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]()
- [Configuration]()
- [Stateless Applications]()
- [Stateful Applications]()
- [CI/CD Pipeline]()
- [Clusters]()
- [Services]()
- [What's next]()

Basics

- [Kubernetes Basics]() is an in-depth interactive tutorial that helps you understand the Kubernetes system and try out some basic Kubernetes features.

- [Scalable Microservices with Kubernetes (Udacity)]()

- [Introduction to Kubernetes (edX)]()

- [Hello Minikube]()

Configuration

- [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](#)

CI/CD Pipeline

- [Set Up a CI/CD Pipeline with Kubernetes Part 1: Overview](#)

- [Set Up a CI/CD Pipeline with a Jenkins Pod in Kubernetes (Part 2)](#)

- [Run and Scale a Distributed Crossword Puzzle App with CI/CD on Kubernetes (Part 3)](#)

- [Set Up CI/CD for a Distributed Crossword Puzzle App on Kubernetes (Part 4)](#)

Clusters

- [AppArmor](#)

Services

- [Using Source IP](#)

What's next

If you would like to write a tutorial, see [Using Page Templates](#) for information about the tutorial page type and the tutorial template.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Hello Minikube

This tutorial shows you how to run a simple Hello World Node.js 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](#).

- [Objectives](#)

Objectives

- Deploy a hello world application to Minikube.
- Run the app.
- View application logs.

Before you begin

This tutorial provides a container image built from the following files:

minikube/server.js

```javascript
var http = require('http');

var handleRequest = function(request, response) {
  console.log('Received request for URL: ' + request.url);
  response.writeHead(200);
  response.end('Hello World!');
};
var www = http.createServer(handleRequest);
www.listen(8080);
```

minikube/Dockerfile

```dockerfile
FROM node:6.14.2
EXPOSE 8080
COPY server.js .
CMD [ "node", "server.js" ]
```

For more information on the `docker build` command, read the [Docker documentation](#).

Create a Minikube cluster

1. Click **Launch Terminal**

Learn Kubernetes Basics

html

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 in an easy and fast way 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

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](). Open an issue in the GitHub repo if you want to [report a problem]() or [suggest an improvement]().

[Create an Issue]() [Edit This Page]()
Page last modified on October 26, 2019 at 10:31 AM PST by [Fix styling of tutorial pages (#17177)]() ([Page History]())

[Edit This Page]()

Using Minikube to Create a Cluster

html

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 **Master** coordinates the cluster
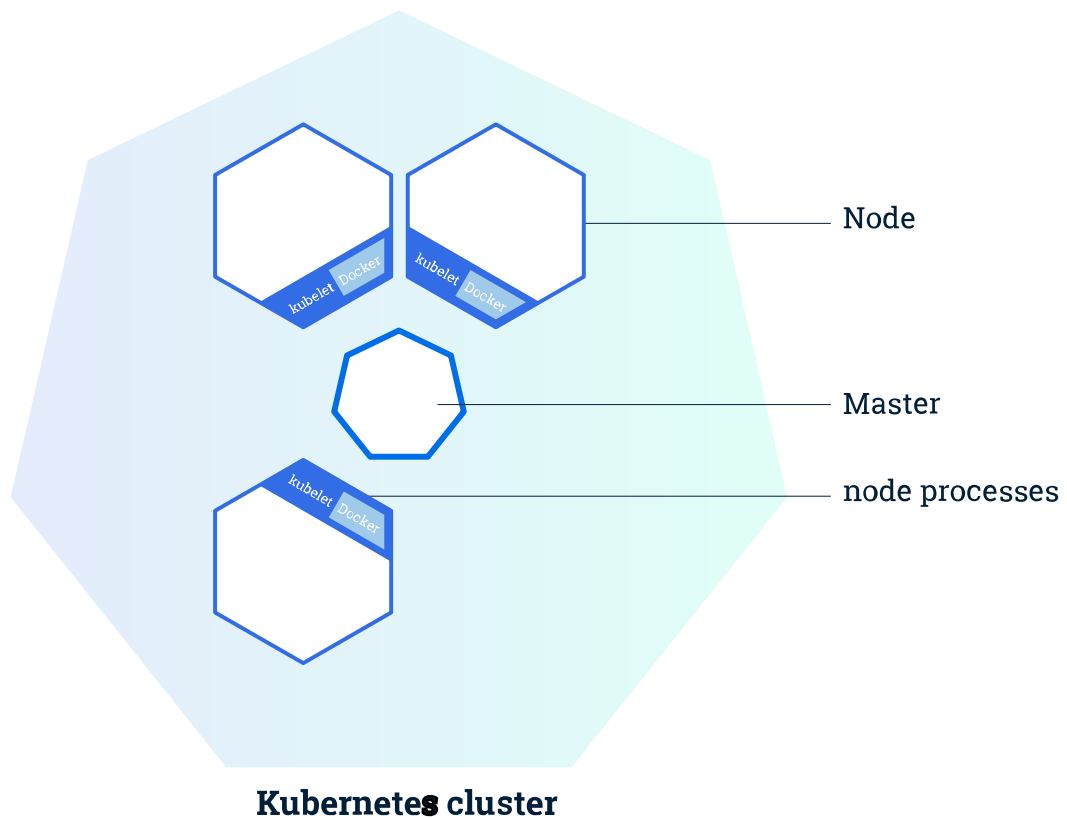- **Nodes** are the workers that run applications

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

Kubernetes cluster

The Master is responsible for managing the cluster. The master 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 master. The node should also have tools for handling container operations, such as Docker or rkt. A Kubernetes cluster that handles production traffic should have a minimum of three nodes.

Masters manage the cluster and the nodes are used to host the running applications.

When you deploy applications on Kubernetes, you tell the master to start the application containers. The master schedules the containers to run on the cluster's nodes. **The nodes communicate with the master using the [Kubernetes API](#)**, which the master 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 â€º

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Interactive Tutorial - Creating a Cluster

html

Using kubectl to Create a Deployment

html

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 master schedules mentioned application instances onto individual Nodes in the cluster.

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

Summary:

- Deployments
- Kubectl

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

Deploying your first app on Kubernetes

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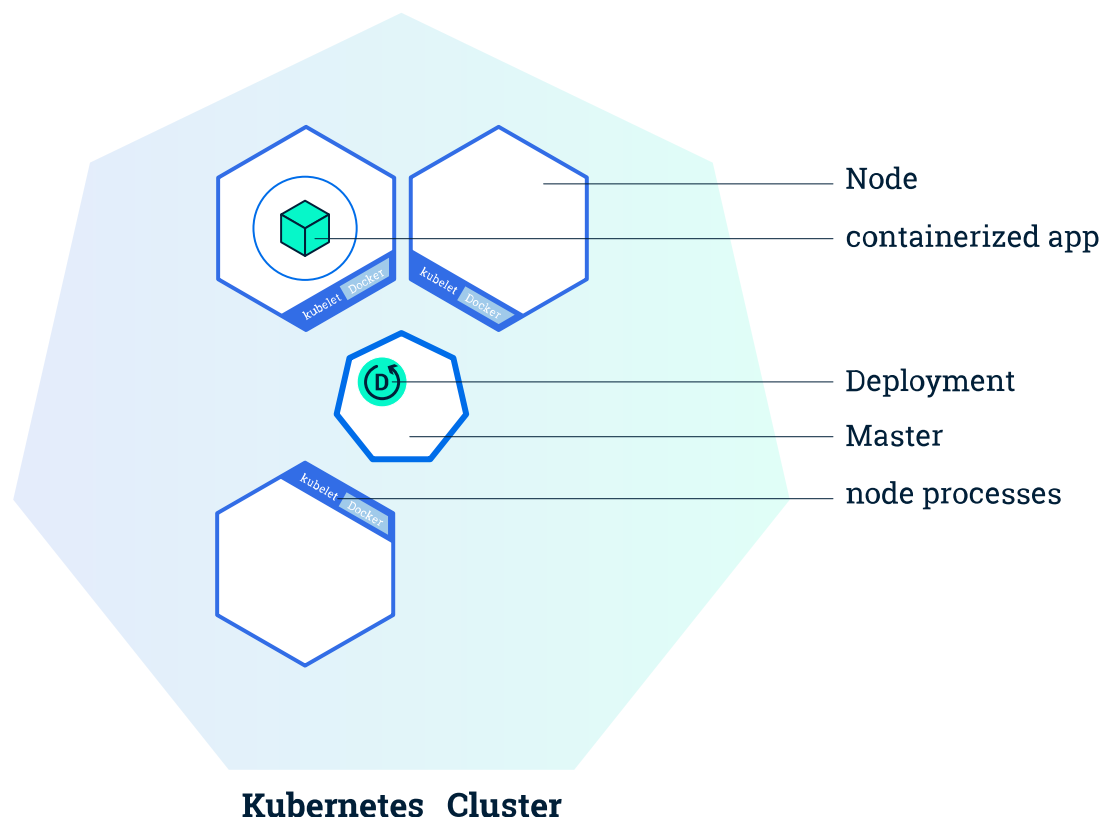
You can create and manage a Deployment by using the Kubernetes command line interface, **Kubectl**. Kubectl uses the Kubernetes API to interact with the cluster. In this module, you'll learn the most common Kubectl commands needed to create Deployments that run your applications on a Kubernetes cluster.

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.

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

For your first Deployment, you'll use a Node.js application packaged in a Docker container. (If you didn't already try creating a Node.js 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!

[Start Interactive Tutorial â€º](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Interactive Tutorial - Deploying an App

html

Viewing Pods and Nodes

html

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 or rkt), 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 or rkt) and includes shared storage (volumes), IP address and information about how to run them.

Pods overview

Pod 1　　**Pod 2**　　**Pod 3**　　**Pod 4**
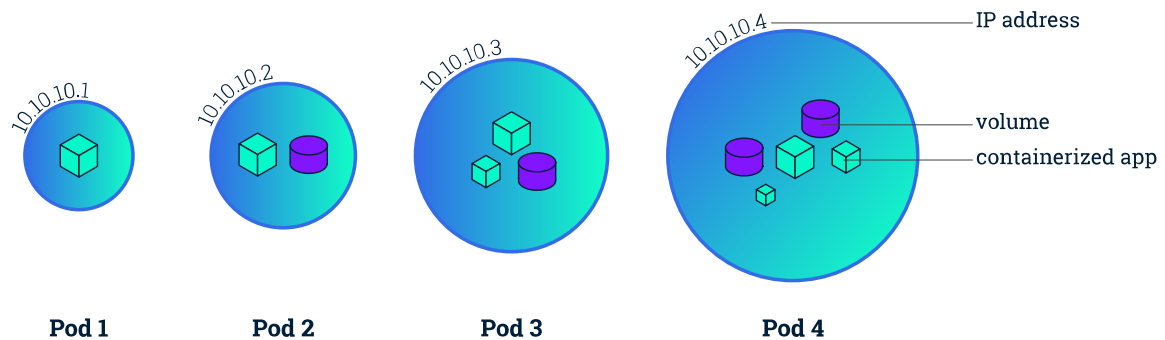
Nodes

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 Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster. The Master's automatic scheduling takes into account the available resources on each Node.

Every Kubernetes Node runs at least:

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

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

Node overview
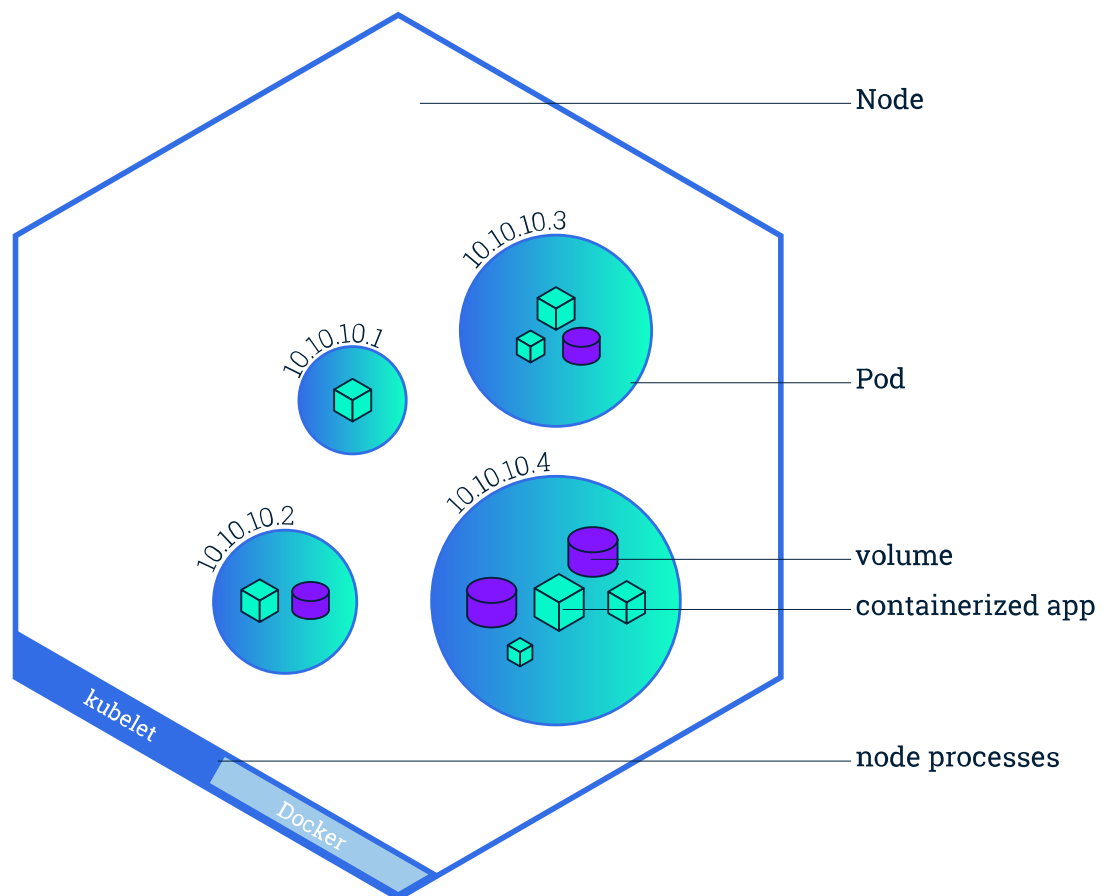
Troubleshooting with kubectl

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.

