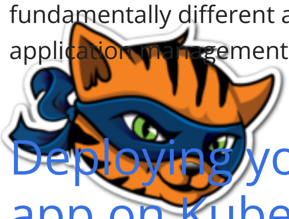
A Deployment is responsible for creating and updating instances of your application

Once the application instances are created, a Kubernetes Deployment Controller continuously monitors those instances. If the Node hosting an instance goes down or is deleted, the Deployment controller replaces the instance with an instance on another Node in the cluster. **This provides a self-healing mechanism to address machine failure or maintenance.**

In a pre-orchestration world, installation scripts would often be used to start applications, but they did not allow recovery from machine failure. By both creating your application instances and keeping them running across

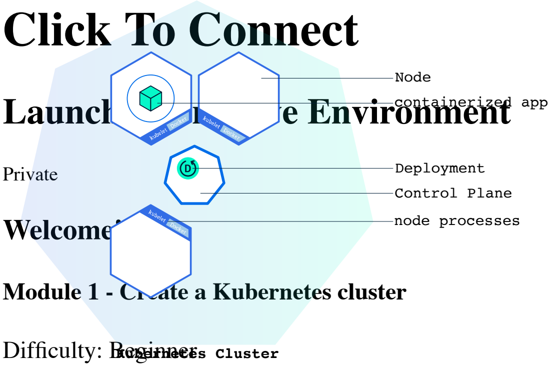
Click To Connect
Launch Interactive Environment
Private
Welcome!
Module 1 - Create a Kubernetes cluster
Difficulty: Beginner
Estimated Time: 10 minutes
The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube
The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the blocks of code followed by the

Nodes, Kubernetes Deployments provide a fundamentally different approach to application management.



Deploying your first app on Kubernetes

Click To Connect



Launch Interactive Environment

Private

Welcome!

Module 1 - Create a Kubernetes cluster

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

You can create and manage a Deployment by using the Kubernetes command line interface, **Kubectl**. Kubectl uses the Kubernetes API to interact with the cluster. In this module, you'll learn the most common Kubectl commands needed to create Deployments that run your applications on a Kubernetes cluster.

Applications need to be packaged into one of the supported container formats in order to be deployed on Kubernetes

When you create a Deployment, you'll need to specify the container image for your application and the number of replicas that you want to run. You can change that information later by updating your Deployment; Modules [5](#) and [6](#) of the bootcamp discuss how you can scale and update your Deployments.

For your first Deployment, you'll use a hello-node application packaged in a Docker container that uses NGINX to echo back all the requests. (If you didn't already try creating a hello-node application and deploying it using a container, you can do that first by following the instructions from the [Hello Minikube tutorial](#)).

Now that you know what Deployments are, let's go to the online tutorial and deploy our first app!

Click To Connect
Launch Interactive Environment
Private
Welcome!
Module 1 - Create a Kubernetes cluster
Difficulty: Beginner
Estimated Time: 10 minutes
The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube
The online terminal is a pre-configured Linux environment that can be used as a regular co ng on the blocks of code followed by the

Start Interactive Tutorial ›

2.2.2 - Interactive Tutorial - D ng an App

A Pod is the basic execution unit of a Kubernetes application. Each Pod represents a part of a workload that is running on your cluster. [Learn more about Pods](https://kubernetes.io/docs/concepts/workloads/pods/).

Click To Connect

Launch Interactive Environment

Private

Wel

M ernetes cluster

Difficulty:
Estimated Time: minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

Click To Connect

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Launch Interactive Environment

Private

Welcome!

Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

Click To Connect

The online terminal is a pre-configured Linux environment that can be used as a regular

Launch Interactive Environment

< Return to Module 1 Home

Continue to Module 3 >

Pr

W

Module 1 - Create a Kubernetes cluster

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

2.3 - Explore Your App

2. ing Pods and N

Objectives

- Learn about Kubernetes Pods.
- Learn about Kubernetes Nodes.
- Troubleshoot deployed applications.

Kubernetes Pods

When you created a Deployment in Module 2, Kubernetes created a **Pod** to host your application instance. A Pod is a Kubernetes abstraction that represents a group of one or more application containers (such as Docker), and some shared resources for those containers. Those resources include:

- Shared storage, as Volumes
- Networking, as a unique cluster IP address
- Information about how to run each container, such as the container image version or specific ports to use

A Pod models an application-specific "logical host" and can contain different application containers which are relatively tightly coupled. For example, a Pod might include both the container with your Node.js app as well as a different container that feeds the data to be published by the Node.js webserver. The containers in a Pod share an IP Address and port space, are always co-located and co-scheduled, and run in a shared context on the same Node.

Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly). Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion. In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster.

Summary:

- Pods
- Nodes
- Kubectl main commands

A Pod is a group of one or more application containers (such as Docker) and includes shared storage (volumes), IP address and information about how to run them.

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 1 - Create a Kubernetes cluster

Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to deploy a local development Kubernetes cluster using minikube

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

The online terminal is a pre-configured Linux environment that can be used as a regular

Click To Connect

Launch Interactive Environment

Private

Welcome!

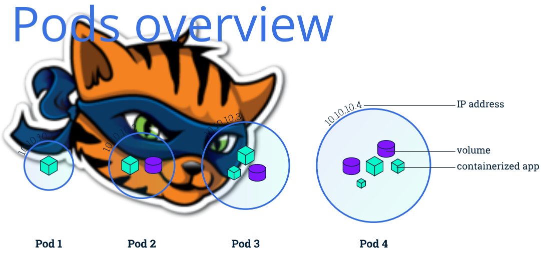
Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

Pods overview



Nodes

A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the control plane. A Node can have multiple pods, and the Kubernetes control plane automatically handles scheduling the pods across the Nodes in the cluster. The control plane's automatic scheduling takes into account the available resources on each Node.

Containers should only be scheduled together in a single Pod if they are tightly coupled and need to share resources such as disk.

Every Kubernetes Node runs at least:

- Kubelet, a process responsible for communication between the Kubernetes control plane and the Node; it manages the Pods and the containers running on a machine.
- A container runtime (like Docker) responsible for pulling the container image from a registry, unpacking the container, and running the application.

Click To Connect

Launch Interactive Environment

Private

Welcome!

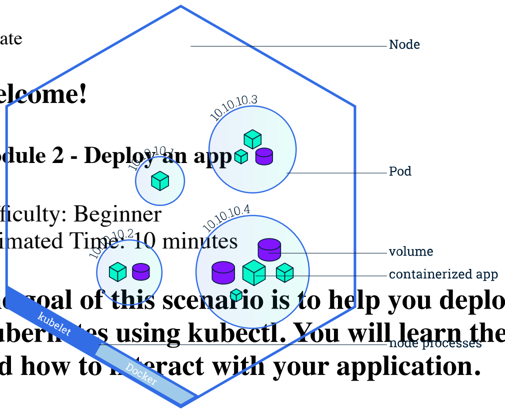
Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy yo
Kubernetes using kubectl. You will learn the bas
and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

Node overview



Click To Connect

Launch Interactive Environment

Private

Welcome!

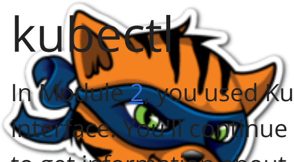
Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

Troubleshooting with kubectl



In Module 2 you used Kubectl command-line interface. You'll continue to use it in Module 3 to get information about deployed applications and their environments. The most common operations can be done with the following kubectl commands:

- **kubectl get** - list resources
- **kubectl describe** - show detailed information about a resource
- **kubectl logs** - print the logs from a container in a pod
- **kubectl exec** - execute a command on a container in a pod

You can use these commands to see when applications were deployed, what their current statuses are, where they are running and what their configurations are.

Now that we know more about our cluster components and the command line, let's explore our application.

A node is a worker machine in Kubernetes and may be a VM or physical machine, depending on the cluster. Multiple Pods can run on one Node.

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

Start Interactive Tutorial ›

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

2.3.2 - Interactive Tutorial - Exploring Your App

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 2 - Deploy an app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to help you deploy your first app on Kubernetes using kubectl. You will learn the basics about kubectl cli and how to interact with your application.

The online terminal is a pre-configured Linux environment that can be used as a regular

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

< Return to Module 2 Home
Continue to Module 4 >

2.4 - Expose Your App Pu

2. ng a Service to Expose Your App

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Objectives

- Learn about a Service in Kubernetes
- Understand how labels and LabelSelector objects relate to a Service
- Expose an application outside a Kubernetes cluster using a Service

Overview of Kubernetes Services

Kubernetes Pods are mortal. Pods have a lifecycle. When a worker node dies, the Pods running on the Node are also lost. A ReplicaSet might then dynamically drive the cluster back to the desired state via the creation of new Pods to keep your application running. As another example, consider an image-processing backend with 3 replicas. Those replicas are exchangeable; the front-end system should not care about backend replicas or even if a Pod is lost and recreated. That said, each Pod in a Kubernetes cluster has a unique IP address, even Pods on the same Node, so there needs to be a way of automatically reconciling changes among Pods so that your applications continue to function.

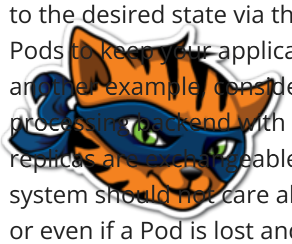
A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML (preferred) or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a *LabelSelector* (see below for why you might want a Service without including `selector` in the spec).

