Edit This Page

Tutorials

- – Basics
  – Configuration
  – Stateless Applications
  – Stateful Applications
  – CI/CD Pipeline
  – Clusters
  – Services
  – What's next

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

Basics

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

Create an Issue Edit this Page

Edit This Page

Hello Minikube

The goal of this tutorial is for you to turn a simple Hello World Node.js app into an application running on Kubernetes. The tutorial shows you how to take

code that you have developed on your machine, turn it into a Docker container image and then run that image on Minikube. Minikube provides a simple way of running Kubernetes on your local machine for free.

Objectives

- Run a hello world Node.js application.
- Deploy the application to Minikube.
- View application logs.
- Update the application image.

Before you begin

- For OS X, you need Homebrew to install the `xhyve` driver.

  Note: If you see the following Homebrew error when you run `brew update` after you update your computer to MacOS 10.13:

  ```
  Error: /usr/local is not writable. You should change the ownership
  and permissions of /usr/local back to your user account:
  sudo chown -R $(whoami) /usr/local
  ```

  You can resolve the issue by reinstalling Homebrew:

  ```
  /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/mas
  ```

- NodeJS is required to run the sample application.
- Install Docker. On OS X, we recommend Docker for Mac.

Create a Minikube cluster

This tutorial uses Minikube to create a local cluster. This tutorial also assumes you are using Docker for Mac on OS X. If you are on a different platform

like Linux, or using VirtualBox instead of Docker for Mac, the instructions to install Minikube may be slightly different. For general Minikube installation instructions, see the Minikube installation guide.

Use `curl` to download and install the latest Minikube release:

```
curl -Lo minikube https://storage.googleapis.com/minikube/releases/latest/minikube-darwin-am
  chmod +x minikube && \
  sudo mv minikube /usr/local/bin/
```

Use Homebrew to install the xhyve driver and set its permissions:

```
brew install docker-machine-driver-xhyve
sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-dr
sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xh
```

Use Homebrew to download the `kubectl` command-line tool, which you can use to interact with Kubernetes clusters:

```
brew install kubectl
```

Determine whether you can access sites like https://cloud.google.com/container-registry/ directly without a proxy, by opening a new terminal and using

```
curl --proxy "" https://cloud.google.com/container-registry/
```

Make sure that the Docker daemon is started. You can determine if docker is running by using a command such as:

```
docker images
```

If NO proxy is required, start the Minikube cluster:

```
minikube start --vm-driver=xhyve
```

If a proxy server is required, use the following method to start Minikube cluster with proxy setting:

```
minikube start --vm-driver=xhyve --docker-env HTTP_PROXY=http://your-http-proxy-host:your-ht
```

The `--vm-driver=xhyve` flag specifies that you are using Docker for Mac. The default VM driver is VirtualBox.

Note if `minikube start --vm-driver=xhyve` is unsuccessful due to the error:

```
Error creating machine: Error in driver during machine creation: Could not convert the UUID
```

Then the following may resolve the `minikube start --vm-driver=xhyve` issue:

```
rm -rf ~/.minikube
sudo chown root:wheel $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-dr
sudo chmod u+s $(brew --prefix)/opt/docker-machine-driver-xhyve/bin/docker-machine-driver-xh
```

Now set the Minikube context. The context is what determines which cluster `kubectl` is interacting with. You can see all your available contexts in the `~/.kube/config` file.

```
kubectl config use-context minikube
```

Verify that `kubectl` is configured to communicate with your cluster:

```
kubectl cluster-info
```

Open the Kubernetes dashboard in a browser:

```
minikube dashboard
```

Create your Node.js application

The next step is to write the application. Save this code in a folder named `hellonode` with the filename `server.js`:

server.js docs/tutorials

```
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);
```

Run your application:

```
node server.js
```

You should be able to see your "Hello World!" message at http://localhost:8080/.

Stop the running Node.js server by pressing **Ctrl-C**.

The next step is to package your application in a Docker container.

Create a Docker container image

Create a file, also in the `hellonode` folder, named `Dockerfile`. A Dockerfile describes the image that you want to build. You can build a Docker container image by extending an existing image. The image in this tutorial extends an existing Node.js image.

```
Dockerfile docs/tutorials
```

```
FROM node:6.9.2
EXPOSE 8080
COPY server.js .
CMD node server.js
```

This recipe for the Docker image starts from the official Node.js LTS image found in the Docker registry, exposes port 8080, copies your `server.js` file to the image and starts the Node.js server.