Start Interactive Tutorial â€º

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on Stack Overflow. Open an issue in the GitHub repo if you want to report a problem or suggest an improvement.

Interactive Tutorial - Exploring Your App

html

Using a Service to Expose Your App

html

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 in fact 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 desired state via 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 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* - Exposes the Service using an arbitrary name (specified by `externalName` in the spec) by returning a CNAME record with the name. No proxy is used. This type requires v1.7 or higher of `kube-dns`.

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 `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`.

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.

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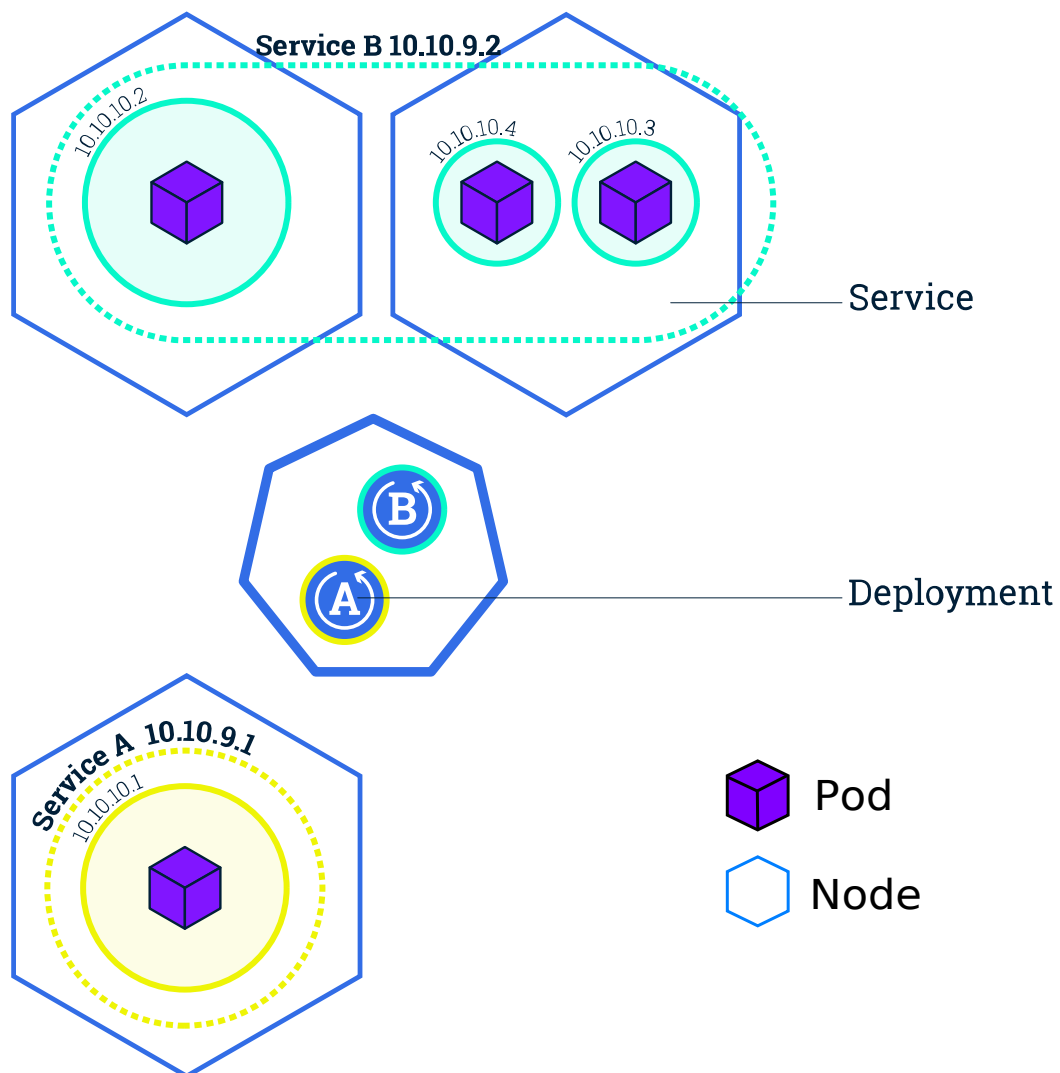
A Service routes traffic across a set of Pods. Services are the abstraction that allow pods to die and replicate in Kubernetes without impacting your application. Discovery and routing among dependent Pods (such as the frontend and backend components in an application) is handled by Kubernetes Services.

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

You can create a Service at the same time you create a Deployment by using
`--expose` *in kubectl.*
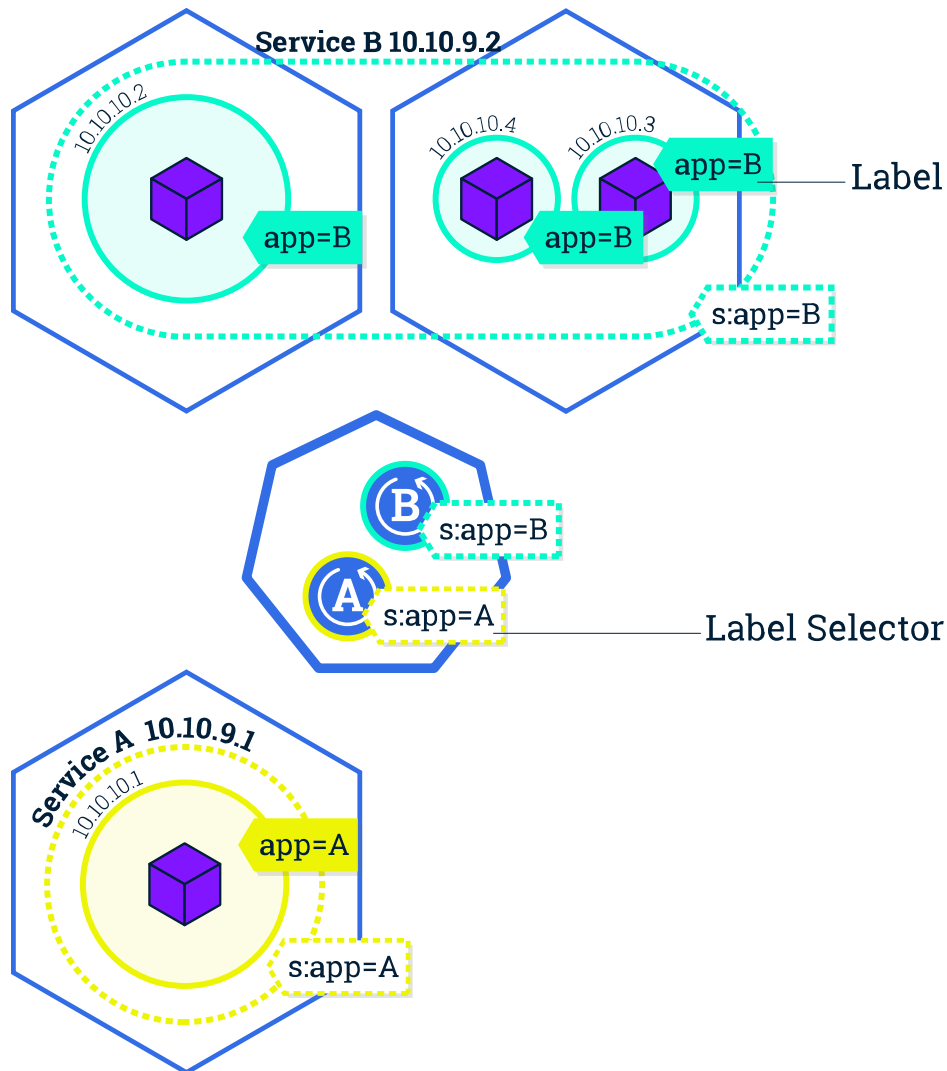
Labels can be attached to objects at creation time or later on. They can be modified at any time. Let's expose our application now using a Service and apply some labels.

[Start Interactive Tutorialâ€º](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](). Open an issue in the GitHub repo if you want to [report a problem]() or [suggest an improvement]().

[Create an Issue]() [Edit This Page]()
Page last modified on January 15, 2019 at 2:28 AM PST by [Fix a broken anchor (#12215)]() ([Page History]())

[Edit This Page]()

Interactive Tutorial - Exposing Your App

html

[Create an Issue](#) [Edit This Page](#)
Page last modified on November 14, 2018 at 10:36 AM PST by [Use absolute URLs and avoid aliases in the tutorial (#10924)](#) ([Page History](#))

[Edit This Page](#)

Running Multiple Instances of Your App

html

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 run command

Scaling overview

1. 2.

Service A 10.3.250.236

10.0.0.5

Service

Deployment

Service A 10.3.250.236

10.0.0.5

Service

10.0.2.4   10.0.2.5

Deployment

10.0.1.5
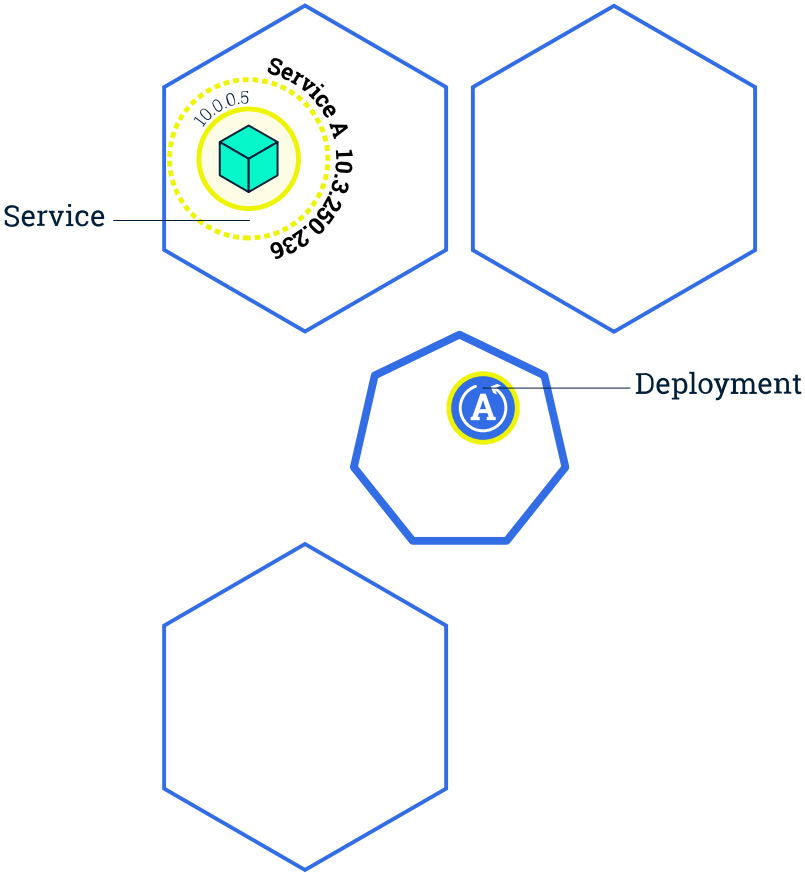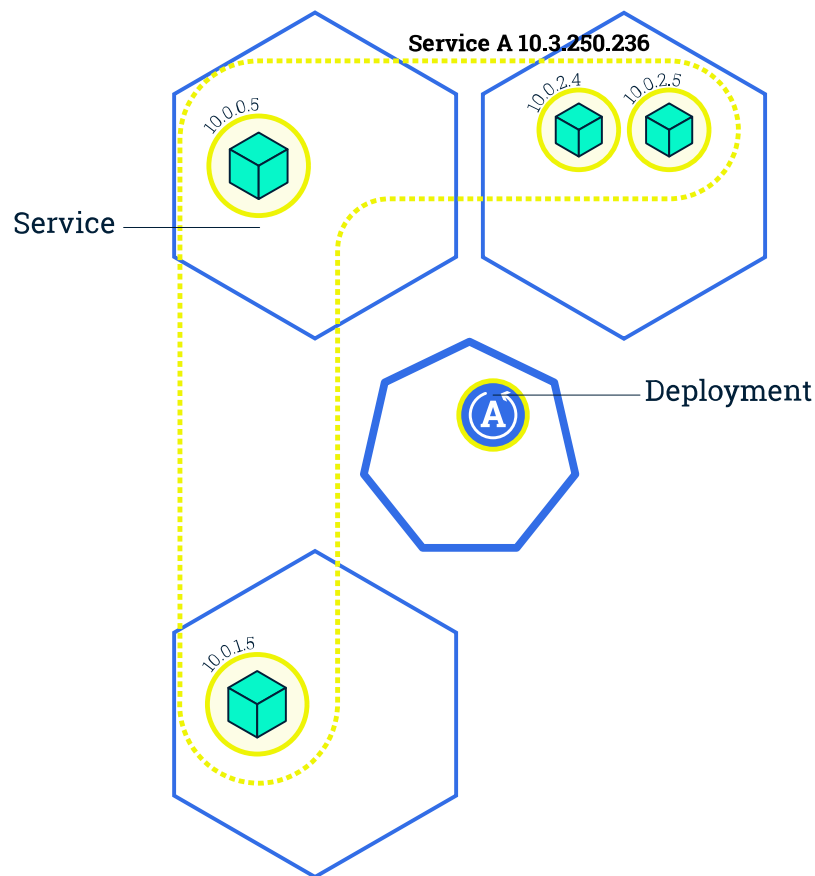
Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling will increase the number of Pods to the new desired state. Kubernetes also supports autoscaling of Pods, but it is outside of the scope of this tutorial. Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment.