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your

Summary

- Exposing Pods to external traffic
- Load balancing traffic across multiple Pods
- Using labels

A Kubernetes Service is an abstraction layer which defines a logical set of Pods and enables external traffic exposure, load balancing and service discovery for those Pods.



Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoo applications using the kubectl ge , logs

The online terminal is a pre-configured Linux environment th r console (you can type commands). Clicking on the blocks o

applications to receive traffic. Services can be exposed in different ways by specifying a `type` in the ServiceSpec:

- *ClusterIP* (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
- *NodePort* - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using `<NodeIP>:<NodePort>`. Superset of ClusterIP.
- *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
- *ExternalName* - Maps the Service to the contents of the `externalName` field (e.g. `foo.bar.example.com`), by returning a `CNAME` record with its value. No proxying of any kind is set up. This type requires v1.7 or higher of `kube-dns`, or CoreDNS version 0.0.8 or higher.

More information about the different types of Services can be found in the Using Source IP tutorial. Also see Connecting Applications with Services.

Additionally, note that there are some use cases with Services that involve not defining a `selector` in the spec. A Service created without `selector` will also not create the corresponding Endpoints object. This allows users to manually map a Service to specific endpoints. Another possibility why there may be no selector is you are strictly using `type: ExternalName`.

Services and Labels

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

A Service routes traffic across a set of Pods. Services are the abstraction that allows pods to die and replicate in Kubernetes without impacting your application. Discovery and routing among dependent Pods (such as the frontend and backend components in an application) are handled by Kubernetes Services.

Services match a set of Pods using [labels and selectors](), a grouping primitive that allows logical operation on objects in Kubernetes. Labels are key/value pairs attached to objects and can be used in any number of ways:

- Designate objects for development, test, and production
- Embed version tags
- Classify an object using tags

Click To Connect

Launch Interactive Environment

Private

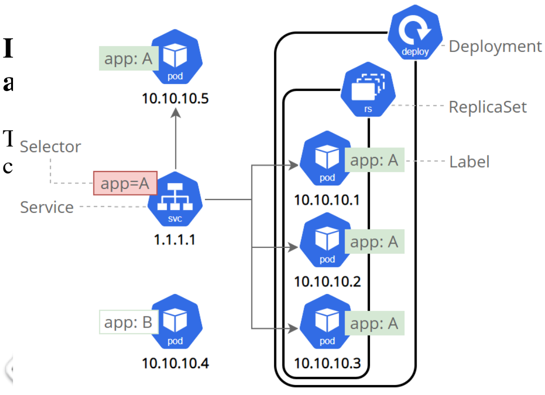
Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

I shoot Kubernetes a ogs and exec commands.

T nt that can be used as a regular c ks of code followed by the



Labels can be attached to objects at creation time or later on. They can be modified at any time. Let's expose our application now using a Service and apply some labels.

Click To Connect

Launch Interactive Environment

Start Interactive Tutorial›

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

2.4.2 - Interactive Tutorial - E[illegible]g Your App

Cli[illegible] To Connect

I[illegible]ve Environment

Private

Welcome!

Module 3 - Explore your app

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to troubleshoot Kubernetes applications using the kubectl get, describe, logs and exec commands.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

< Return to Module 3 Home

Continue to Module 5 >

2.5 - Scale Your App

2. ning Multiple In of Your App

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kul s outside the cluster using the kubectl expose comr learn how to view and apply labels to objects wit command.

Objectives

- Scale an app using kubectl.

Scaling an application

In the previous modules we created a [Deployment](), and then exposed it publicly via a [Service](). The Deployment created only one Pod for running our application. When traffic increases, we will need to scale the application to keep up with user demand.

Scaling is accomplished by changing the number of replicas in a Deployment

Summary:

- Scaling a Deployment

You can create from the start a Deployment with multiple instances using the --replicas parameter for the kubectl create deployment command

Click To Connect

Launch Interactive Environment

Private

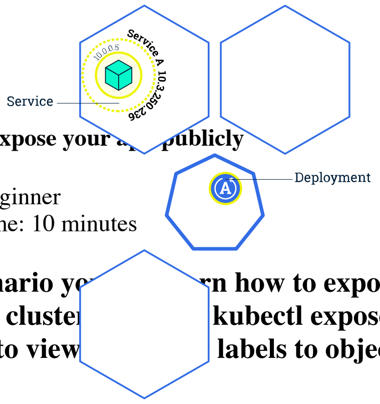
Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

Scaling overview

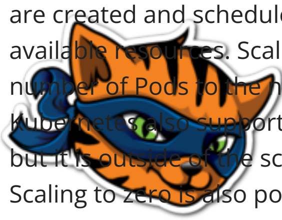


Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling will increase the number of Pods to the new desired state. Kubernetes also supports autoscaling of Pods, but it is outside of the scope of this tutorial. Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment.

Scaling is accomplished by changing the number of replicas in a Deployment.

Running multiple instances of an application will require a way to distribute the traffic to all of them. Services have an integrated load-balancer that will distribute network traffic to all Pods of an exposed Deployment. Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods.

Once you have multiple instances of an Application running, you would be able to do Rolling updates without downtime. We'll cover that in the next module. Now, let's go to the online terminal and scale our application.



Click To Connect

Launch Interactive Environment

Private

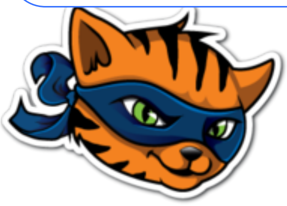
Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

Start Interactive Tutorial ›



Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

2.5.2 - Interactive Tutorial - S[illegible] Your App

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

Click To Connect

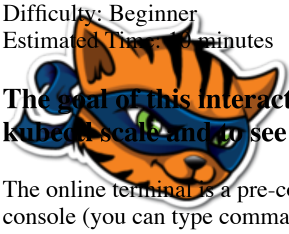
Launch Interactive Environment

Private

Welcome!

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes



The goal of this interactive scenario is to scale a deployment with kubectl scale and to see the load balancing in action

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

< Return to Module 4 Home

Continue to Module 6 >

C

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

In this scenario you will learn how to expose Kubernetes applications outside the cluster using the kubectl expose command. You will also learn how to view and apply labels to objects with the kubectl label command.

2.6 - Update Your App

2. orming a Ko pdate

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 4 - Expose your app publicly

Difficulty: Beginner
Estimated Time: 10 minutes

[illegible]
[illegible]
learn how to view and apply labels to objects wit
command.

The online terminal is a pre-configured Linux environment th
console (you can type commands). Clicking on the `blocks o`

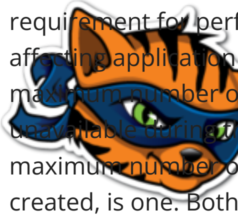
Objectives

- Perform a rolling update using kubectl.

Updating an application

Users expect applications to be available all the time and developers are expected to deploy new versions of them several times a day. In Kubernetes this is done with rolling updates. **Rolling updates** allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones. The new Pods will be scheduled on Nodes with available resources.

In the previous module we scaled our application to run multiple instances. This is a requirement for performing updates without affecting application availability. By default, the maximum number of Pods that can be unavailable during the update and the maximum number of new Pods that can be created, is one. Both options can be configured to either numbers or percentages (of Pods). In Kubernetes, updates are versioned and any Deployment update can be reverted to a previous (stable) version.



Summary:

- Updating an app

Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.

Click To Connect

Launch Interactive Environment

Private

Welcome!

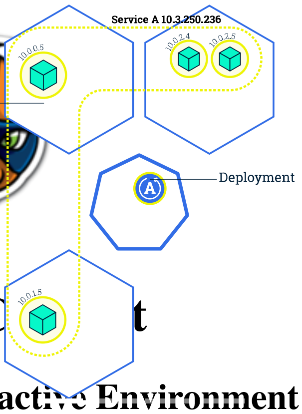
Rolling updates overview

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to scale a deployment with kubectl scale and to see the load balancing in action

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the



Click To C t

Launch Interactive Environment

Private

Welcome!

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to scale a
kubectl scale and to see the load balancing in act

The online terminal is a pre-configured Linux environment th r
console (you can type commands). Clicking on the `blocks o`

Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

If a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update.

Rolling updates allow the following actions:

- Promote an application from one environment to another (via container image updates)
- Rollback to previous versions
- Continuous Integration and Continuous Delivery of applications with zero downtime

In the following interactive tutorial, we'll update our application to a new version, and also perform a rollback.

Start Interactive Tutorial ›

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to scale a deployment with kubectl scale and to see the load balancing in action

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

2.6.2 - Interactive Tutorial - Updating Your App

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to scale a deployment with kubectl scale and to see the load balancing in action

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

< Return to Module 5 Return to Kubernetes Basics

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 5 - Scale up your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this interactive scenario is to scale a deployment with kubectl scale and to see the load balancing in action

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

3 - Configuration

3.1 - Example: Configuring a Java Microservice

3.1.1 - Externalizing config using MicroProfile, ConfigMaps and Secrets

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

In this tutorial you will learn how and why to externalize your microservice's configuration. Specifically, you will learn how to use Kubernetes ConfigMaps and Secrets to set environment variables and then consume them using MicroProfile Config.

Before you begin

Creating Kubernetes ConfigMaps & Secrets

There are several ways to set environment variables for a Docker container in Kubernetes, including: Dockerfile, kubernetes.yml, Kubernetes ConfigMaps, and Kubernetes Secrets. In the tutorial, you will learn how to use the latter two for setting your environment variables whose values will be injected into your microservices. One of the benefits for using ConfigMaps and Secrets is that they can be re-used across multiple containers, including being assigned to different environment variables for the different containers.