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.

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

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.

[Start Interactive Tutorial â€º](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Interactive Tutorial - Scaling Your App

html

Performing a Rolling Update

html

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.

Rolling updates overview

1.

Service A 10.3.250.236

10.0.0.5

10.0.2.4   10.0.2.5

Service

Deployment

A

10.0.1.5

Service A 10.3.250.236

new IP address

updated app

Service

10.0.0.5   10.0.0.6

10.0.2.4   10.0.2.5

Deployment

A

10.0.1.5

Service A 10.3.250.236

10.0.0.6

10.0.2.4 10.0.2.5

Service

Deployment

new IP address

10.0.1.6

updated app

Service A 10.3.250.236

new IP address

updated app

Service

Deployment

10.0.0.6
10.0.2.6
10.0.2.7
10.0.1.6
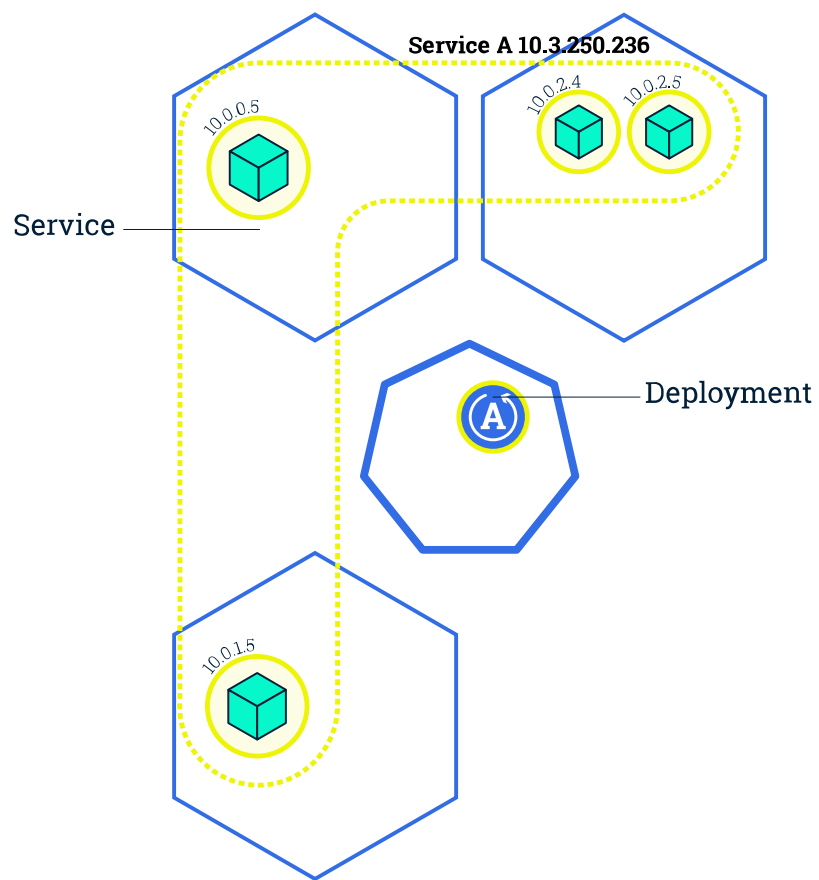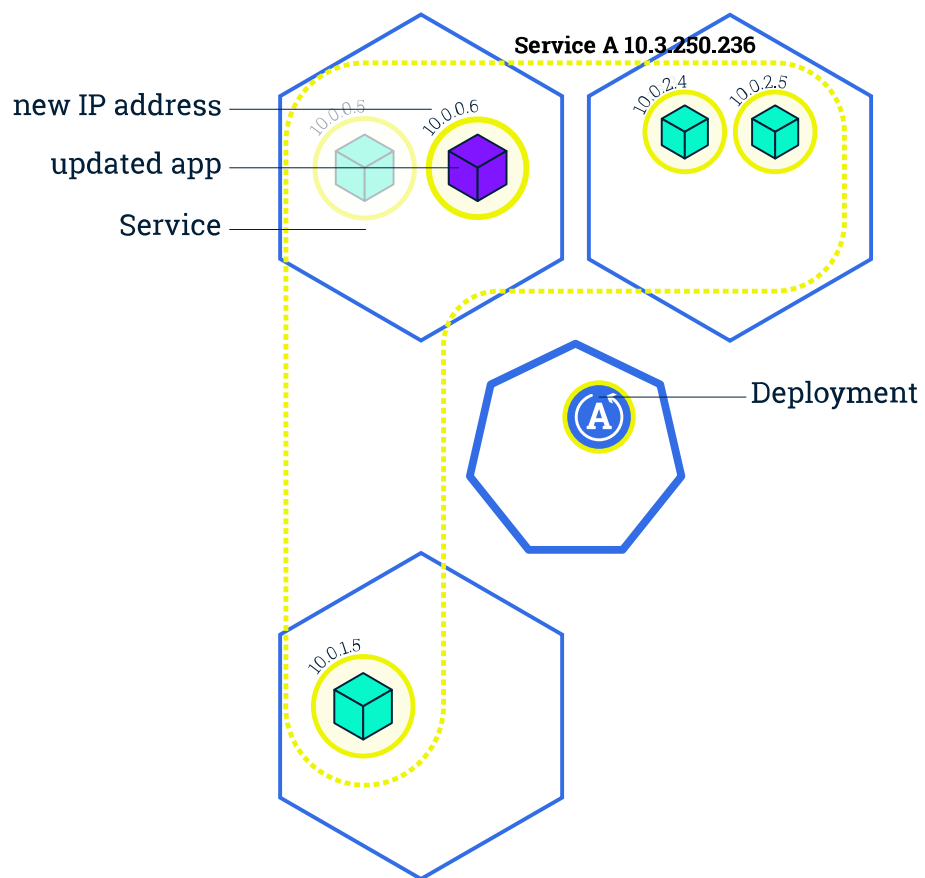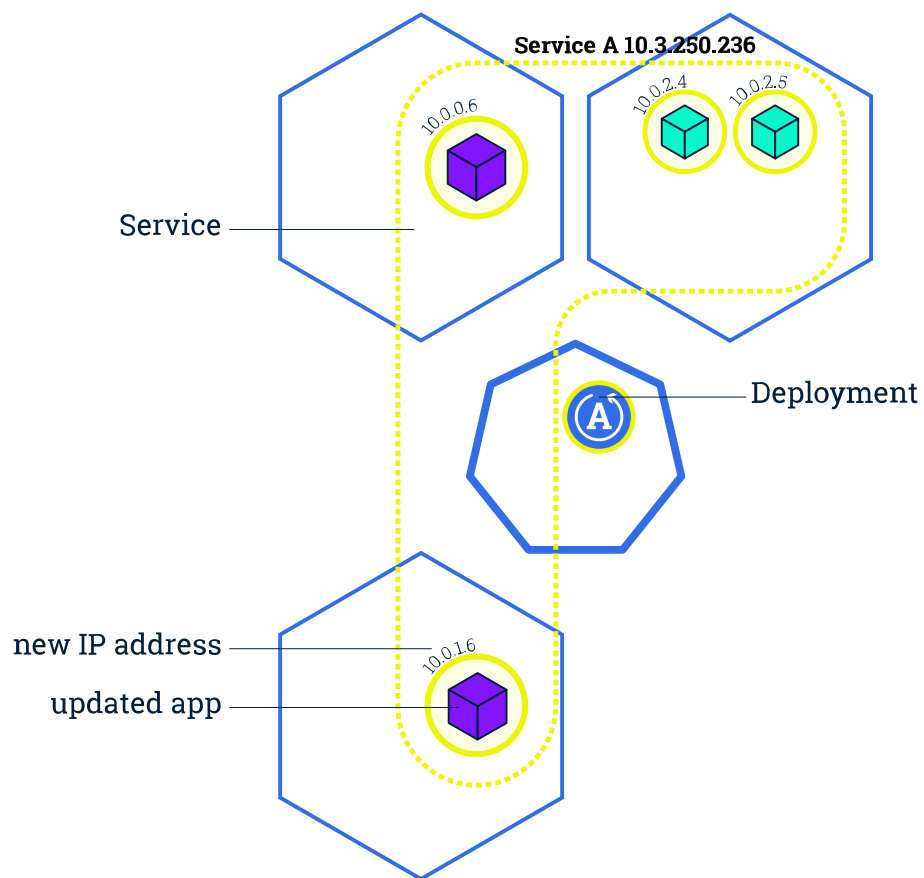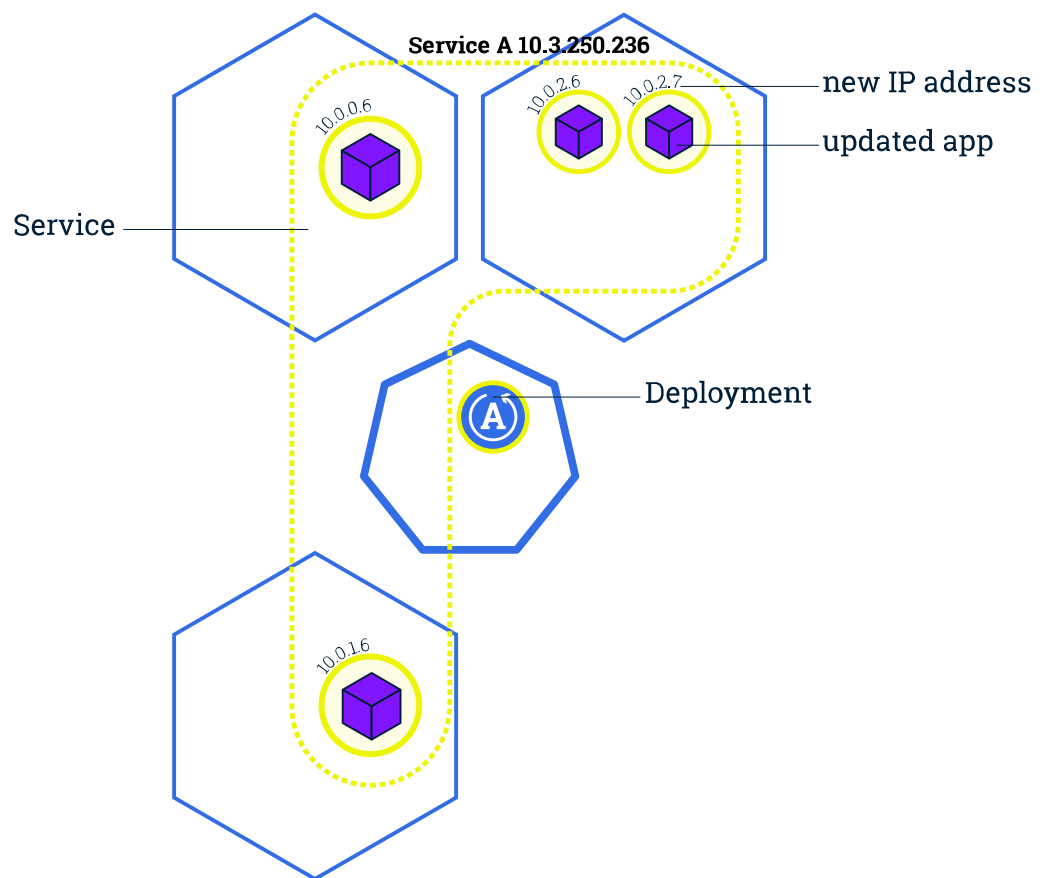
Similar to application Scaling, if a Deployment is exposed publicly, the Service will load-balance the traffic only to available Pods during the update. An available Pod is an instance that is available to the users of the application.