ConfigMaps are API Objects that store non-confidential key-value pairs. In the Interactive Tutorial you will learn how to use a ConfigMap to store the application's name. For more information regarding ConfigMaps, you can find the documentation [here](#).

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Although Secrets are also used to store key-value pairs, they differ from ConfigMaps in that they're intended for confidential/sensitive information and are stored using Base64 encoding. This makes secrets the appropriate choice for storing such things as credentials, keys, and tokens, the former of which you'll do in the Interactive Tutorial. For more information on Secrets, you can find the documentation [here](#).

Externalizing Config from Code

Externalized application configuration is useful because configuration usually changes depending on your environment. In order to accomplish this, we'll use Java's Contexts and Dependency Injection (CDI) and MicroProfile

Config. MicroProfile Config is a feature of MicroProfile, a set of open Ja a technologies for developing and deploying cloud-native

CD dependency injection capability en to be assembled from collaborating, loosely-co . MicroProfile Config provides apps and microservices a standard way to obtain config properties from various sources, including the application, runtime, and environment. Based on the source's defined priority, the properties are automatically combined into a single set of properties that the application can access via an API. Together, CDI & MicroProfile will be used in the Interactive Tutorial to retrieve the externally provided properties from the Kubernetes ConfigMaps and Secrets and get injected into your application code.

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

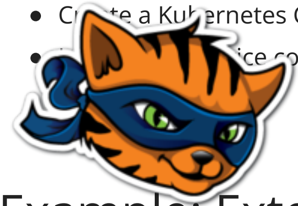
The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Many open source frameworks and runtimes implement and support MicroProfile Config. Throughout the interactive tutorial, you'll be using Open Liberty, a flexible open-source Java runtime for building and running cloud-native apps and microservices. However, any MicroProfile compatible runtime could be used instead.

Objectives

- C te a Ku ernetes ConfigMap and Secret
- ice configuration using MicroProfile



Example: Externalizing config using MicroProfile, ConfigMaps and Secrets

[Start Interactive Tutorial]

Click To Connect

Launch Interactive Environment

Private

Welcome!

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

3.1.2 - Interactive Tutorial - C ring a Java M ice

Click To Connect

Lau ctive Environment

Pr

Welco

Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

Click To Connect

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

Launch Interactive Environment

Private

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Welcome!

Configuring a Kubernetes Microservice

Difficulty: beginner
Estimated Time: 15 minutes

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets

Click To Connect

Launch Interactive Environment

Private

Welcome!

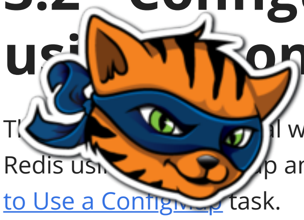
Module 6 - Update your app

Difficulty: Beginner
Estimated Time: 10 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

3.2 - Configuring Redis usi onfigMap



T l world example of how to configure Redis usi p and builds upon the Configure a Pod to Use a ConfigMap task.

Objectives

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

Before you begin

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

-
- es

To check nter `kubectl version`.

- The example shown on this page works with `kubectl` 1.14 and above.
- Understand Configure a Pod to Use a ConfigMap.

Real World Example: Configuring Redis using a ConfigMap

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

First create a ConfigMap with an empty configuration block:

Click To Connect

Launch Interactive Environment

Private

Welcome!

Updating your app

Configuring a Kubernetes Microservice

Difficulty: Beginner

Difficulty: beginner

Estimated Time: 10 minutes

Estimated Time: 15 minutes

The goal of this scenario is to update a deployed application with kubectl set image and to rollback with the rollout undo command.

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets.

The online terminal is a pre-configured Linux environment that can be used as a regular console (you can type commands). Clicking on the `blocks of code` followed by the

Click To Connect

Launch Interactive Environment

Private

Welcome!

Configuring a Kubernetes Microservice

Difficulty: beginner
Estimated Time: 15 minutes

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets.

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

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

```
kubectl apply -f example-redis-config.yaml
kubectl apply -f https://raw.githubusercontent.com/ku
```

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

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

Th exposing the data in `data.redis-config` `ple-redis-config` ConfigMap above as `/redis-mas` `edis.conf` inside the Pod.

Click To Connect

Launch Interactive Environment

Configuring a Kubernetes Microservice

Difficulty: beginner
Estimated Time: 15 minutes

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets

Click To Connect

Launch Interactive Environment

Private

Welcome!

Configuring a Kubernetes Microservice

Difficulty: beginner
Estimated Time: 15 minutes

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets

pods/config/redis-pod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: redis:5.0.4
    command:
      - redis-server
      - "/redis-master/redis.conf"
    env:
    - name: MASTER
      value: "true"
    ports:
    - containerPort: 6379
    resources:
      limits:
        cpu: "0.1"
    volumeMounts:
    - mountPath: /redis-master-data
      name: data
    - mountPath: /redis-master
      name: config
  volumes:
    - name: data
      emptyDir: {}
    - name: config
      configMap:
        name: example-redis-config
        items:
        - key: redis-config
          path: redis.conf
```

loy two
e their
bernetes

Examine the created objects:

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

You should see the following output:

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

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

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

Launch Interactive Environment

Private

Configuring a Kubernetes Microservice

Difficulty: beginner
Estimated Time: 15 minutes

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets

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

Yo redis-config key:

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

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

Welcome!

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

Configuring a Kubernetes Microservice

Difficulty: beginner

```
kubectl exec -it redis -- redis-cli
```

The goal of this interactive scenario is to deploy two Java microservices to Kubernetes and change their configuration using MicroProfile Config, Kubernetes ConfigMaps and Secrets

Check `maxmemory`:

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

It should show the default value of 0:

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

Similarly, check `maxmemory-policy`:

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

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

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

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

pods/config/example-redis-config.yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-redis-config
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
```

Apply the updated ConfigMap:

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

Confirm that the ConfigMap was updated:

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

You should see the configuration values we just added:

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

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

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

```
kubectl exec -it redis -- redis-cli
```

Check `maxmemory`:

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

It remains at the default value of 0:

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

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

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

Returns:

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

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

```
kubectl delete pod redis
kubectl apply -f https://raw.githubusercontent.com/ku
```

Now re-check the configuration values one last time:

```
kubectl exec -it redis -- redis-cli
```

Check `maxmemory`:

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

It should now return the updated value of 2097152:

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

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

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

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

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

Clean up your work by deleting the created resources:

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

What's next

- Learn more about ConfigMaps.

4 - Security

4.1 - Apply Pod Security Standards at the Cluster Level

> Note
>
> This tutorial applies only for new clusters.

Pod Security admission (PSA) is enabled by default in v1.23 and later, as it has graduated to beta. Pod Security is an admission controller that carries out checks against the Kubernetes Pod Security Standards when new pods are created. 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 v1.26, 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:v1.26.0) 🖼
   ✓ 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-clust

   Thanks for using kind! 😊
   ```

2. Set the kubectl context to the new cluster:

   ```
   kubectl cluster-info --context kind-psa-wo-clust
   ```

   The output is similar to this:

   ```
   Kubernetes control plane is running at https://1

   CoreDNS is running at https://127.0.0.1:61350/ap

   To further debug and diagnose cluster problems,
   ```

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
pod-security.kubernetes.io/enforce=privileg
```

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
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-sy
Warning: etcd-psa-wo-cluster-pss-control-pla
Warning: kindnet-vzj42: non-default capabili
Warning: kube-proxy-m6hwf: host namespaces,
namespace/kube-system labeled
namespace/local-path-storage labeled
```

3. Restricted

```
kubectl label --dry-run=server --overwrite
pod-security.kubernetes.io/enforce=restrict
```

The output is similar to:

```
namespace/default labeled
namespace/kube-node-lease labeled
namespace/kube-public labeled
Warning: existing pods in namespace "kube-sy
Warning: coredns-7bb9c7b568-hsptc (and 1 oth
Warning: etcd-psa-wo-cluster-pss-control-pla
Warning: kindnet-vzj42: non-default capabili
Warning: kube-proxy-m6hwf: host namespaces,
namespace/kube-system labeled
Warning: existing pods in namespace "local-p
Warning: local-path-provisioner-d6d9f7ffc-lw
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.k8
    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/co
        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
    # default false
    selinuxRelabel: false
    # optional: set propagation mode (None, Host
    # see https://kubernetes.io/docs/concepts/st
    # 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
```

The output is similar to this:

```
Creating cluster "psa-with-cluster-pss" ...
 ✓ Ensuring node image (kindest/node:v1.26.0) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
Set kubectl context to "kind-psa-with-cluster-ps
You can now use your cluster with:

kubectl cluster-info --context kind-psa-with-clu

Have a question, bug, or feature request? Let us
```

6. Point kubectl to the cluster:

   ```
   kubectl cluster-info --context kind-psa-with-clu
   ```

   The output is similar to this:

   ```
   Kubernetes control plane is running at https://1
   CoreDNS is running at https://127.0.0.1:63855/ap

   To further debug and diagnose cluster problems,
   ```

7. Create a Pod in the default namespace:

   ```
   kubectl apply -f https://k8s.io/examples/securit
   ```

   The pod is started normally, but the output includes a warning:

   ```
   Warning: would violate PodSecurity "restricted:l
   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

4.2 - Apply Pod Security Standards at the Namespace Level

> **Note**
>
> This tutorial applies only for new clusters.

Pod Security admission (PSA) is enabled by default in v1.23 and later, as it graduated to beta. Pod Security Admission is an admission controller that applies Pod Security Standards when pods are created. 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:v1.26.0) 🖼
    ✓ 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://
   ```

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://1
   CoreDNS is running at https://127.0.0.1:50996/ap

   To further debug and diagnose cluster problems,
   ```

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=lates
   ```

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=lat
  pod-security.kubernetes.io/warn=restricted \
  pod-security.kubernetes.io/warn-version=latest
  pod-security.kubernetes.io/audit=restricted \
  pod-security.kubernetes.io/audit-version=lates
```

Verify the Pod Security Standard enforcement

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

   ```
   kubectl apply -n example -f https://k8s.io/examp
   ```

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

   ```
   Warning: would violate PodSecurity "restricted:l
   pod/nginx created
   ```

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

   ```
   kubectl apply -n default -f https://k8s.io/examp
   ```

   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

4.3 - Restrict a Container's Access to Resources with AppArmor

FEATURE STATE: `Kubernetes v1.4 [beta]`

AppArmor is a Linux kernel security module that supplements the standard Linux user and group based permissions to confine programs to a limited set of resources. AppArmor can be configured for any application to reduce its potential attack surface and provide greater in-depth defense. It is configured through profiles tuned to allow the access needed by a specific program or container, such as Linux capabilities, network access, file permissions, etc. Each profile can be run in either *enforcing* mode, which blocks access to disallowed resources, or *complain* mode, which only reports violations.

AppArmor can help you to run a more secure deployment by restricting what containers are allowed to do, and/or provide better auditing through system logs. However, it is important to keep in mind that AppArmor is not a silver bullet and can only do so much to protect against exploits in your application code. It is important to provide good, restrictive profiles, and harden your applications and cluster from other angles as well.

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

Make sure:

1. Kubernetes version is at least v1.4 -- Kubernetes support for AppArmor was added in v1.4. Kubernetes components older than v1.4 are not aware of the new AppArmor annotations, and will **silently ignore** any AppArmor settings that are provided. To ensure that your Pods are receiving the expected protections, it is important to verify the Kubelet version of your nodes:

```
kubectl get nodes -o=jsonpath=$'{range .items[*]
```

```
gke-test-default-pool-239f5d02-gyn2: v1.4.0
gke-test-default-pool-239f5d02-x1kf: v1.4.0
gke-test-default-pool-239f5d02-xwux: v1.4.0
```

2. 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
   ```

   If the Kubelet contains AppArmor support (>= v1.4), it will refuse to run a Pod with AppArmor options if the kernel module is not enabled.

> **Note:** Ubuntu carries many AppArmor patches that have not been merged into the upstream Linux kernel, including patches that add additional hooks and features. Kubernetes has only been tested with the upstream version, and does not promise support for other features.

3. Container runtime supports AppArmor -- Currently all common Kubernetes-supported container runtimes should support AppArmor, like Docker, CRI-O or containerd. Please refer to the corresponding runtime documentation and verify that the cluster fulfills the requirements to use AppArmor.

4. 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 is not already loaded in the kernel, the Kubelet (>= v1.4) 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 ca
   ```

```
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.

As long as the Kubelet version includes AppArmor support (>= v1.4), the Kubelet will reject a Pod with AppArmor options if any of the prerequisites are not met. You can also verify AppArmor support on nodes by checking the node ready condition message (though this is likely to be removed in a later release):

```
kubectl get nodes -o=jsonpath='{range .items[*]}{@.me
```

```
gke-test-default-pool-239f5d02-gyn2: kubelet is posti
gke-test-default-pool-239f5d02-x1kf: kubelet is posti
gke-test-default-pool-239f5d02-xwux: kubelet is posti
```

Securing a Pod

> **Note:** AppArmor is currently in beta, so options are specified as annotations. Once support graduates to general availability, the annotations will be replaced with first-class fields (more details in Upgrade path to GA).

AppArmor profiles are specified *per-container*. To specify the AppArmor profile to run a Pod container with, add an annotation to the Pod's metadata:

```
container.apparmor.security.beta.kubernetes.io/<conta
```

Where `<container_name>` is the name of the container to apply the profile to, and `<profile_ref>` specifies the profile to apply. The `profile_ref` can be one of:

- `runtime/default` to apply the runtime's default profile
- `localhost/<profile_name>` to apply the profile loaded on the host with the name `<profile_name>`
- `unconfined` to indicate that no profiles will be loaded

See the API Reference for the full details on the annotation and profile name formats.

Kubernetes AppArmor enforcement works by first checking that all the prerequisites have been met, and then forwarding the profile selection to the container runtime for enforcement. If the prerequisites have not been met, the Pod will be rejected, and will not run.

To verify that the profile was applied, you can look for the AppArmor security option listed in the container created event:

```
kubectl get events | grep Created
```

```
22s        22s         1         hello-apparmor     P
```

You can also verify directly that the container's root process is running with the correct profile by checking its proc attr:

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

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

Example

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

First, we need to load the profile we want to use onto our nodes. This profile denies all file writes:

```
#include <tunables/global>

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

  file,

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

Since we don't know where the Pod will be scheduled, we'll need to load the profile on all our nodes. For this example we'll use SSH to install the profiles, but other approaches are discussed in [Setting up nodes with profiles](#).

```
NODES=(
    # The SSH-accessible domain names of your nodes
    gke-test-default-pool-239f5d02-gyn2.us-central1-a
    gke-test-default-pool-239f5d02-x1kf.us-central1-a
    gke-test-default-pool-239f5d02-xwux.us-central1-a
for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_
#include <tunables/global>

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

  file,

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

Next, we'll run a simple "Hello AppArmor" pod with the deny-write profile:

pods/security/hello-apparmor.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    # Tell Kubernetes to apply the AppArmor profile '
    # Note that this is ignored if the Kubernetes noc
    container.apparmor.security.beta.kubernetes.io/he
spec:
  containers:
  - name: hello
    image: busybox:1.28
    command: [ "sh", "-c", "echo 'Hello AppArmor!' &&
```

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

If we look at the pod events, we can see that the Pod container was created with the AppArmor profile "k8s-apparmor-example-deny-write":

```
kubectl get events | grep hello-apparmor
```

```
14s         14s         1         hello-apparmor    Pod
14s         14s         1         hello-apparmor    Pod
13s         13s         1         hello-apparmor    Pod
13s         13s         1         hello-apparmor    Pod
13s         13s         1         hello-apparmor    Pod
```

We can verify that the container is actually running with that profile by checking its proc attr:

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

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

Finally, we can see what happens if we try to 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 termin
```

To wrap up, let's look at what happens if we 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
  annotations:
    container.apparmor.security.beta.kubernetes.io/he
spec:
  containers:
  - name: hello
    image: busybox:1.28
    command: [ "sh", "-c", "echo 'Hello AppArmor!' &&
EOF
pod/hello-apparmor-2 created
```

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

```
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kuber
Status:        Pending
Reason:        AppArmor
Message:       Pod Cannot enforce AppArmor: profile "
IP:
Controllers:   <none>
Containers:
  hello:
    Container ID:
    Image:     busybox
    Image ID:
    Port:
    Command:
      sh
      -c
      echo 'Hello AppArmor!' && sleep 1h
    State:              Waiting
      Reason:           Blocked
    Ready:              False
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount f
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-dnz7v:
    Type:    Secret (a volume populated by a Secret)
    SecretName:    default-token-dnz7v
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen    LastSeen    Count    From
  ---------    --------    -----    ----
  23s          23s         1        {default-schedule
  23s          23s         1        {kubelet e2e-test
```

Note the pod status is Pending, with a helpful error message: `Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" is not loaded`. An event was also recorded with the same message.

Administration

Setting up nodes with profiles

Kubernetes does not currently provide any native mechanisms for loading AppArmor profiles onto nodes. There are lots of ways to set up the profiles though, such as:

- Through a DaemonSet that runs a Pod on each node to ensure the correct profiles are loaded. An example implementation can be found here.
- At node initialization time, using your node initialization scripts (e.g. Salt, Ansible, etc.) or image.
- By copying the profiles to each node and loading them through SSH, as demonstrated in the Example.

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.

Disabling AppArmor

If you do not want AppArmor to be available on your cluster, it can be disabled by a command-line flag:

```
--feature-gates=AppArmor=false
```

When disabled, any Pod that includes an AppArmor profile will fail validation with a "Forbidden" error.

> **Note:** Even if the Kubernetes feature is disabled, runtimes may still enforce the default profile. The option to disable the AppArmor feature will be removed when AppArmor graduates to general availability (GA).

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.

API Reference

Pod Annotation

Specifying the profile a container will run with:

- **key**: `container.apparmor.security.beta.kubernetes.io/<container_name>` Where `<container_name>` matches the name of a container in the Pod. A separate profile can be specified for each container in the Pod.
- **value**: a profile reference, described below

Profile Reference

- `runtime/default` : Refers to the default runtime profile.
    - Equivalent to not specifying a profile, except it still requires AppArmor to be enabled.
    - In practice, many container runtimes use the same OCI default profile, defined here: https://github.com/containers/common/blob/main/pkg/apparmor/apparmor_linux_template.go
- `localhost/<profile_name>` : Refers to a profile loaded on the node (localhost) by name.
    - The possible profile names are detailed in the core policy reference.
- `unconfined` : This effectively disables AppArmor on the container.

Any other profile reference format is invalid.

What's next

Additional resources:

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

4.4 - 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.