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

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

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

[Start Interactive Tutorial â€º](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Interactive Tutorial - Updating Your App

html

Page last modified on September 19, 2018 at 10:45 AM PST by Adding a back button on last module (#10089) (Page History)

Overview of Kubernetes Online Training

Here are some of the sites that offer online training for Kubernetes:

- [AIOps Essentials (Autoscaling Kubernetes with Prometheus Metrics) with Hands-On Labs (Linux Academy)](#)

- [Amazon EKS Deep Dive with Hands-On Labs (Linux Academy)](#)

- [Cloud Native Certified Kubernetes Administrator (CKA) with Hands-On Labs & Practice Exams (Linux Academy)](#)

- [Certified Kubernetes Administrator (CKA) Preparation Course (CloudYuga)](#)

- [Certified Kubernetes Administrator Preparation Course with Practice Tests (KodeKloud)](#)

- [Certified Kubernetes Application Developer (CKAD) with Hands-On Labs & Practice Exams (Linux Academy)](#)

- [Certified Kubernetes Application Developer (CKAD) Preparation Course (CloudYuga)](#)

- [Certified Kubernetes Application Developer Preparation Course with Practice Tests (KodeKloud)](#)

- [Getting Started with Google Kubernetes Engine (Coursera)](#)

- [Getting Started with Kubernetes (Pluralsight)](#)

- [Getting Started with Kubernetes Clusters on OCI Oracle Kubernetes Engine (OKE) (Learning Library)](#)

- [Google Kubernetes Engine Deep Dive (Linux Academy)](#)

- [Helm Deep Dive with Hands-On Labs (Linux Academy)](#)

- [Hands-on Introduction to Kubernetes (Instruqt)](#)

- [IBM Cloud: Deploying Microservices with Kubernetes (Coursera)](#)

- [Introduction to Kubernetes (edX)](#)

- [Kubernetes Essentials with Hands-On Labs (Linux Academy)](#)

- [Kubernetes for the Absolute Beginners with Hands-on Labs (KodeKloud)](#)

- [Kubernetes Fundamentals (LFS258) (The Linux Foundation)](#)

- [Kubernetes Quick Start with Hands-On Labs (Linux Academy)](#)

- [Kubernetes the Hard Way with Hands-On Labs (Linux Academy)](#)

- [Kubernetes Security with Hands-On Labs (Linux Academy)](#)

- [Launch Your First OpenShift Operator with Hands-On Labs (Linux Academy)](#)

- [Learn Kubernetes by Doing - 100% Hands-On Experience (Linux Academy)](#)

- [Learn Kubernetes using Interactive Hands-on Scenarios (Katacoda)](#)

- [Microservice Applications in Kubernetes - 100% Hands-On Experience (Linux Academy)](#)

- [Monitoring Kubernetes With Prometheus with Hands-On Labs (Linux Academy)](#)

- [Service Mesh with Istio with Hands-On Labs (Linux Academy)](#)

- [Scalable Microservices with Kubernetes (Udacity)](#)

- [Self-paced Kubernetes online course (Learnk8s Academy)](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

[Edit This Page](#)

Configuring Redis using a ConfigMap

This page provides a real world example of how to configure Redis using a ConfigMap and builds upon the [Configure Containers Using a ConfigMap](#) task.

- [Objectives](#)
- [Before you begin](#)
- [Real World Example: Configuring Redis using a ConfigMap](#)
- [What's next](#)

Objectives

- Create a `kustomization.yaml` file containing:
  - a ConfigMap generator
  - a Pod resource config using the ConfigMap
- Apply the directory by running `kubectl apply -k ./`
- 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. If you do not already have a cluster, you can create one by using [Minikube](#), or you can use one of these Kubernetes playgrounds:

- [Katacoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

- The example shown on this page works with `kubectl` 1.14 and above.
- Understand [Configure Containers Using a ConfigMap](#).

Real World Example: Configuring Redis using a ConfigMap

You can follow the steps below to configure a Redis cache using data stored in a ConfigMap.

First create a `kustomization.yaml` containing a ConfigMap from the `redis-config` file:

pods/config/redis-config
maxmemory 2mb maxmemory-policy allkeys-lru

```
curl -OL https://k8s.io/examples/pods/config/redis-config

cat <<EOF >./kustomization.yaml
configMapGenerator:
- name: example-redis-config
  files:
  - redis-config
EOF
```

Add the pod resource config to the `kustomization.yaml`:

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
```

```
curl -OL https://raw.githubusercontent.com/kubernetes/
website/master/content/en/examples/pods/config/redis-pod.yaml

cat <<EOF >>./kustomization.yaml
resources:
- redis-pod.yaml
EOF
```

Apply the kustomization directory to create both the ConfigMap and Pod objects:

```
kubectl apply -k .
```

Examine the created objects by

```
> kubectl get -k .
NAME                                         DATA    AGE
configmap/example-redis-config-dgh9dg555m    1       52s


NAME         READY    STATUS     RESTARTS    AGE
pod/redis    1/1      Running    0           52s
```

In the example, the config volume is mounted at `/redis-master`. It uses `path` to add the `redis-config` key to a file named `redis.conf`. The file path for the redis config, therefore, is `/redis-master/redis.conf`. This is where the image will look for the config file for the redis master.

Use `kubectl exec` to enter the pod and run the `redis-cli` tool to verify that the configuration was correctly applied:

```
kubectl exec -it redis redis-cli
127.0.0.1:6379> CONFIG GET maxmemory
1) "maxmemory"
2) "2097152"
127.0.0.1:6379> CONFIG GET maxmemory-policy
1) "maxmemory-policy"
2) "allkeys-lru"
```

Delete the created pod:

```
kubectl delete pod redis
```

What's next

- Learn more about [ConfigMaps](#).

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Page last modified on December 01, 2019 at 1:21 PM PST by ()

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.

- Objectives
- Before you begin
- Creating a service for an application running in five pods

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.

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.

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-example
  name: hello-world
spec:
  replicas: 5
  selector:
    matchLabels:
      app.kubernetes.io/name: load-balancer-example
  template:
    metadata:
      labels:
        app.kubernetes.io/name: load-balancer-example
    spec:
      containers:
      - image: gcr.io/google-samples/node-hello:1.0
        name: hello-world
        ports:
        - containerPort: 8080
```

```
kubectl apply -f https://k8s.io/examples/service/load-
balancer-example.yaml
```

The preceding command creates a [Deployment](#) object and an associated [ReplicaSet](#) object. The ReplicaSet has five [Pods](#), each of which runs the Hello World application.

1. Display information about the Deployment:

   ```
   kubectl get deployments hello-world
   kubectl describe deployments hello-world
   ```

2. Display information about your ReplicaSet objects:

   ```
   kubectl get replicasets
   kubectl describe replicasets
   ```

3. Create a Service object that exposes the deployment:

   ```
   kubectl expose deployment hello-world --
   type=LoadBalancer --name=my-service
   ```

4. Display information about the Service:

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

   The output is similar to this:

   ```
   NAME          TYPE             CLUSTER-IP      EXTERNAL-
   IP       PORT(S)     AGE
   my-service   LoadBalancer    10.3.245.137
   104.198.205.71   8080/TCP    54s
   ```

   > **Note:** The `type=LoadBalancer` service is backed by external cloud providers, which is not covered in this example, please refer to [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.

5. Display detailed information about the Service:

   ```
   kubectl describe services my-service
   ```

   The output is similar to this:

   ```
   Name:           my-service
   Namespace:      default
   Labels:         app.kubernetes.io/name=load-balancer-
   example
   Annotations:    <none>
   Selector:       app.kubernetes.io/name=load-balancer-
   example
   Type:           LoadBalancer
   ```

```
IP:                10.3.245.137
LoadBalancer Ingress:   104.198.205.71
Port:              <unset> 8080/TCP
NodePort:          <unset> 32377/TCP
Endpoints:
10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...
Session Affinity:    None
Events:            <none>
```

Make a note of the external IP address (`LoadBalancer Ingress`) exposed by your service. In this example, the external IP address is 104.198.205.71. Also note the value of `Port` and `NodePort`. In this example, the `Port` is 8080 and the `NodePort` is 32377.

6. 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 this:

```
NAME                            ...   IP         NODE
hello-world-2895499144-1jaz9 ...   10.0.1.6   gke-
cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-2e5uh ...   10.0.1.8   gke-
cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-9m4h1 ...   10.0.0.6   gke-
cluster-1-default-pool-e0b8d269-5v7a
hello-world-2895499144-o4z13 ...   10.0.1.7   gke-
cluster-1-default-pool-e0b8d269-1afc
hello-world-2895499144-segjf ...   10.0.2.5   gke-
cluster-1-default-pool-e0b8d269-cpuc
```

7. 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](#).

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Page last modified on February 04, 2020 at 3:01 AM PST by doc: add link for type=LoadBalancer service in tutorial. (#18916) (Page History)

Example: Deploying PHP Guestbook application with Redis

This tutorial shows you how to build and deploy a simple, multi-tier web application using Kubernetes and Docker. This example consists of the following components:

- A single-instance Redis master to store guestbook entries
- Multiple replicated Redis instances to serve reads

- Multiple web frontend instances

Objectives

- Start up a Redis master.
- Start up Redis slaves.
- 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. If you do not already have a cluster, you can create one by using [Minikube](#), or you can use one of these Kubernetes playgrounds:

- [Katacoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Start up the Redis Master

The guestbook application uses Redis to store its data. It writes its data to a Redis master instance and reads data from multiple Redis slave instances.

Creating the Redis Master Deployment

The manifest file, included below, specifies a Deployment controller that runs a single replica Redis master Pod.

application/guestbook/redis-master-deployment.yaml

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/
v1beta2
kind: Deployment
metadata:
  name: redis-master
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: master
      tier: backend
  replicas: 1
  template:
    metadata:
      labels:
        app: redis
        role: master
        tier: backend
    spec:
      containers:
      - name: master
        image: k8s.gcr.io/redis:e2e   # or just image: redis
        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 Master Deployment from the `redis-master-deployment.yaml` file:

   ```
   kubectl apply -f https://k8s.io/examples/application/
   guestbook/redis-master-deployment.yaml
   ```

3. Query the list of Pods to verify that the Redis Master Pod is running:

   ```
   kubectl get pods
   ```

   The response should be similar to this:

   ```
   NAME                            READY      STATUS
   RESTARTS    AGE
   ```

```
redis-master-1068406935-3lswp   1/1        Running   0
        28s
```

4. Run the following command to view the logs from the Redis Master Pod:

```
kubectl logs -f POD-NAME
```

Note: Replace POD-NAME with the name of your Pod.

Creating the Redis Master Service

The guestbook applications needs to communicate to the Redis master to write its data. You need to apply a [Service](#) to proxy the traffic to the Redis master Pod. A Service defines a policy to access the Pods.

[application/guestbook/redis-master-service.yaml](#)

```
apiVersion: v1
kind: Service
metadata:
  name: redis-master
  labels:
    app: redis
    role: master
    tier: backend
spec:
  ports:
  - port: 6379
    targetPort: 6379
  selector:
    app: redis
    role: master
    tier: backend
```

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

```
kubectl apply -f https://k8s.io/examples/application/
guestbook/redis-master-service.yaml
```

2. Query the list of Services to verify that the Redis Master Service is running:

```
kubectl get service
```

The response should be similar to this:

```
NAME            TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)    AGE
kubernetes      ClusterIP   10.0.0.1      <none>
```

```
443/TCP      1m
  redis-master    ClusterIP   10.0.0.151    <none>
6379/TCP    8s
```

Note: This manifest file creates a Service named `redis-master` with a set of labels that match the labels previously defined, so the Service routes network traffic to the Redis master Pod.

Start up the Redis Slaves

Although the Redis master is a single pod, you can make it highly available to meet traffic demands by adding replica Redis slaves.

Creating the Redis Slave Deployment

Deployments scale based off of the configurations set in the manifest file. In this case, the Deployment object specifies two replicas.

If there are not any replicas running, this Deployment would start the two replicas on your container cluster. Conversely, if there are more than two replicas are running, it would scale down until two replicas are running.