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

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

Run these commands:

```
mkdir ./profiles
curl -L -o profiles/audit.json https://k8s.io/example
curl -L -o profiles/violation.json https://k8s.io/exa
curl -L -o profiles/fine-grained.json https://k8s.io/
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

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

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

```
curl -L -O https://k8s.io/examples/pods/security/secc
```

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 v1.26.

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](enable) 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
6a96207fed4b        kindest/node:v1.18.2    "/usr/loca
```

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 fr
docker exec -it 6a96207fed4b ls /var/lib/kubelet/secc
```

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

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

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

FEATURE STATE: `Kubernetes v1.25 [beta]`

To use seccomp profile defaulting, you must run the kubelet with the `SeccompDefault` feature gate enabled (this is the default). You must also explicitly enable the defaulting behavior for each node where you want to use this with the corresponding `--seccomp-default` command line flag. Both have to be enabled simultaneously to use the feature.

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 1.26 lets you configure the seccomp profile that applies when the spec for a Pod doesn't define a specific seccomp profile. This is a beta feature and the corresponding `SeccompDefault` feature gate is enabled by default. However, you still need to enable this defaulting for each node where you would like to use it.

If you are running a Kubernetes 1.26 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/v1alpha4
featureGates:
  SeccompDefault: true
nodes:
  - role: control-plane
    image: kindest/node:v1.23.0@sha256:49824ab1727c04
    kubeadmConfigPatches:
      - |
        kind: JoinConfiguration
        nodeRegistration:
          kubeletExtraArgs:
            seccomp-default: "true"
  - role: worker
    image: kindest/node:v1.23.0@sha256:49824ab1727c04
    kubeadmConfigPatches:
      - |
        kind: JoinConfiguration
        nodeRegistration:
          kubeletExtraArgs:
            feature-gates: SeccompDefault=true
            seccomp-default: "true"
```

If the cluster is ready, then running a pod:

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

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) | j
```

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": ["SCMP_ARCH_X86_64", "SCMP_ARCH_X8
  "syscalls": [
    {
      "names": ["..."]
    }
  ]
}
```

Create 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` feature gate 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

```
apiVersion: v1
kind: Pod
metadata:
  name: default-pod
  labels:
    app: default-pod
spec:
  securityContext:
    seccompProfile:
      type: RuntimeDefault
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some more syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

Create that Pod:

```
kubectl apply -f https://k8s.io/examples/pods/securit
```

```
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
```

Finally, now that you saw that work OK, clean up:

```
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

```
apiVersion: v1
kind: Pod
metadata:
  name: audit-pod
  labels:
    app: audit-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/audit.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

Note:
The functional support for the already deprecated seccomp annotations `seccomp.security.alpha.kubernetes.io/pod` (for the whole pod) and `container.seccomp.security.alpha.kubernetes.io/[name]` (for a single container) is going to be removed with a future release of Kubernetes. Please always use the native API fields in favor of the annotations.

Since Kubernetes v1.25, kubelets no longer support the annotations, use of the annotations in static pods is no longer supported, and the seccomp annotations are no longer auto-populated when pods with seccomp fields are created. Auto-population of the seccomp fields from the annotations is planned to be removed in a future release.

Create the Pod in the cluster:

```
kubectl apply -f https://k8s.io/examples/pods/securit
```

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 Services that allows access to the endpoint from inside the kind control plane container.

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

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
audit-pod    NodePort   10.111.36.142   <none>
```

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
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`. Open up a new terminal window and `tail` the output for calls from `http-echo`:

```
tail -f /var/log/syslog | grep 'http-echo'
```

You should already see some logs of syscalls made by `http-echo`, and if you `curl` the endpoint in the control plane container you will see more written.

For example:

```
Jul  6 15:37:40 my-machine kernel: [369128.669452] au
Jul  6 15:37:40 my-machine kernel: [369128.669453] au
Jul  6 15:37:40 my-machine kernel: [369128.669455] au
Jul  6 15:37:40 my-machine kernel: [369128.669456] au
Jul  6 15:37:40 my-machine kernel: [369128.669517] au
Jul  6 15:37:40 my-machine kernel: [369128.669519] au
Jul  6 15:38:40 my-machine kernel: [369188.671648] au
Jul  6 15:38:40 my-machine kernel: [369188.671726] au
```

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.

Clean up that Pod and Service before moving to the next section:

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

Create 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: violation-pod
  labels:
    app: violation-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/violation.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

Attempt to create the Pod in the cluster:

```
kubectl apply -f https://k8s.io/examples/pods/securit
```

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
violation-pod   0/1     CrashLoopBackOff   1
```

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.

Clean up that Pod before moving to the next section:

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

Create 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

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: fine-pod
  labels:
    app: fine-pod
spec:
  securityContext:
    seccompProfile:
      type: Localhost
      localhostProfile: profiles/fine-grained.json
  containers:
  - name: test-container
    image: hashicorp/http-echo:0.2.3
    args:
    - "-text=just made some syscalls!"
    securityContext:
      allowPrivilegeEscalation: false
```

Create the Pod in your cluster:

```
kubectl apply -f https://k8s.io/examples/pods/securit
```

```
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 fr
tail -f /var/log/syslog | grep 'http-echo'
```

Next, expose the Pod with a NodePort Service:

```
kubectl expose pod fine-pod --type NodePort --port 56
```

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
fine-pod    NodePort   10.111.36.142   <none>
```

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

```
# Change 6a96207fed4b to the control plane container
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.

Clean up that Pod and Service before moving to the next section:

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

What's next

You can learn more about Linux seccomp:

- A seccomp Overview
- Seccomp Security Profiles for Docker

5 - Stateless Applications

5.1 - 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

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: load-balancer-exampl
  name: hello-world
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: load-balancer-exam
  template:
    metadata:
      labels:
        app.kubernetes.io/name: load-balancer-ex
    spec:
      containers:
      - image: gcr.io/google-samples/node-hello:
        name: hello-world
        ports:
        - containerPort: 8080
```

```
kubectl apply -f https://k8s.io/examples/service
```

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=Loa
```

5. Display information about the Service:

```
kubectl get services my-service
```

The output is similar to:

```
NAME         TYPE           CLUSTER-IP     EXTER
my-service   LoadBalancer   10.3.245.137   104.1
```

> **Note:** The `type=LoadBalancer` service is backed by external cloud providers, which is not covered in this example, please refer to this page 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-bala
Annotations:    <none>
Selector:       app.kubernetes.io/name=load-bala
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
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         NOD
   hello-world-2895499144-1jaz9 ...  10.0.1.6   gke
   hello-world-2895499144-2e5uh ...  10.0.1.8   gke
   hello-world-2895499144-9m4h1 ...  10.0.0.6   gke
   hello-world-2895499144-o4z13 ...  10.0.1.7   gke
   hello-world-2895499144-segjf ...  10.0.2.5   gke
   ```

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 Kubernetes!
   ```

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.

5.2 - 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:

- Killercoda
- 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]

```
# SOURCE: https://cloud.google.com/kubernetes-engine/
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-leader
  labels:
    app: redis
    role: leader
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
        role: leader
        tier: backend
    spec:
      containers:
      - name: leader
        image: "docker.io/redis:6.0.5"
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

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/applica
```

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
redis-leader-fb76b4755-xjr2n  1/1     Running
```

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

```
# SOURCE: https://cloud.google.com/kubernetes-engine/
apiVersion: v1
kind: Service
metadata:
  name: redis-leader
  labels:
    app: redis
    role: leader
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: leader
    tier: backend
```

1. Apply the Redis Service from the following `redis-leader-service.yaml` file:

```
kubectl apply -f https://k8s.io/examples/applica
```

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
kubernetes     ClusterIP   10.0.0.1      <none>
redis-leader   ClusterIP   10.103.78.24  <none>
```

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

```yaml
# SOURCE: https://cloud.google.com/kubernetes-engine/
apiVersion: apps/v1
kind: Deployment
metadata:
  name: redis-follower
  labels:
    app: redis
    role: follower
    tier: backend
spec:
  replicas: 2
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
        role: follower
        tier: backend
    spec:
      containers:
      - name: follower
        image: gcr.io/google_samples/gb-redis-followe
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 6379
```

1. Apply the Redis Deployment from the following `redis-follower-deployment.yaml` file:

```
kubectl apply -f https://k8s.io/examples/applica
```

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
redis-follower-dddfbdcc9-82sfr    1/1     Running
redis-follower-dddfbdcc9-qrt5k    1/1     Running
redis-leader-fb76b4755-xjr2n      1/1     Running
```

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

```
# SOURCE: https://cloud.google.com/kubernetes-engine/
apiVersion: v1
kind: Service
metadata:
  name: redis-follower
  labels:
    app: redis
    role: follower
    tier: backend
spec:
  ports:
    # the port that this service should serve on
  - port: 6379
  selector:
    app: redis
    role: follower
    tier: backend
```

1. Apply the Redis Service from the following `redis-follower-service.yaml` file:

```
kubectl apply -f https://k8s.io/examples/applica
```

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      EXT
kubernetes       ClusterIP   10.96.0.1       <no
redis-follower   ClusterIP   10.110.162.42   <no
redis-leader     ClusterIP   10.103.78.24    <no
```

> **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

```yaml
# SOURCE: https://cloud.google.com/kubernetes-engine/
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
        app: guestbook
        tier: frontend
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google_samples/gb-frontend:v5
        env:
        - name: GET_HOSTS_FROM
          value: "dns"
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        ports:
        - containerPort: 80
```

1. Apply the frontend Deployment from the `frontend-deployment.yaml` file:

```
kubectl apply -f https://k8s.io/examples/applica
```

2. Query the list of Pods to verify that the three frontend replicas are running:

```
kubectl get pods -l app=guestbook -l tier=fronte
```

The response should be similar to this:

```
NAME                       READY   STATUS    RE
frontend-85595f5bf9-5tqhb  1/1     Running   0
frontend-85595f5bf9-qbzwm  1/1     Running   0
frontend-85595f5bf9-zchwc  1/1     Running   0
```

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]()

```yaml
# SOURCE: https://cloud.google.com/kubernetes-engine
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # if your cluster supports it, uncomment the follo
  # an external load-balanced IP for the frontend se
  # type: LoadBalancer
  #type: LoadBalancer
  ports:
    # the port that this service should serve on
  - port: 80
  selector:
    app: guestbook
    tier: frontend
```

1. Apply the frontend Service from the `frontend-service.yaml` file:

```
kubectl apply -f https://k8s.io/examples/applica
```

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      EXT
frontend         ClusterIP   10.97.28.230    <no
kubernetes       ClusterIP   10.96.0.1       <no
redis-follower   ClusterIP   10.110.162.42   <no
redis-leader     ClusterIP   10.103.78.24    <no
```

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      EXTERN
frontend   LoadBalancer   10.51.242.136   109.19
```

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
frontend-85595f5bf9-5df5m        1/1     Running
frontend-85595f5bf9-7zmg5        1/1     Running
frontend-85595f5bf9-cpskg        1/1     Running
frontend-85595f5bf9-l2l54        1/1     Running
frontend-85595f5bf9-l9c8z        1/1     Running
redis-follower-dddfbdcc9-82sfr   1/1     Running
redis-follower-dddfbdcc9-qrt5k   1/1     Running
redis-leader-fb76b4755-xjr2n     1/1     Running
```

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
frontend-85595f5bf9-cpskg        1/1     Running
frontend-85595f5bf9-l9c8z        1/1     Running
redis-follower-dddfbdcc9-82sfr   1/1     Running
redis-follower-dddfbdcc9-qrt5k   1/1     Running
redis-leader-fb76b4755-xjr2n     1/1     Running
```

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 Managing Resources

6 - Stateful Applications

6.1 - 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
- PersistentVolume Provisioning
- StatefulSets
- The kubectl command line tool

> **Note:** This tutorial assumes that your cluster is configured to dynamically provision PersistentVolumes. If your cluster is not configured to do so, you will have to manually provision two 1 GiB volumes prior to starting this tutorial.

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 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

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Download the example above, and save it to a file named `web.yaml`

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

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

In the second terminal, use `kubectl apply` to create the headless Service and StatefulSet defined in `web.yaml`.

```
kubectl apply -f 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   POR
nginx     ClusterIP    None         <none>        80/
```

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

```
kubectl get statefulset web
```

```
NAME      DESIRED   CURRENT   AGE
web       2         1         20s
```

Ordered Pod Creation

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.

```
kubectl get pods -w -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).

> **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 'host
```

```
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 --
```

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

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

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:

```
kubectl get pod -w -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:

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

```
NAME      READY     STATUS              RESTARTS   AG
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 'hostna
```

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

then, run:

```
kubectl run -i --tty --image busybox:1.28 dns-test --
```

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

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

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 IP address.

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.

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
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-420
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-420
```

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

for i in 0 1; do kubectl exec -i -t "web-$i" -- curl
```

```
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](https://github.com/kubernetes/kubernetes/issues/2630)), 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:

```
kubectl get pod -w -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.

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

```
NAME      READY     STATUS              RESTARTS   AG
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
```

```
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. This is accomplished by updating the `replicas` field. You can use either `kubectl scale` or `kubectl patch` to scale a StatefulSet.

Scaling Up

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

```
kubectl get pods -w -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.

```
kubectl get pods -w -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

In one terminal, watch the StatefulSet's Pods:

```
kubectl get pods -w -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.

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

```
NAME      READY     STATUS              RESTARTS   AG
web-0     1/1       Running             0          3h
web-1     1/1       Running             0          3h
web-2     1/1       Running             0          55
web-3     1/1       Running             0          36
web-4     0/1       ContainerCreating   0          18
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 controller deleted one Pod at a time, in reverse order with respect to its ordinal index, and it waited for each to be completely shutdown before deleting the next.

Get the StatefulSet's PersistentVolumeClaims:

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

```
NAME        STATUS    VOLUME
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-420
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-420
www-web-2   Bound     pvc-e1125b27-b508-11e6-932f-420
www-web-3   Bound     pvc-e1176df6-b508-11e6-932f-420
www-web-4   Bound     pvc-e11bb5f8-b508-11e6-932f-420
```

There are still five PersistentVolumeClaims and five PersistentVolumes. When exploring a Pod's stable storage, we 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

In Kubernetes 1.7 and later, 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` and `OnDelete`.

`RollingUpdate` update strategy is the default for StatefulSets.

Rolling Update

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

Patch the `web` StatefulSet to apply the `RollingUpdate` update strategy:

```
kubectl patch statefulset web -p '{"spec":{"updateStr
```

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

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

```
kubectl patch statefulset web --type='json' -p='[{"op
```

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

In another terminal, watch the Pods in the StatefulSet:

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

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 its current 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" --templat
```

```
registry.k8s.io/nginx-slim:0.8
registry.k8s.io/nginx-slim:0.8
registry.k8s.io/nginx-slim:0.8
```

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 stage an update to a StatefulSet by using the `partition` parameter of the `RollingUpdate` update strategy. A staged update will keep all of the Pods in the StatefulSet at the current version while allowing mutations to the StatefulSet's `.spec.template`.

Patch the `web` StatefulSet to add a partition to the `updateStrategy` field:

```
kubectl patch statefulset web -p '{"spec":{"updateStr
```

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

Patch the StatefulSet again to change the container's image:

```
kubectl patch statefulset web --type='json' -p='[{"op
```

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

Delete a Pod in the StatefulSet:

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

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

Wait for the Pod to be Running and Ready.

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

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

Get the Pod's container image:

```
kubectl get pod web-2 --template '{{range $i, $c := .
```

```
registry.k8s.io/nginx-slim:0.8
```

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

Rolling Out a Canary

You can roll out a canary to test a modification by decrementing the `partition` you specified above.

Patch the StatefulSet to decrement the partition:

```
kubectl patch statefulset web -p '{"spec":{"updateStr
```

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

Wait for `web-2` to be Running and Ready.

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

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

Get the Pod's container:

```
kubectl get pod web-2 --template '{{range $i, $c := .
```

```
registry.k8s.io/nginx-slim:0.7
```

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.

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

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 := .
```

```
registry.k8s.io/nginx-slim:0.8
```

`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":{"updateStr
```

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

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

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

The output is similar to:

```
NAME      READY     STATUS              RESTARTS   AG
web-0     1/1       Running             0          3m
web-1     0/1       ContainerCreating   0          11
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" --templat
```

```
registry.k8s.io/nginx-slim:0.7
registry.k8s.io/nginx-slim:0.7
registry.k8s.io/nginx-slim:0.7
```

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

On Delete

The `OnDelete` update strategy implements the legacy (1.6 and prior) behavior, When you select this update strategy, the StatefulSet controller will not automatically update Pods when a modification is made to the StatefulSet's `.spec.template` field. This strategy can be selected by setting the `.spec.template.updateStrategy.type` to `OnDelete`.

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.

Non-Cascading Delete

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

```
kubectl get pods -w -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.

```
kubectl get pods -w -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 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.

```
kubectl get pods -w -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.

Let's 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
```

```
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.

```
kubectl get pods -w -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.

```
kubectl get pods -w -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 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
```

```
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. To address this, in Kubernetes 1.7, we introduced `.spec.podManagementPolicy` to the StatefulSet API Object.

OrderedReady Pod Management

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

Parallel Pod Management

`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. This option only affects the behavior for scaling operations. Updates are not affected.

application/web/web-parallel.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  serviceName: "nginx"
  podManagementPolicy: "Parallel"
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: registry.k8s.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

Download the example above, and save it to a file named `web-parallel.yaml`

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.

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

In another terminal, create the StatefulSet and Service in the manifest:

```
kubectl apply -f web-parallel.yaml
```

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

Examine the output of the `kubectl get` command that you executed in the first terminal.

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

```
NAME      READY     STATUS    RESTARTS   AGE
web-0     0/1       Pending   0          0s
web-0     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-1     0/1       Pending   0         0s
web-0     0/1       ContainerCreating   0         0s
web-1     0/1       ContainerCreating   0         0s
web-0     1/1       Running   0         10s
web-1     1/1       Running   0         10s
```

The StatefulSet controller launched both `web-0` and `web-1` at the same time.

Keep the second terminal open, and, in another terminal window scale the StatefulSet:

```
kubectl scale statefulset/web --replicas=4
```

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

Examine the output of the terminal where the `kubectl get` command is running.

```
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     1/1       Running   0         10s
web-3     1/1       Running   0         26s
```

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

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.