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/
v1beta2
kind: Deployment
metadata:
  name: redis-slave
  labels:
    app: redis
spec:
  selector:
    matchLabels:
      app: redis
      role: slave
      tier: backend
  replicas: 2
  template:
    metadata:
      labels:
        app: redis
        role: slave
        tier: backend
    spec:
      containers:
      - name: slave
        image: gcr.io/google_samples/gb-redisslave:v3
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your
cluster to
          # provide a dns service. As of Kubernetes 1.3,
DNS is a built-in
          # service launched automatically. However, if the
cluster you are using
          # does not have a built-in DNS service, you can
instead
          # access an environment variable to find the
master
          # service's host. To do so, comment out the
'value: dns' line above, and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 6379
```

1. Apply the Redis Slave Deployment from the `redis-slave-deployment.yaml` file:

```
kubectl apply -f https://k8s.io/examples/application/guestbook/redis-slave-deployment.yaml
```

2. Query the list of Pods to verify that the Redis Slave Pods are running:

```
kubectl get pods
```

The response should be similar to this:

```
   NAME                           READY
STATUS              RESTARTS    AGE
   redis-master-1068406935-3lswp   1/1
Running                 0           1m
   redis-slave-2005841000-fpvqc    0/1
ContainerCreating   0           6s
   redis-slave-2005841000-phfv9    0/1
ContainerCreating   0           6s
```

Creating the Redis Slave Service

The guestbook application needs to communicate to Redis slaves to read data. To make the Redis slaves discoverable, you need to set up a Service. A Service provides transparent load balancing to a set of Pods.

application/guestbook/redis-slave-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: redis-slave
  labels:
    app: redis
    role: slave
    tier: backend
spec:
  ports:
  - port: 6379
  selector:
    app: redis
    role: slave
    tier: backend
```

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

```
kubectl apply -f https://k8s.io/examples/application/guestbook/redis-slave-service.yaml
```

2. Query the list of Services to verify that the Redis slave service is running:

```
kubectl get services
```

The response should be similar to this:

```
  NAME            TYPE        CLUSTER-IP    EXTERNAL-IP
PORT(S)     AGE
  kubernetes      ClusterIP   10.0.0.1      <none>
443/TCP     2m
  redis-master    ClusterIP   10.0.0.151    <none>
6379/TCP    1m
  redis-slave     ClusterIP   10.0.0.223    <none>
6379/TCP    6s
```

Set up and Expose the Guestbook Frontend

The guestbook application has a web frontend serving the HTTP requests written in PHP. It is configured to connect to the `redis-master` Service for write requests and the `redis-slave` service for Read requests.

Creating the Guestbook Frontend Deployment

application/guestbook/frontend-deployment.yaml

```yaml
apiVersion: apps/v1 # for versions before 1.9.0 use apps/
v1beta2
kind: Deployment
metadata:
  name: frontend
  labels:
    app: guestbook
spec:
  selector:
    matchLabels:
      app: guestbook
      tier: frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: guestbook
        tier: frontend
    spec:
      containers:
      - name: php-redis
        image: gcr.io/google-samples/gb-frontend:v4
        resources:
          requests:
            cpu: 100m
            memory: 100Mi
        env:
        - name: GET_HOSTS_FROM
          value: dns
          # Using `GET_HOSTS_FROM=dns` requires your
cluster to
          # provide a dns service. As of Kubernetes 1.3,
DNS is a built-in
          # service launched automatically. However, if the
cluster you are using
          # does not have a built-in DNS service, you can
instead
          # access an environment variable to find the
master
          # service's host. To do so, comment out the
'value: dns' line above, and
          # uncomment the line below:
          # value: env
        ports:
        - containerPort: 80
```

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

   ```
   kubectl apply -f https://k8s.io/examples/application/
   guestbook/frontend-deployment.yaml
   ```

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

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

   The response should be similar to this:

   ```
   NAME                          READY     STATUS
   RESTARTS    AGE
     frontend-3823415956-dsvc5   1/1       Running
   0           54s
     frontend-3823415956-k22zn   1/1       Running
   0           54s
     frontend-3823415956-w9gbt   1/1       Running
   0           54s
   ```

Creating the Frontend Service

The `redis-slave` and `redis-master` Services you applied are only accessible within the container 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 container cluster. Minikube can only expose Services through `NodePort`.

> **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, simply delete or comment out `type: NodePort`, and uncomment `type: LoadBalancer`.

application/guestbook/frontend-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    app: guestbook
    tier: frontend
spec:
  # comment or delete the following line if you want to use
a LoadBalancer
  type: NodePort
  # if your cluster supports it, uncomment the following to
automatically create
  # an external load-balanced IP for the frontend service.
  # type: LoadBalancer
  ports:
  - 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/application/
   guestbook/frontend-service.yaml
   ```

2. Query the list of Services to verify that the frontend Service is running:

   ```
   kubectl get services
   ```

   The response should be similar to this:

   ```
     NAME            TYPE        CLUSTER-IP   EXTERNAL-IP
   PORT(S)         AGE
     frontend        NodePort    10.0.0.112   <none>
   80:31323/TCP    6s
     kubernetes      ClusterIP   10.0.0.1     <none>
   443/TCP         4m
     redis-master    ClusterIP   10.0.0.151   <none>
   6379/TCP        2m
     redis-slave     ClusterIP   10.0.0.223   <none>
   6379/TCP        1m
   ```

Viewing the Frontend Service via `NodePort`

If you deployed this application to Minikube or a local cluster, 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.

   ```
   minikube service frontend --url
   ```

   The response should be similar to this:

   ```
   http://192.168.99.100:31323
   ```

2. Copy the IP address, and load the page in your browser to view your guestbook.

Viewing the Frontend Service via `LoadBalancer`

If you deployed the `frontend-service.yaml` manifest with type: `LoadBalancer` you need to find the IP address to view your Guestbook.

1. Run the following command to get the IP address for the frontend Service.

   ```
   kubectl get service frontend
   ```

   The response should be similar to this:

   ```
    NAME         TYPE          CLUSTER-IP        EXTERNAL-
   IP          PORT(S)        AGE
    frontend    ClusterIP    10.51.242.136
   109.197.92.229     80:32372/TCP    1m
   ```

2. Copy the external IP address, and load the page in your browser to view your guestbook.

Scale the Web Frontend

Scaling up or down is easy because your servers are defined as a Service that uses a Deployment controller.

1. Run the following command to scale up the number of frontend Pods:

   ```
   kubectl scale deployment frontend --replicas=5
   ```

2. Query the list of Pods to verify the number of frontend Pods running:

   ```
   kubectl get pods
   ```

   The response should look similar to this:

```
   NAME                           READY      STATUS
RESTARTS    AGE
   frontend-3823415956-70qj5       1/1        Running
0           5s
   frontend-3823415956-dsvc5       1/1        Running
0           54m
   frontend-3823415956-k22zn       1/1        Running
0           54m
   frontend-3823415956-w9gbt       1/1        Running
0           54m
   frontend-3823415956-x2pld       1/1        Running
0           5s
   redis-master-1068406935-3lswp   1/1        Running
0           56m
   redis-slave-2005841000-fpvqc    1/1        Running
0           55m
   redis-slave-2005841000-phfv9    1/1        Running
0           55m
```

3. Run the following command to scale down the number of frontend Pods:

```
kubectl scale deployment frontend --replicas=2
```

4. Query the list of Pods to verify the number of frontend Pods running:

```
kubectl get pods
```

The response should look similar to this:

```
   NAME                           READY      STATUS
RESTARTS    AGE
   frontend-3823415956-k22zn       1/1        Running
0           1h
   frontend-3823415956-w9gbt       1/1        Running
0           1h
   redis-master-1068406935-3lswp   1/1        Running
0           1h
   redis-slave-2005841000-fpvqc    1/1        Running
0           1h
   redis-slave-2005841000-phfv9    1/1        Running
0           1h
```

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 -l app=guestbook
kubectl delete service -l app=guestbook
```

The responses should be:

```
deployment.apps "redis-master" deleted
deployment.apps "redis-slave" deleted
service "redis-master" deleted
service "redis-slave" 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 be this:

```
No resources found.
```

What's next

- Add [ELK logging and monitoring](#) to your Guestbook application
- Complete the [Kubernetes Basics](#) Interactive Tutorials
- Use Kubernetes to create a blog using [Persistent Volumes for MySQL and Wordpress](#)
- Read more about [connecting applications](#)
- Read more about [Managing Resources](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Example: Add logging and metrics to the PHP / Redis Guestbook example

This tutorial builds upon the [PHP Guestbook with Redis](#) tutorial. Lightweight log, metric, and network data open source shippers, or *Beats*, from Elastic are deployed in the same Kubernetes cluster as the guestbook. The Beats collect, parse, and index the data into Elasticsearch so that you can view and analyze the resulting

operational information in Kibana. This example consists of the following components:

- A running instance of the [PHP Guestbook with Redis tutorial](#)
- Elasticsearch and Kibana
- Filebeat
- Metricbeat
- Packetbeat

Objectives

- Start up the PHP Guestbook with Redis.
- Install kube-state-metrics.
- Create a Kubernetes secret.
- Deploy the Beats.
- View dashboards of your logs and metrics.

Before you begin

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

- [Katacoda](#)
- [Play with Kubernetes](#)

To check the version, enter `kubectl version`.

Additionally you need:

- A running deployment of the [PHP Guestbook with Redis](#) tutorial.

- A running Elasticsearch and Kibana deployment. You can use [Elasticsearch Service in Elastic Cloud](#), run the [download files](#) on your workstation or servers, or the [Elastic Helm Charts](#).

Start up the PHP Guestbook with Redis

This tutorial builds on the [PHP Guestbook with Redis](#) tutorial. If you have the guestbook application running, then you can monitor that. If you do not have it running then follow the instructions to deploy the guestbook and do not perform the **Cleanup** steps. Come back to this page when you have the guestbook running.

Add a Cluster role binding

Create a [cluster level role binding](#) so that you can deploy kube-state-metrics and the Beats at the cluster level (in kube-system).

```
kubectl create clusterrolebinding cluster-admin-binding \
 --clusterrole=cluster-admin --user=<your email associated
with the k8s provider account>
```

Install kube-state-metrics

Kubernetes *kube-state-metrics* is a simple service that listens to the Kubernetes API server and generates metrics about the state of the objects. Metricbeat reports these metrics. Add kube-state-metrics to the Kubernetes cluster that the guestbook is running in.

Check to see if kube-state-metrics is running

```
kubectl get pods --namespace=kube-system | grep kube-state
```

Install kube-state-metrics if needed

```
git clone https://github.com/kubernetes/kube-state-
metrics.git kube-state-metrics
kubectl create -f examples/standard
kubectl get pods --namespace=kube-system | grep kube-state-
metrics
```

Verify that kube-state-metrics is running and ready

```
kubectl get pods -n kube-system -l app.kubernetes.io/name=kub
e-state-metrics
```

Output:

```
NAME                                 READY    STATUS
RESTARTS    AGE
kube-state-metrics-89d656bf8-vdthm   2/2      Running    0
       21s
```

Clone the Elastic examples GitHub repo

```
git clone https://github.com/elastic/examples.git
```

The rest of the commands will reference files in the `examples/beats-k8s-send-anywhere` directory, so change dir there:

```
cd examples/beats-k8s-send-anywhere
```

Create a Kubernetes Secret

A Kubernetes [SecretStores sensitive information, such as passwords, OAuth tokens, and ssh keys.](#) is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in an image; putting it in a Secret object allows for more control over how it is used, and reduces the risk of accidental exposure.

> **Note:** There are two sets of steps here, one for *self managed* Elasticsearch and Kibana (running on your servers or using the Elastic Helm Charts), and a second separate set for the *managed service* Elasticsearch Service in Elastic Cloud. Only create the secret for the type of Elasticsearch and Kibana system that you will use for this tutorial.
>
> ○ [Self Managed](#)
> ○ [Managed service](#)

Self managed

Switch to the **Managed service** tab if you are connecting to Elasticsearch Service in Elastic Cloud.

Set the credentials

There are four files to edit to create a k8s secret when you are connecting to self managed Elasticsearch and Kibana (self managed is effectively anything other than the managed Elasticsearch Service in Elastic Cloud). The files are:

1. ELASTICSEARCH_HOSTS
2. ELASTICSEARCH_PASSWORD
3. ELASTICSEARCH_USERNAME
4. KIBANA_HOST

Set these with the information for your Elasticsearch cluster and your Kibana host. Here are some examples

ELASTICSEARCH_HOSTS

1. A nodeGroup from the Elastic Elasticsearch Helm Chart:

```
["http://elasticsearch-master.default.svc.cluster.local:
9200"]
```

2. A single Elasticsearch node running on a Mac where your Beats are running in Docker for Mac:

```
["http://host.docker.internal:9200"]
```

3. Two Elasticsearch nodes running in VMs or on physical hardware:

```
["http://host1.example.com:9200", "http://
host2.example.com:9200"]
```

Edit `ELASTICSEARCH_HOSTS`

```
vi ELASTICSEARCH_HOSTS
```

ELASTICSEARCH_PASSWORD

Just the password; no whitespace, quotes, or <>:

```
<yoursecretpassword>
```

Edit `ELASTICSEARCH_PASSWORD`

```
vi ELASTICSEARCH_PASSWORD
```

ELASTICSEARCH_USERNAME

Just the username; no whitespace, quotes, or <>:

```
<your ingest username for Elasticsearch>
```

Edit `ELASTICSEARCH_USERNAME`

```
vi ELASTICSEARCH_USERNAME
```

KIBANA_HOST

1. The Kibana instance from the Elastic Kibana Helm Chart. The subdomain `default` refers to the default namespace. If you have deployed the Helm Chart using a different namespace, then your subdomain will be different:

```
"kibana-kibana.default.svc.cluster.local:5601"
```

2. A Kibana instance running on a Mac where your Beats are running in Docker for Mac:

```
"host.docker.internal:5601"
```

3. Two Elasticsearch nodes running in VMs or on physical hardware:

```
"host1.example.com:5601"
```

Edit `KIBANA_HOST`

```
vi KIBANA_HOST
```

Create a Kubernetes secret

This command creates a secret in the Kubernetes system level namespace (kube-system) based on the files you just edited:

```
kubectl create secret generic dynamic-logging \
  --from-file=./ELASTICSEARCH_HOSTS \
  --from-file=./ELASTICSEARCH_PASSWORD \
  --from-file=./ELASTICSEARCH_USERNAME \
  --from-file=./KIBANA_HOST \
  --namespace=kube-system
```

Managed service

This tab is for Elasticsearch Service in Elastic Cloud only, if you have already created a secret for a self managed Elasticsearch and Kibana deployment, then continue with [Deploy the Beats](#).

Set the credentials

There are two files to edit to create a k8s secret when you are connecting to the managed Elasticsearch Service in Elastic Cloud. The files are:

1. ELASTIC_CLOUD_AUTH
2. ELASTIC_CLOUD_ID

Set these with the information provided to you from the Elasticsearch Service console when you created the deployment. Here are some examples:

ELASTIC_CLOUD_ID

```
devk8s:ABC123def456ghi789jkl123mno456pqr789stu123vwx456yza789
bcd012efg345hijj678klm901nop345zEwOTJjMTc5YWQ0YzQ5OThlN2U5MjA
wYTg4NTIzZQ==
```

ELASTIC_CLOUD_AUTH

Just the username, a colon (:), and the password, no whitespace or quotes:

```
elastic:VFxJJf9Tjwer90wnfTghsn8w
```

Edit the required files:

```
vi ELASTIC_CLOUD_ID
vi ELASTIC_CLOUD_AUTH
```

Create a Kubernetes secret

This command creates a secret in the Kubernetes system level namespace (kube-system) based on the files you just edited:

```
kubectl create secret generic dynamic-logging \
  --from-file=./ELASTIC_CLOUD_ID \
  --from-file=./ELASTIC_CLOUD_AUTH \
  --namespace=kube-system
```

[Edit This Page](#)

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.

- [Objectives](#)
- [Before you begin](#)
- [Creating a StatefulSet](#)
- [Pods in a StatefulSet](#)
- [Scaling a StatefulSet](#)
- [Updating StatefulSets](#)
- [Deleting StatefulSets](#)
- [Pod Management Policy](#)
- [Cleaning up](#)

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

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
- kubectl CLI

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.

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

```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: k8s.gcr.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 and the `web` StatefulSet to verify that they were created successfully.

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


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, in order 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 and Ready](#).

Pods in a StatefulSet

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

Examining the Pod's Ordinal Index

Get the StatefulSet's Pods.

```
kubectl get pods -l app=nginx
NAME       READY      STATUS     RESTARTS   AGE
web-0      1/1        Running    0          1m
web-1      1/1        Running    0          1m
```

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

Using Stable Network Identities

Each Pod has a stable hostname based on its ordinal index. Use [kubectl exec](#) to execute the `hostname` command in each Pod.

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

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

```
kubectl run -i --tty --image busybox:1.28 dns-test --restart=Never --rm
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:      web-0.nginx
Address 1: 10.244.1.6

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

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

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

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

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

kubectl run -i --tty --image busybox:1.28 dns-test --
restart=Never --rm /bin/sh
nslookup web-0.nginx
Server:    10.0.0.10
Address 1: 10.0.0.10 kube-dns.kube-system.svc.cluster.local

Name:       web-0.nginx
Address 1: 10.244.1.7

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

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

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

The StatefulSet controller created two PersistentVolumeClaims that are bound to two [PersistentVolumes](). As the cluster used in this tutorial is configured to dynamically provision PersistentVolumes, the PersistentVolumes were created and bound automatically.

The NGINX webservers, by default, will serve an index file at `/usr/share/nginx/html/index.html`. The `volumeMounts` field in the StatefulSets `spec` ensures that the `/usr/share/nginx/html` directory is backed by a PersistentVolume.

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

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

for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

Note:

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

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

Verify the web servers continue to serve their hostnames.

```
for i in 0 1; do kubectl exec -it web-$i -- curl localhost; done
web-0
web-1
```

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

Scaling a StatefulSet

Scaling a StatefulSet refers to increasing or decreasing the number of replicas. 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    AGE
web-0     1/1       Running              0           3h
web-1     1/1       Running              0           3h
web-2     1/1       Running              0           55s
web-3     1/1       Running              0           36s
web-4     0/1       ContainerCreating    0           18s
NAME      READY     STATUS      RESTARTS    AGE
web-4     1/1       Running     0              19s
web-4     1/1       Terminating    0           24s
web-4     1/1       Terminating    0           24s
web-3     1/1       Terminating    0           42s
web-3     1/1       Terminating    0           42s
```

Ordered Pod Termination

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

There are still five PersistentVolumeClaims and five PersistentVolumes.
When exploring a Pod's stable storage, 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.updat
eStrategy` 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":{"updateStrategy":
{"type":"RollingUpdate"}}}'
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":
"replace", "path": "/spec/template/spec/containers/0/image",
"value":"gcr.io/google_containers/nginx-slim:0.8"}]'
statefulset.apps/web patched
```

In another terminal, watch the Pods in the StatefulSet.

```
kubectl get po -l app=nginx -w
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 po web-$p --template '{{range
$i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
k8s.gcr.io/nginx-slim:0.8
k8s.gcr.io/nginx-slim:0.8
k8s.gcr.io/nginx-slim:0.8
```

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

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

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":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":3}}}}'
statefulset.apps/web patched
```

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

```
kubectl patch statefulset web --type='json' -p='[{"op":
"replace", "path": "/spec/template/spec/containers/0/image",
"value":"k8s.gcr.io/nginx-slim:0.7"}]'
statefulset.apps/web patched
```

Delete a Pod in the StatefulSet.

```
kubectl delete po web-2
pod "web-2" deleted
```

Wait for the Pod to be Running and Ready.

```
kubectl get po -l app=nginx -w
NAME        READY     STATUS              RESTARTS    AGE
web-0       1/1       Running             0           4m
web-1       1/1       Running             0           4m
web-2       0/1       ContainerCreating   0           11s
web-2       1/1       Running   0         18s
```

Get the Pod's container.

```
kubectl get po web-2 --template '{{range $i,
$c := .spec.containers}}{{$c.image}}{{end}}'
k8s.gcr.io/nginx-slim:0.8
```

Notice that, even though the update strategy is `RollingUpdate` the StatefulSet controller 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":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":2}}}}'
statefulset.apps/web patched
```

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

```
kubectl get po -l app=nginx -w
NAME        READY     STATUS              RESTARTS    AGE
web-0       1/1       Running             0           4m
web-1       1/1       Running             0           4m
web-2       0/1       ContainerCreating   0           11s
web-2       1/1       Running   0         18s
```

Get the Pod's container.

```
kubectl get po web-2 --template '{{range $i,
$c := .spec.containers}}{{$c.image}}{{end}}'
k8s.gcr.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 po web-1
pod "web-1" deleted
```

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

```
kubectl get po -l app=nginx -w
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` Pods container.

```
kubectl get po web-1 --template '{{range $i,
$c := .spec.containers}}{{$c.image}}{{end}}'
k8s.gcr.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":{"updateStrategy":
{"type":"RollingUpdate","rollingUpdate":{"partition":0}}}}'
statefulset.apps/web patched
```

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

```
kubectl get po -l app=nginx -w
NAME      READY      STATUS            RESTARTS    AGE
web-0     1/1        Running           0           3m
web-1     0/1        ContainerCreating 0           11s
web-2     1/1        Running           0           2m
web-1     1/1        Running   0       18s
web-0     1/1        Terminating  0       3m
web-0     1/1        Terminating  0       3m
web-0     0/1        Terminating  0       3m
web-0     0/1        Terminating  0       3m
web-0     0/1        Terminating  0       3m
web-0     0/1        Terminating  0       3m
web-0     0/1        Pending   0       0s
web-0     0/1        Pending   0       0s
web-0     0/1        ContainerCreating  0          0s
web-0     1/1        Running   0       3s
```

Get the Pod's containers.

```
for p in 0 1 2; do kubectl get po web-$p --template '{{range
$i, $c := .spec.containers}}{{$c.image}}{{end}}'; echo; done
k8s.gcr.io/nginx-slim:0.7
k8s.gcr.io/nginx-slim:0.7
k8s.gcr.io/nginx-slim:0.7
```

By moving the `partition` to `0`, you allowed the StatefulSet controller 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=false` 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=false
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 simply 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 -it web-$i -- curl localhost; done
web-0
web-1
```

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

Cascading Delete

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

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

In another terminal, delete the StatefulSet again. This time, omit the `--cascade=false` 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 that, while a cascading delete will delete the StatefulSet and its Pods, it will 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 -it web-$i -- curl localhost; done
web-0
web-1
```

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

Finally delete the web StatefulSet and the nginx service.

```
kubectl delete service nginx
service "nginx" deleted

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.

```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: k8s.gcr.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 po -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 po -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 controller launched two new Pods, and it did not wait for the first to become Running and Ready prior to launching the second.

Keep this terminal open, and in another terminal delete the `web` StatefulSet.

```
kubectl delete sts web
```

Again, examine the output of the `kubectl get` command running in the other terminal.

```
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
```

The StatefulSet controller deletes 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
```

Cleaning up

You will 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.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](). Open an issue in the GitHub repo if you want to [report a problem]() or [suggest an improvement]().

[Create an Issue]() [Edit This Page]()
Page last modified on December 16, 2019 at 6:17 AM PST by [Fix Pods concepts link in StatefulSet tutorial (#18063)]() ([Page History]())

[Edit This Page]()

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](#)
- [Before you begin](#)
- [Create PersistentVolumeClaims and PersistentVolumes](#)
- [Create a kustomization.yaml](#)
- [Add resource configs for MySQL and WordPress](#)
- [Apply and Verify](#)
- [Cleaning up](#)
- [What's next](#)

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

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

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.

```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 # for versions before 1.9.0 use apps/v1beta2
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
```

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.

```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 # for versions before 1.9.0 use apps/
v1beta2
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:
```

1. Download the MySQL deployment configuration file.

   ```
   curl -LO https://k8s.io/examples/application/
   wordpress/mysql-deployment.yaml
   ```

2. Download the WordPress configuration file.

   ```
   curl -LO https://k8s.io/examples/application/
   wordpress/wordpress-deployment.yaml
   ```

3. Add them to `kustomization.yaml` file.

```
cat <<EOF >>./kustomization.yaml
resources:
- mysql-deployment.yaml
- wordpress-deployment.yaml
EOF
```

Apply and Verify

The `kustomization.yaml` contains all the resources for deploying a WordPress site and a MySQL database. You can apply the directory by

```
kubectl apply -k ./
```

Now you can verify that all objects exist.

1. Verify that the Secret exists by running the following command:

   ```
   kubectl get secrets
   ```

   The response should be like this:

   ```
   NAME
   TYPE                                        DATA   AGE
   mysql-pass-c57bb4t7mf
   Opaque                                      1      9s
   ```

2. Verify that a PersistentVolume got dynamically provisioned.

   ```
   kubectl get pvc
   ```

   > **Note:** It can take up to a few minutes for the PVs to be provisioned and bound.

   The response should be like this:

   ```
   NAME               STATUS
   VOLUME                                      CAPACITY
   ACCESS MODES   STORAGECLASS        AGE
   mysql-pv-claim    Bound       pvc-8cbd7b2e-4044-11e9-
   b2bb-42010a800002   20Gi        RWO
   standard             77s
   ```

```
   wp-pv-claim        Bound        pvc-8cd0df54-4044-11e9-
b2bb-42010a800002   20Gi          RWO
standard            77s
```

3. Verify that the Pod is running by running the following command:

```
kubectl get pods
```

> **Note:** It can take up to a few minutes for the Pod's
> Status to be RUNNING.

The response should be like this:

```
   NAME                                      READY
STATUS     RESTARTS    AGE
   wordpress-mysql-1894417608-x5dzt    1/1
Running    0            40s
```

4. Verify that the Service is running by running the following command:

```
kubectl get services wordpress
```

The response should be like this:

```
   NAME           TYPE          CLUSTER-IP    EXTERNAL-IP
PORT(S)          AGE
   wordpress    ClusterIP    10.0.0.89    <pending>
80:32406/TCP    4m
```

> **Note:** Minikube can only expose Services through NodeP
> ort. 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.

English (United States)
العربية المغربية
العربية
Azərbaycan dili
گؤنئی آذربایجان
Български
বাংলা
Bosanski
Català
Cebuano
Cymraeg
Dansk
Deutsch (Schweiz)
Deutsch (Sie)

Continue

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](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Example: Deploying Cassandra with Stateful Sets

This tutorial shows you how to develop a native cloud Cassandra deployment on Kubernetes. In this example, a custom Cassandra *SeedProvider* enables Cassandra to discover new Cassandra nodes as they join the cluster.

StatefulSets make it easier to deploy stateful applications within a clustered environment. For more information on the features used in this tutorial, see the *StatefulSet* documentation.

Cassandra on Docker

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`.

ENV VAR	DEFAULT VALUE
`CASSANDRA_CLUSTER_NAME`	`'Test Cluster'`
`CASSANDRA_NUM_TOKENS`	`32`
`CASSANDRA_RPC_ADDRESS`	`0.0.0.0`

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

To complete this tutorial, you should already have a basic familiarity with [Pods](), [Services](), and [StatefulSets](). In addition, you should:

- [Install and Configure]() the *kubectl* command-line tool

- Download [cassandra-service.yaml]() and [cassandra-statefulset.yaml]()

- Have a supported Kubernetes cluster running

  Note: Please read the [setup]() if you do not already have a cluster.

Additional Minikube Setup Instructions

  Caution:

[Minikube](#) defaults to 1024MB of memory and 1 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 Cassandra Headless Service

A Kubernetes [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 the Kubernetes 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
```

1. Launch a terminal window in the directory you downloaded the manifest files.

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

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

Validating (optional)

Get the Cassandra Service.

```
kubectl get svc cassandra
```

The response is

```
NAME        TYPE        CLUSTER-IP   EXTERNAL-IP
PORT(S)     AGE
cassandra   ClusterIP   None         <none>         9042/
TCP    45s
```

Service creation failed if anything else is returned. Read [Debug Services](#) for 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.

```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.svc.cluste
```

1. Update the StatefulSet if necessary.

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

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

Validating The Cassandra StatefulSet

1. Get the Cassandra StatefulSet:

   ```
   kubectl get statefulset cassandra
   ```

   The response should be:

   ```
   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:

   ```
   NAME          READY     STATUS              RESTARTS
   AGE
   cassandra-0   1/1       Running             0
   1m
   cassandra-1   0/1       ContainerCreating   0
   8s
   ```

   It can take several minutes for all three Pods to deploy. Once they are deployed, the same command returns:

   ```
   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](#) to display the status of the ring.