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

```
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
www-web-0   Bound    pvc-2bf00408-d366-4a12-bad0-1869
www-web-1   Bound    pvc-ba3bfe9c-413e-4b95-a2c0-3ea8
www-web-2   Bound    pvc-cba6cfa6-3a47-486b-a138-db59
www-web-3   Bound    pvc-0c04d7f0-787a-4977-8da3-d9d3
www-web-4   Bound    pvc-b2c73489-e70b-4a4e-9ec1-9eab
```

```
kubectl get pv
```

```
NAME                                       CAPACITY
pvc-0c04d7f0-787a-4977-8da3-d9d3a6d8d752   1Gi
pvc-2bf00408-d366-4a12-bad0-1869c65d0bee   1Gi
pvc-b2c73489-e70b-4a4e-9ec1-9eab439aa43e   1Gi
pvc-ba3bfe9c-413e-4b95-a2c0-3ea8a54dbab4   1Gi
pvc-cba6cfa6-3a47-486b-a138-db5930207eaf   1Gi
```

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

```
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.

6.2 - 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:

- Killercoda
- Play with Kubernetes

To check the version, enter `kubectl version`. The example shown on this page works with `kubectl` 1.14 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

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  ports:
    - port: 3306
  selector:
    app: wordpress
    tier: mysql
  clusterIP: None
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
```

```
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress-mysql
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: mysql
    spec:
      containers:
      - image: mysql:5.6
        name: mysql
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: mysql-pass
              key: password
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

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

```yaml
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: LoadBalancer
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: wp-pv-claim
  labels:
    app: wordpress
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 20Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
    spec:
      containers:
      - image: wordpress:4.8-apache
        name: wordpress
        env:
```

```
          - name: WORDPRESS_DB_HOST
            value: wordpress-mysql
          - name: WORDPRESS_DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: mysql-pass
                key: password
          ports:
          - containerPort: 80
            name: wordpress
          volumeMounts:
          - name: wordpress-persistent-storage
            mountPath: /var/www/html
        volumes:
        - name: wordpress-persistent-storage
          persistentVolumeClaim:
            claimName: wp-pv-claim
```

1. Download the MySQL deployment configuration file.

```
curl -LO https://k8s.io/examples/application/wor
```

2. Download the WordPress configuration file.

```
curl -LO https://k8s.io/examples/application/wor
```

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
   mysql-pass-c57bb4t7mf   Opaque
   ```

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
   mysql-pv-claim   Bound     pvc-8cbd7b2e-4044-11e
   wp-pv-claim      Bound     pvc-8cd0df54-4044-11e
   ```

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     STA
   wordpress-mysql-1894417608-x5dzt    1/1       Run
   ```

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   EXTERNA
wordpress   LoadBalancer   10.0.0.89    <pendin
```

> **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:

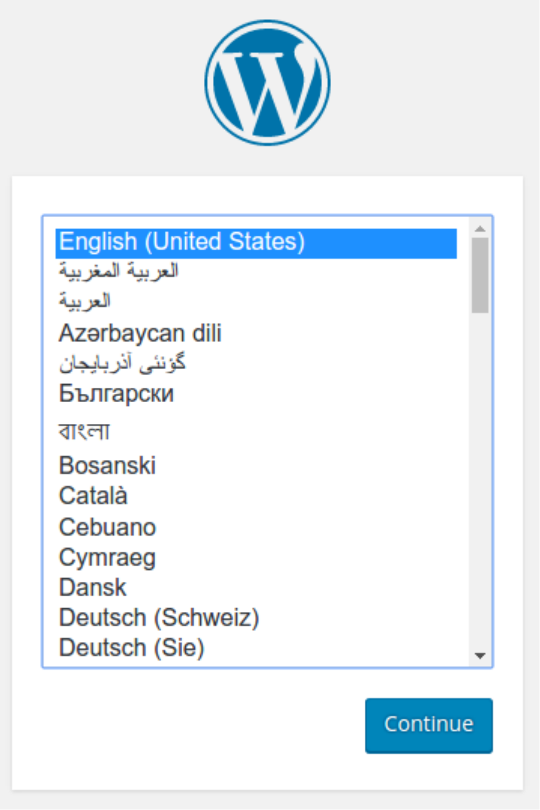
```
minikube service wordpress --url
```

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.



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

6.3 - 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:

- Killercoda

- 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

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: cassandra
  name: cassandra
spec:
  clusterIP: None
  ports:
  - port: 9042
  selector:
    app: cassandra
```

Create a Service to track all Cassandra StatefulSet members from the `cassandra-service.yaml` file:

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

Validating (optional)

Get the Cassandra Service.

```
kubectl get svc cassandra
```

The response is

```
NAME         TYPE         CLUSTER-IP    EXTERNAL-IP    PO
cassandra    ClusterIP    None          <none>         90
```

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

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: cassandra
  labels:
    app: cassandra
spec:
  serviceName: cassandra
  replicas: 3
  selector:
    matchLabels:
      app: cassandra
  template:
    metadata:
      labels:
        app: cassandra
    spec:
      terminationGracePeriodSeconds: 1800
```

```
      containers:
      - name: cassandra
        image: gcr.io/google-samples/cassandra:v13
        imagePullPolicy: Always
        ports:
        - containerPort: 7000
          name: intra-node
        - containerPort: 7001
          name: tls-intra-node
        - containerPort: 7199
          name: jmx
        - containerPort: 9042
          name: cql
        resources:
          limits:
            cpu: "500m"
            memory: 1Gi
          requests:
            cpu: "500m"
            memory: 1Gi
        securityContext:
          capabilities:
            add:
              - IPC_LOCK
        lifecycle:
          preStop:
            exec:
              command:
              - /bin/sh
              - -c
              - nodetool drain
        env:
          - name: MAX_HEAP_SIZE
            value: 512M
          - name: HEAP_NEWSIZE
            value: 100M
          - name: CASSANDRA_SEEDS
            value: "cassandra-0.cassandra.default.sv
          - name: CASSANDRA_CLUSTER_NAME
            value: "K8Demo"
          - name: CASSANDRA_DC
            value: "DC1-K8Demo"
          - name: CASSANDRA_RACK
            value: "Rack1-K8Demo"
          - name: POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP
        readinessProbe:
          exec:
            command:
            - /bin/bash
            - -c
            - /ready-probe.sh
          initialDelaySeconds: 15
          timeoutSeconds: 5
        # These volume mounts are persistent. They a
        # but not exactly because the names need to r
```

```
        # the stateful pod volumes.
        volumeMounts:
        - name: cassandra-data
          mountPath: /cassandra_data
  # These are converted to volume claims by the cont
  # and mounted at the paths mentioned above.
  # do not use these in production until ssd GCEPers
  volumeClaimTemplates:
  - metadata:
      name: cassandra-data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: fast
      resources:
        requests:
          storage: 1Gi
---
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: fast
provisioner: k8s.io/minikube-hostpath
parameters:
  type: pd-ssd
```

Create the Cassandra StatefulSet from the `cassandra-statefulset.yaml` file:

```
# Use this if you are able to apply cassandra-statefu
kubectl apply -f https://k8s.io/examples/application/
```

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-stateful
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              REST
cassandra-0   1/1       Running             0
cassandra-1   0/1       ContainerCreating   0
```

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 (ef
UN  172.17.0.5  83.57 KiB  32           74.0%
UN  172.17.0.4  101.04 KiB  32           58.8%
UN  172.17.0.6  84.74 KiB  32           67.1%
```

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
   # and an empty file will abort the edit. If an e
   # reopened with the relevant failures.
   #
   apiVersion: apps/v1
   kind: StatefulSet
   metadata:
     creationTimestamp: 2016-08-13T18:40:58Z
     generation: 1
     labels:
     app: cassandra
     name: cassandra
     namespace: default
     resourceVersion: "323"
     uid: 7a219483-6185-11e6-a910-42010a8a0fc0
   spec:
     replicas: 3
   ```

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=
  && kubectl delete statefulset -l app=cassandra
  && echo "Sleeping ${grace} seconds" 1>&2 \
  && sleep $grace \
  && kubectl delete persistentvolumeclaim -l app
```

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

6.4 - 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
- PersistentVolume Provisioning
- 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

```yaml
apiVersion: v1
kind: Service
metadata:
  name: zk-hs
  labels:
    app: zk
spec:
  ports:
  - port: 2888
    name: server
  - port: 3888
    name: leader-election
```

```
  clusterIP: None
  selector:
    app: zk
---
apiVersion: v1
kind: Service
metadata:
  name: zk-cs
  labels:
    app: zk
spec:
  ports:
  - port: 2181
    name: client
  selector:
    app: zk
---
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
  name: zk-pdb
spec:
  selector:
    matchLabels:
      app: zk
  maxUnavailable: 1
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: zk
spec:
  selector:
    matchLabels:
      app: zk
  serviceName: zk-hs
  replicas: 3
  updateStrategy:
    type: RollingUpdate
  podManagementPolicy: OrderedReady
  template:
    metadata:
      labels:
        app: zk
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecu
            - labelSelector:
                matchExpressions:
                  - key: "app"
                    operator: In
                    values:
                    - zk
              topologyKey: "kubernetes.io/hostname"
      containers:
      - name: kubernetes-zookeeper
        imagePullPolicy: Always
```

```
        image: "registry.k8s.io/kubernetes-zookeeper
        resources:
          requests:
            memory: "1Gi"
            cpu: "0.5"
        ports:
        - containerPort: 2181
          name: client
        - containerPort: 2888
          name: server
        - containerPort: 3888
          name: leader-election
        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"
        readinessProbe:
          exec:
            command:
            - sh
            - -c
            - "zookeeper-ready 2181"
          initialDelaySeconds: 10
          timeoutSeconds: 5
        livenessProbe:
          exec:
            command:
            - sh
            - -c
            - "zookeeper-ready 2181"
          initialDelaySeconds: 10
          timeoutSeconds: 5
        volumeMounts:
        - name: datadir
          mountPath: /var/lib/zookeeper
      securityContext:
        runAsUser: 1000
        fsGroup: 1000
  volumeClaimTemplates:
  - metadata:
      name: datadir
```