   ```
   kubectl exec -it cassandra-0 -- nodetool status
   ```

   The response should look something like this:

   ```
   Datacenter: DC1-K8Demo
   ======================
   Status=Up/Down
   |/ State=Normal/Leaving/Joining/Moving
   --  Address      Load      Tokens      Owns
   ```

```
(effective)  Host ID                                          Rack
UN  172.17.0.5  83.57 KiB  32
74.0%               e2dd09e6-d9d3-477e-96c5-45094c08db0f
Rack1-K8Demo
UN  172.17.0.4  101.04 KiB  32
58.8%               f89d6835-3a42-4419-92b3-0e62cae1479c
Rack1-K8Demo
UN  172.17.0.6  84.74 KiB  32
67.1%               a6a1e8c2-3dc5-4417-b1a0-26507af2aaad
Rack1-K8Demo
```

Modifying the Cassandra StatefulSet

Use `kubectl edit` to modify the size of a Cassandra StatefulSet.

1. Run the following command:

   ```
   kubectl edit statefulset cassandra
   ```

   This command opens an editor in your terminal. The line you need
   to change is the `replicas` field. The following sample is an excerpt
   of the `StatefulSet` file:

   ```
   # Please edit the object below. Lines beginning with a
   '#' will be ignored,
   # and an empty file will abort the edit. If an error
   occurs while saving this file will be
   # reopened with the relevant failures.
   #
   apiVersion: apps/v1 # for versions before 1.9.0 use
   apps/v1beta2
   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 contains 4 Pods.

3. Get the Cassandra StatefulSet to verify:

   ```
   kubectl get statefulset cassandra
   ```

The response should be

```
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 po cassandra-0 -o=jsonpath='{.spec.terminationGracePeriodSeconds}') \
  && kubectl delete statefulset -l app=cassandra \
  && echo "Sleeping $grace" \
  && sleep $grace \
  && kubectl delete pvc -l app=cassandra
```

2. Run the following command to delete the Cassandra Service.

```
kubectl delete service -l app=cassandra
```

What's next

- Learn how to [Scale a StatefulSet](#).
- Learn more about the *KubernetesSeedProvider*
- See more custom [Seed Provider Configurations](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).

Page last modified on August 09, 2019 at 4:59 AM PST by **Remove references to selflinks (#15751)** (Page History)

Running ZooKeeper, A Distributed System Coordinator

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

- Objectives
- Before you begin
- Creating a ZooKeeper Ensemble
- Ensuring Consistent Configuration
- Managing the ZooKeeper Process

Objectives

After this tutorial, you will know the following.

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

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 will require 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.

ZooKeeper Basics

[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.

```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/v1beta1
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
```

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

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

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

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

Use [kubectl get](#) to watch the StatefulSet controller create the StatefulSet's Pods.

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

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

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

The StatefulSet controller creates three Pods, and each Pod has a container with a [ZooKeeper](#) server.

Facilitating Leader Election

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

Use [kubectl exec](#) to get the hostnames of the Pods in the `zk` StatefulSet.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Achieving Consensus

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

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

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

The A records for each Pod are entered when the Pod becomes Ready. Therefore, the FQDNs of the ZooKeeper servers will resolve to a single

endpoint, and that endpoint will be the unique ZooKeeper server claiming the identity configured in its `myid` file.

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

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

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

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

Sanity Testing the Ensemble

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

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

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

WATCHER::

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

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

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

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

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
world
cZxid = 0x100000002
ctime = Thu Dec 08 15:13:30 UTC 2016
mZxid = 0x100000002
mtime = Thu Dec 08 15:13:30 UTC 2016
pZxid = 0x100000002
cversion = 0
```

```
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
```

Providing Durable Storage

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

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

```
kubectl delete statefulset zk
statefulset.apps "zk" deleted
```

Watch the termination of the Pods in the StatefulSet.

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

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

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

Reapply the manifest in `zookeeper.yaml`.

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

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

Watch the StatefulSet controller recreate the StatefulSet's Pods.

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

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

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

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

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

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

```
WATCHER::

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

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

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

```
      requests:
        storage: 20Gi
```

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

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

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

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

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

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

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

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

Ensuring Consistent Configuration

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

Get the `zk` StatefulSet.

```
kubectl get sts zk -o yaml
â€¦
command:
```

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

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

Configuring Logging

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

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

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

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

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

This is the simplest possible way to safely log inside the container. Because the applications write logs to standard out, Kubernetes will

handle log rotation for you. Kubernetes also implements a sane retention policy that ensures application logs written to standard out and standard error do not exhaust local storage media.

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

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

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

```
2016-12-06 19:34:16,236 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52740
2016-12-06 19:34:16,237 [myid:1] - INFO  [Thread-1136:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52740 (no session established for client)
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52749
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52749
2016-12-06 19:34:26,156 [myid:1] - INFO  [Thread-1137:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52749 (no session established for client)
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52750
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52750
2016-12-06 19:34:26,226 [myid:1] - INFO  [Thread-1138:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52750 (no session established for client)
2016-12-06 19:34:36,151 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52760
2016-12-06 19:34:36,152 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52760
2016-12-06 19:34:36,152 [myid:1] - INFO  [Thread-1139:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52760 (no session established for client)
2016-12-06 19:34:36,230 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52761
2016-12-06 19:34:36,231 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52761
2016-12-06 19:34:36,231 [myid:1] - INFO  [Thread-1140:NIOServ
erCnxn@1008] - Closed socket connection for client /
```

```
127.0.0.1:52761 (no session established for client)
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52767
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52767
2016-12-06 19:34:46,149 [myid:1] - INFO  [Thread-1141:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52767 (no session established for client)
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxnFactory@192] - Accepted
socket connection from /127.0.0.1:52768
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factor
y:0.0.0.0/0.0.0.0:2181:NIOServerCnxn@827] - Processing ruok c
ommand from /127.0.0.1:52768
2016-12-06 19:34:46,230 [myid:1] - INFO  [Thread-1142:NIOServ
erCnxn@1008] - Closed socket connection for client /
127.0.0.1:52768 (no session established for client)
```

Kubernetes supports more powerful, but more complex, logging integrations with Stackdriver and Elasticsearch and Kibana. For cluster level log shipping and aggregation, consider deploying a sidecar container to rotate and ship your logs.

Configuring a Non-Privileged User

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

The zk StatefulSet's Pod template contains a SecurityContext.

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

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

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

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

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

```
F S UID         PID  PPID  C PRI  NI ADDR SZ WCHAN  STIME
TTY          TIME CMD
4 S zookeep+     1     0  0  80   0 -  1127 -       20:
46 ?        00:00:00 sh -c zkGenConfig.sh && zkServer.sh
```

```
start-foreground
0 S zookeep+   27   1  0  80   0 - 1155556 -   20:
46 ?       00:00:19 /usr/lib/jvm/java-8-openjdk-amd64/bin/
java -Dzookeeper.log.dir=/var/log/zookeeper -
Dzookeeper.root.logger=INFO,CONSOLE -cp /usr/bin/../build/
classes:/usr/bin/../build/lib/*.jar:/usr/bin/../share/
zookeeper/zookeeper-3.4.9.jar:/usr/bin/../share/zookeeper/
slf4j-log4j12-1.6.1.jar:/usr/bin/../share/zookeeper/slf4j-
api-1.6.1.jar:/usr/bin/../share/zookeeper/
netty-3.10.5.Final.jar:/usr/bin/../share/zookeeper/
log4j-1.2.16.jar:/usr/bin/../share/zookeeper/
jline-0.9.94.jar:/usr/bin/../src/java/lib/*.jar:/usr/bin/../
etc/zookeeper: -Xmx2G -Xms2G -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.local.only=false
org.apache.zookeeper.server.quorum.QuorumPeerMain /usr/
bin/../etc/zookeeper/zoo.cfg
```

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

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

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

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

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

Managing the ZooKeeper Process

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

Updating the Ensemble

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

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

```
kubectl patch sts zk --type='json' -p='[{"op": "replace",
"path": "/spec/template/spec/containers/0/resources/requests/
cpu", "value":"0.3"}]'
```

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

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

```
kubectl rollout status sts/zk

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

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

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

```
kubectl rollout history sts/zk

statefulsets "zk"
REVISION
1
2
```

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

```
kubectl rollout undo sts/zk

statefulset.apps/zk rolled back
```

Handling Process Failure

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

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

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

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

```
UID          PID   PPID  C STIME TTY          TIME CMD
zookeep+      1      0  0 15:03 ?        00:00:00 sh -c
zkGenConfig.sh && zkServer.sh start-foreground
zookeep+     27      1  0 15:03 ?        00:00:03 /usr/lib/jvm/
java-8-openjdk-amd64/bin/java -Dzookeeper.log.dir=/var/log/
zookeeper -Dzookeeper.root.logger=INFO,CONSOLE -cp /usr/
bin/../build/classes:/usr/bin/../build/lib/*.jar:/usr/bin/../
share/zookeeper/zookeeper-3.4.9.jar:/usr/bin/../share/
zookeeper/slf4j-log4j12-1.6.1.jar:/usr/bin/../share/
zookeeper/slf4j-api-1.6.1.jar:/usr/bin/../share/zookeeper/
netty-3.10.5.Final.jar:/usr/bin/../share/zookeeper/
log4j-1.2.16.jar:/usr/bin/../share/zookeeper/
jline-0.9.94.jar:/usr/bin/../src/java/lib/*.jar:/usr/bin/../
etc/zookeeper: -Xmx2G -Xms2G -Dcom.sun.management.jmxremote -
Dcom.sun.management.jmxremote.local.only=false
org.apache.zookeeper.server.quorum.QuorumPeerMain /usr/
bin/../etc/zookeeper/zoo.cfg
```

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 `zkOk.sh` script from the file system of Pod `zk-0`.

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

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

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

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

```
zk-0     0/1     Running   0          1h
zk-0     0/1     Running   1          1h
zk-0     1/1     Running   1          1h
```

Testing for Readiness

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

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

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

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

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

Tolerating Node Failure

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

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

You should always provision additional capacity to allow the processes of critical systems to be rescheduled in the event of node failures. If you do so, then the outage will only last until the Kubernetes scheduler reschedules one of the ZooKeeper servers. However, if you want your

service to tolerate node failures with no downtime, you should set `podAntiAffinity`.

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

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

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

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

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

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

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

Surviving Maintenance

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

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

Use this command to get the nodes in your cluster.

```
kubectl get nodes
```

Use [kubectl cordon](#) to cordon all but four of the nodes in your cluster.

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

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

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

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

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

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

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

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

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

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

Use [kubectl drain](#) to cordon and drain the node on which the `zk-0` Pod is scheduled.

```
kubectl drain $(kubectl get pod zk-0 --template {{.spec.nodeN
ame}}) --ignore-daemonsets --force --delete-local-data
node "kubernetes-node-pb41" cordoned

WARNING: Deleting pods not managed by ReplicationController,
ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-
kubernetes-node-pb41, kube-proxy-kubernetes-node-pb41;
Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-
o5elz
pod "zk-0" deleted
node "kubernetes-node-pb41" drained
```

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

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

```
zk-0     0/1        Terminating   2          2h
zk-0     0/1        Terminating   2          2h
zk-0     0/1        Pending   0          0s
zk-0     0/1        Pending   0          0s
zk-0     0/1        ContainerCreating   0          0s
zk-0     0/1        Running   0          51s
zk-0     1/1        Running   0          1m
```

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

```
kubectl drain $(kubectl get pod zk-1 --template {{.spec.nodeN
ame}}) --ignore-daemonsets --force --delete-local-data "kuber
netes-node-ixsl" cordoned

WARNING: Deleting pods not managed by ReplicationController,
ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-
kubernetes-node-ixsl, kube-proxy-kubernetes-node-ixsl;
Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-
voc74
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

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 stateful set, and drain the node on which `zk-2` is scheduled.

```
kubectl drain $(kubectl get pod zk-2 --template {{.spec.nodeN
ame}}) --ignore-daemonsets --force --delete-local-data
node "kubernetes-node-i4c4" cordoned

WARNING: Deleting pods not managed by ReplicationController,
ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-
kubernetes-node-i4c4, kube-proxy-kubernetes-node-i4c4;
Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-
dyrog
WARNING: Ignoring DaemonSet-managed pods: node-problem-
detector-v0.1-dyrog; Deleting pods not managed by
ReplicationController, ReplicaSet, Job, or DaemonSet:
fluentd-cloud-logging-kubernetes-node-i4c4, kube-proxy-
kubernetes-node-i4c4
There are pending pods when an error occurred: Cannot evict
pod as it would violate the pod's disruption budget.
pod/zk-2
```

Use `CTRL-C` to terminate to 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
```

```
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
```

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

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

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

The output:

```
node "kubernetes-node-i4c4" already cordoned
WARNING: Deleting pods not managed by ReplicationController,
ReplicaSet, Job, or DaemonSet: fluentd-cloud-logging-
kubernetes-node-i4c4, kube-proxy-kubernetes-node-i4c4;
Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-
dyrog
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
```

```
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 will 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.

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](). Open an issue in the GitHub repo if you want to [report a problem]() or [suggest an improvement]().

AppArmor

FEATURE STATE: `Kubernetes v1.4` [beta](#)
This feature is currently in a *beta* state, meaning:

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.

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

Before you begin

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

- Katacoda
- Play with Kubernetes

To check the version, enter `kubectl version`.

Terminology

This document makes use of the following terms:

- NAT: network address translation
- Source NAT: replacing the source IP on a packet, usually with a node's IP
- Destination NAT: replacing the destination IP on a packet, usually with a pod IP
- VIP: a virtual IP, such as the one assigned to every Kubernetes Service
- Kube-proxy: a network daemon that orchestrates Service VIP management on every node

Prerequisites

You must have a working Kubernetes 1.5 cluster to run the examples in this document. 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=k8s.gcr.io/
echoserver:1.4
```

The output is:

```
deployment.apps/source-ip-app created
```

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](#), which is the default since Kubernetes 1.2. Kube-proxy exposes its mode through a `proxyMode` endpoint:

```
kubectl get nodes
```

The output is similar to this:

```
NAME                          STATUS    ROLES     AGE
VERSION
kubernetes-node-6jst    Ready       <none>    2h        v1.13.0
kubernetes-node-cx31    Ready       <none>    2h        v1.13.0
kubernetes-node-jj1t    Ready       <none>    2h        v1.13.0
```

Get the proxy mode on one of the node

```
kubernetes-node-6jst $ curl localhost:10249/proxyMode
```

The output is:

```
iptables
```

You can test source IP preservation by creating a Service over the source IP app:

```
kubectl expose deployment source-ip-app --name=clusterip --
port=80 --target-port=8080
```

The output is:

```
service/clusterip exposed
```

```
kubectl get svc clusterip
```

The output is similar to:

```
NAME            TYPE            CLUSTER-IP      EXTERNAL-IP
PORT(S)    AGE
clusterip     ClusterIP    10.0.170.92    <none>            80/
TCP       51s
```

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

```
kubectl run busybox -it --image=busybox --restart=Never --rm
```

The output is similar to this:

```
Waiting for pod default/busybox to be running, status is
Pending, pod ready: false
If you don't see a command prompt, try pressing enter.

# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
3: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1460 qdisc
noqueue
    link/ether 0a:58:0a:f4:03:08 brd ff:ff:ff:ff:ff:ff
    inet 10.244.3.8/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::188a:84ff:feb0:26a5/64 scope link
       valid_lft forever preferred_lft forever

# 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

As of Kubernetes 1.5, packets sent to Services with [Type=NodePort](#) are source NAT'd by default. You can test this by creating a `NodePort` Service:

```
kubectl expose deployment source-ip-app --name=nodeport --port=80 --target-port=8080 --type=NodePort
```

The output is:

```
service/nodeport exposed
```

```
NODEPORT=$(kubectl get -o
jsonpath="{.spec.ports[0].nodePort}" services nodeport)
NODES=$(kubectl get nodes -o
jsonpath='{ $.items[*].status.addresses[?
(@.type=="ExternalIP")].address }')
```

If you're running on a cloudprovider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

```
for node in $NODES; do curl -s $node:$NODEPORT | grep -i
client_address; done
```
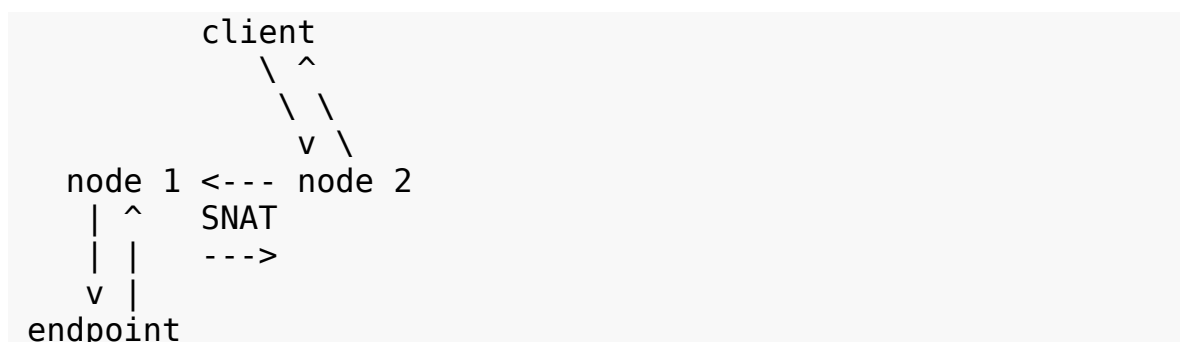
The output is similar to:

```
client_address=10.180.1.1
client_address=10.240.0.5
client_address=10.240.0.3
```

Note that these are not the correct client IPs, they're cluster internal IPs. This is what happens:

- Client sends packet to `node2:nodePort`
- `node2` replaces the source IP address (SNAT) in the packet with its own IP address
- `node2` replaces the destination IP on the packet with the pod IP
- packet is routed to node 1, and then to the endpoint
- the pod's reply is routed back to node2
- the pod's reply is sent back to the client

Visually:

```
          client
             \ ^
              \ \
               v \
   node 1 <--- node 2
     | ^    SNAT
     | |    --->
     v |
  endpoint
```

To avoid this, Kubernetes has a feature to preserve the client source IP [(check here for feature availability)](#). Setting `service.spec.externalTrafficPolicy` to the value `Local` will only proxy requests to local endpoints, never forwarding traffic to other nodes and thereby preserving the original source IP address. If there are no local endpoints, packets sent to the node are dropped, so you can rely on the correct source-ip in any packet processing rules you might apply a packet that make it through to the endpoint.

Set the `service.spec.externalTrafficPolicy` field as follows:

```
kubectl patch svc nodeport -p '{"spec":
{"externalTrafficPolicy":"Local"}}'
```

The output is:

```
service/nodeport patched
```

Now, re-run the test:

```
for node in $NODES; do curl --connect-timeout 1 -s $node:
$NODEPORT | grep -i client_address; done
```

The output is:

```
client_address=104.132.1.79
```

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:

```
       client
      ^ /   \
     / /     \
    / v       X
  node 1     node 2
   ^ |
   | |
   | v
 endpoint
```
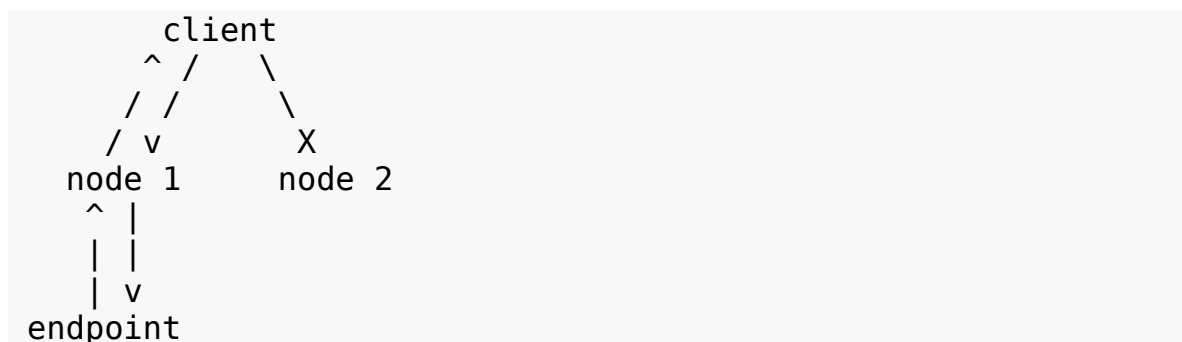
Source IP for Services with Type=LoadBalancer

As of Kubernetes 1.5, 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 loadbalanced 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 loadbalancer

```
kubectl expose deployment source-ip-app --name=loadbalancer
--port=80 --target-port=8080 --type=LoadBalancer
```

The output is:

```
service/loadbalancer exposed
```

Print IPs of the Service:

```
kubectl get svc loadbalancer
```

The output is similar to this:

```
NAME            TYPE           CLUSTER-IP      EXTERNAL-
IP         PORT(S)    AGE
loadbalancer    LoadBalancer    10.0.65.118
104.198.149.140    80/TCP      5m
```
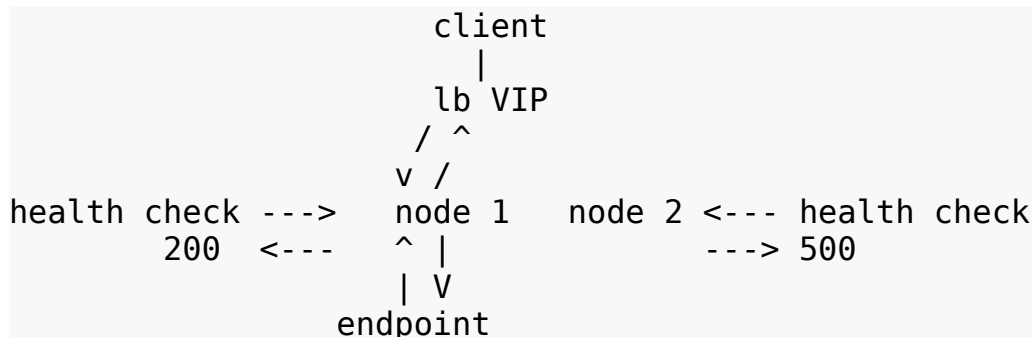
```
curl 104.198.149.140
```

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:

```
                  client
                    |
                  lb VIP
                 / ^
                v /
health check --->   node 1   node 2 <--- health check
       200  <---    ^ |              ---> 500
                    | V
              endpoint
```

You can test this by setting the annotation:

```
kubectl patch svc loadbalancer -p '{"spec":
{"externalTrafficPolicy":"Local"}}'
```

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

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

The output is similar to this:

```
  healthCheckNodePort: 32122
```

The `service.spec.healthCheckNodePort` field points to a port on every node serving the health check at `/healthz`. You can test this:

```
kubectl get pod -o wide -l run=source-ip-app
```

The output is similar to this:

```
NAME                            READY      STATUS
RESTARTS   AGE        IP               NODE
source-ip-app-826191075-qehz4   1/1        Running
0          20h        10.180.1.136    kubernetes-node-6jst
```

Curl the `/healthz` endpoint on different nodes.

```
kubernetes-node-6jst $ curl localhost:32122/healthz
```

The output is similar to this:

```
1 Service Endpoints found
```

```
kubernetes-node-jj1t $ curl localhost:32122/healthz
```

The output is similar to this:

```
No Service Endpoints Found
```

A service controller running on the master is responsible for allocating the cloud loadbalancer, and when it does so, it 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 curl the lb ip:

```
curl 104.198.149.140
```

The output is similar to this:

```
CLIENT VALUES:
client_address=104.132.1.79
...
```

Cross platform support

As of Kubernetes 1.5, support for source IP preservation through Services with Type=LoadBalancer is only implemented in a subset of cloudproviders (GCP and Azure). The cloudprovider 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.

Loadbalancers 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 [X-FORWARDED-FOR](#) header, or the [proxy protocol](#). Loadbalancers in the second category can leverage the feature described above by simply 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 run=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](#)
- Learn more about [loadbalancing](#)

Feedback

Was this page helpful?

Yes No

Thanks for the feedback. If you have a specific, answerable question about how to use Kubernetes, ask it on [Stack Overflow](#). Open an issue in the GitHub repo if you want to [report a problem](#) or [suggest an improvement](#).