```
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 10Gi
```

Open a terminal, and use the [`kubectl apply`](#) command to create the manifest.

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

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; d
```

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-
```

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;
```

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:2888:38
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:38
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:38
```

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:2888:38
server.2=zk-1.zk-hs.default.svc.cluster.local:2888:38
server.3=zk-2.zk-hs.default.svc.cluster.local:2888:38
```

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/
```

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: any
    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
datadir-zk-0   Bound     pvc-bed742cd-bcb1-11e6-994f-
datadir-zk-1   Bound     pvc-bedd27d2-bcb1-11e6-994f-
datadir-zk-2   Bound     pvc-bee0817e-bcb1-11e6-994f-
```

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.proper
```

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=${zookeeper.root.logger}
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppend
log4j.appender.CONSOLE.Threshold=${zookeeper.console.
log4j.appender.CONSOLE.layout=org.apache.log4j.Patter
log4j.appender.CONSOLE.layout.ConversionPattern=%d{IS
```

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  [NIOServerCx
2016-12-06 19:34:16,237 [myid:1] - INFO  [Thread-1136
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:26,156 [myid:1] - INFO  [Thread-1137
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:26,226 [myid:1] - INFO  [Thread-1138
2016-12-06 19:34:36,151 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:36,152 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:36,152 [myid:1] - INFO  [Thread-1139
2016-12-06 19:34:36,230 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:36,231 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:36,231 [myid:1] - INFO  [Thread-1140
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:46,149 [myid:1] - INFO  [Thread-1141
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCx
2016-12-06 19:34:46,230 [myid:1] - INFO  [Thread-1142
```

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  ST
4 S zookeep+     1     0  0  80   0 -  1127 -      20
0 S zookeep+    27     1  0  80   0 - 1155556 -    20
```

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/da
```

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 /v
```

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": "repla
```

```
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
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 1
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
waiting for statefulset rolling update to complete 2
Waiting for 1 pods to be ready...
Waiting for 1 pods to be ready...
statefulset rolling update complete 3 pods at revisio
```

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
zookeep+    27     1  0 15:03 ?        00:00:03 /usr/
```

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-
```

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
```

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-D
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 {
```

The output is similar to this:

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

Use [`kubectl drain`](https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands/#drain) to cordon and drain the node on which the `zk-0` Pod is scheduled.

```
kubectl drain $(kubectl get pod zk-0 --template {{.sp
```

The output is similar to this:

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

WARNING: Deleting pods not managed by ReplicationCont
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 {{.sp
```

The output is similar to this:

```
"kubernetes-node-ixsl" cordoned
WARNING: Deleting pods not managed by ReplicationCont
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 {{.sp
```

The output is similar to this:

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

WARNING: Deleting pods not managed by ReplicationCont
WARNING: Ignoring DaemonSet-managed pods: node-proble
There are pending pods when an error occurred: Cannot
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 {{.sp
```

The output is similar to this:

```
node "kubernetes-node-i4c4" already cordoned
WARNING: Deleting pods not managed by ReplicationCont
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.

7 - Services

7.1 - 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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      containers:
      - name: my-nginx
        image: nginx
        ports:
        - containerPort: 80
```

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    RESTA
my-nginx-3800858182-jr4a2    1/1       Running   0
my-nginx-3800858182-kna2y    1/1       Running   0
```

Check your pods' IPs:

```
kubectl get pods -l run=my-nginx -o custom-columns=PO
    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 Service. 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 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` the following yaml:

service/networking/nginx-svc.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  ports:
  - port: 80
    protocol: TCP
  selector:
    run: my-nginx
```

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   P
my-nginx   ClusterIP   10.0.162.149   <none>        8
```

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 a 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-n
```

```
NAME             ADDRESSTYPE   PORTS   ENDPOINTS
my-nginx-7vzhx   IPv4          80      10.244.2.5,10.
```

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 |
```

```
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 around 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; kubec

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

```
NAME                        READY     STATUS    RESTA
my-nginx-3800858182-e9ihh   1/1       Running   0
my-nginx-3800858182-j4rm4   1/1       Running   0
```

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

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

```
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   POR
kube-dns   ClusterIP   10.0.0.10    <none>        53/
```

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 -
```

```
Waiting for pod default/curl-131556218-9fnch to be ru
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/ngir
```

```
secret/nginxsecret created
```

```
kubectl get secrets
```

```
NAME                  TYPE
nginxsecret           kubernetes.io/tls
```

And also the configmap:

```
kubectl create configmap nginxconfigmap --from-file=
```

```
configmap/nginxconfigmap created
```

```
kubectl get configmaps
```

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

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 -
# 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/tls
data:
  tls.crt: "LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1J
  tls.key: "LS0tLS1CRUdJTiBQUklWQVRFIEtFWS0tLS0tCk1J
```

Now create the secrets using the file:

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

```
NAME                  TYPE
nginxsecret           kubernetes.io/tls
```

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

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
  labels:
    run: my-nginx
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 80
    protocol: TCP
    name: http
  - port: 443
    protocol: TCP
    name: https
  selector:
    run: my-nginx
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
spec:
  selector:
    matchLabels:
      run: my-nginx
  replicas: 1
  template:
    metadata:
      labels:
        run: my-nginx
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      - name: configmap-volume
        configMap:
          name: nginxconfigmap
      containers:
      - name: nginxhttps
        image: bprashanth/nginxhttps:1.0
        ports:
        - containerPort: 443
        - containerPort: 80
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
        - mountPath: /etc/nginx/conf.d
          name: configmap-volume
```

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 crea
```

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

```
kubectl get pods -l run=my-nginx -o custom-columns=PC
    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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: curl-deployment
spec:
  selector:
    matchLabels:
      app: curlpod
  replicas: 1
  template:
    metadata:
      labels:
        app: curlpod
    spec:
      volumes:
      - name: secret-volume
        secret:
          secretName: nginxsecret
      containers:
      - name: curlpod
        command:
        - sh
        - -c
        - while true; do sleep 1; done
        image: radial/busyboxplus:curl
        volumeMounts:
        - mountPath: /etc/nginx/ssl
          name: secret-volume
```

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

```
NAME                                READY     STATUS
curl-deployment-1515033274-1410r    1/1       Running
```

```
kubectl exec curl-deployment-1515033274-1410r -- curl
...
<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
my-nginx   LoadBalancer   10.0.162.149     xx.xxx.xxx
```

```
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:   a320587ffd19711e5a37606cf4a74
...
```

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]()

7.2 - 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:

- Killercoda
- 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:

```
kubectl create deployment source-ip-app --image=regis
```

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     AG
kubernetes-node-6jst   Ready      <none>    2h      v1
kubernetes-node-cx31   Ready      <none>    2h      v1
kubernetes-node-jj1t   Ready      <none>    2h      v1
```

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=cluste
```

The output is:

```
service/clusterip exposed
```

```
kubectl get svc clusterip
```

The output is similar to:

```
NAME         TYPE        CLUSTER-IP    EXTERNAL-IP
clusterip    ClusterIP   10.0.170.92   <none>
```

And hitting the `ClusterIP` from a pod in the same cluster:

```
kubectl run busybox -it --image=busybox:1.28 --restar
```

The output is similar to this:

```
Waiting for pod default/busybox to be running, status
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 00:00:00:00:0
    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 q
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:f
    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
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=nodepo
```

The output is:

```
service/nodeport exposed
```

```
NODEPORT=$(kubectl get -o jsonpath="{.spec.ports[0].n
NODES=$(kubectl get nodes -o jsonpath='{ $.items[*].s
```

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
```

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:

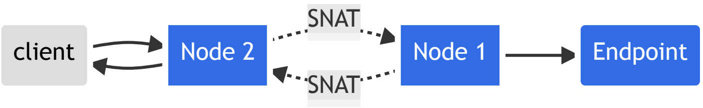


Figure. Source IP Type=NodePort using SNAT

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":{"externalTraf
```

The output is:

```
service/nodeport patched
```

Now, re-run the test:

```
for node in $NODES; do curl --connect-timeout 1 -s $
```

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:

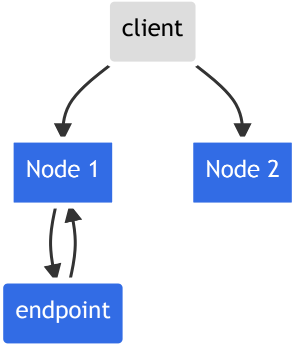


Figure. Source IP Type=NodePort preserves client source IP address

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=loadba
```

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-
loadbalancer   LoadBalancer   10.0.65.118   203.0.113
```

Next, send a request to this Service's external-ip:

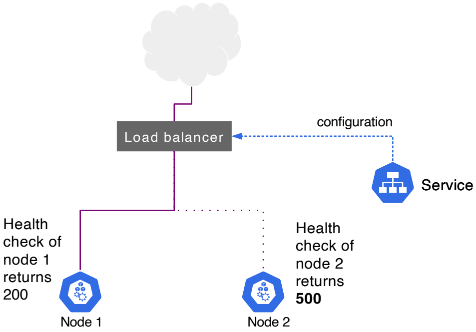
```
curl 203.0.113.140
```

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:



You can test this by setting the annotation:

```
kubectl patch svc loadbalancer -p '{"spec":{"external
```

You should immediately see the `service.spec.healthCheckNodePort` field allocated by Kubernetes:

```
kubectl get svc loadbalancer -o yaml | grep -i health
```

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    R
source-ip-app-826191075-qehz4   1/1       Running   0
```

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