Because this tutorial uses Minikube, instead of pushing your Docker image to a registry, you can simply build the image using the same Docker host as the Minikube VM, so that the images are automatically present. To do so, make sure you are using the Minikube Docker daemon:

```
eval $(minikube docker-env)
```

Note: Later, when you no longer wish to use the Minikube host, you can undo this change by running `eval $(minikube docker-env -u)`.

Build your Docker image, using the Minikube Docker daemon (mind the trailing dot):

```
docker build -t hello-node:v1 .
```

Now the Minikube VM can run the image you built.

Create a Deployment

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

Use the `kubectl run` command to create a Deployment that manages a Pod. The Pod runs a Container based on your `hello-node:v1` Docker image:

```
kubectl run hello-node --image=hello-node:v1 --port=8080
```

View the Deployment:

```
kubectl get deployments
```

Output:

```
NAME         DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
hello-node   1         1         1            1           3m
```

View the Pod:

```
kubectl get pods
```

Output:

```
NAME                          READY     STATUS    RESTARTS   AGE
hello-node-714049816-ztzrb    1/1       Running   0          6m
```

View cluster events:

```
kubectl get events
```

View the `kubectl` configuration:

```
kubectl config view
```

For more information about `kubectl`commands, see the kubectl overview.

Create a Service

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

From your development machine, you can expose the Pod to the public internet using the `kubectl expose` command:

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

View the Service you just created:

```
kubectl get services
```

Output:

```
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
hello-node   10.0.0.71    <pending>     8080/TCP   6m
kubernetes   10.0.0.1     <none>        443/TCP    14d
```

The `--type=LoadBalancer` flag indicates that you want to expose your Service outside of the cluster. On cloud providers that support load balancers, an external IP address would be provisioned to access the Service. On Minikube, the `LoadBalancer` type makes the Service accessible through the `minikube service` command.

```
minikube service hello-node
```

This automatically opens up a browser window using a local IP address that serves your app and shows the "Hello World" message.

Assuming you've sent requests to your new web service using the browser or curl, you should now be able to see some logs:

```
kubectl logs <POD-NAME>
```

Update your app

Edit your `server.js` file to return a new message:

```
response.end('Hello World Again!');
```

Build a new version of your image (mind the trailing dot):

```
docker build -t hello-node:v2 .
```

Update the image of your Deployment:

```
kubectl set image deployment/hello-node hello-node=hello-node:v2
```

Run your app again to view the new message:

```
minikube service hello-node
```

Enable addons

Minikube has a set of built-in addons that can be enabled, disabled and opened in the local Kubernetes environment.

First list the currently supported addons:

```
minikube addons list
```

Output:

```
- storage-provisioner: enabled
- kube-dns: enabled
- registry: disabled
- registry-creds: disabled
- addon-manager: enabled
- dashboard: disabled
- default-storageclass: enabled
- coredns: disabled
- heapster: disabled
- efk: disabled
- ingress: disabled
```

Minikube must be running for these commands to take effect. To enable `heapster` addon, for example:

```
minikube addons enable heapster
```

Output:

```
heapster was successfully enabled
```

View the Pod and Service you just created:

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

Output:

```
NAME                                READY     STATUS    RESTARTS   AGE
po/heapster-zbwzv                   1/1       Running   0          2m
po/influxdb-grafana-gtht9           2/2       Running   0          2m

NAME                          TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)             AGE
svc/heapster                  NodePort    10.0.0.52    <none>        80:31655/TCP        2m
svc/monitoring-grafana        NodePort    10.0.0.33    <none>        80:30002/TCP        2m
svc/monitoring-influxdb       ClusterIP   10.0.0.43    <none>        8083/TCP,8086/TCP   2m
```

Open the endpoint to interacting with heapster in a browser:

```
minikube addons open heapster
```

Output:

```
Opening kubernetes service kube-system/monitoring-grafana in default browser...
```

Clean up

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

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

Optionally, force removal of the Docker images created:

```
docker rmi hello-node:v1 hello-node:v2 -f
```

Optionally, stop the Minikube VM:

```
minikube stop
eval $(minikube docker-env -u)
```

Optionally, delete the Minikube VM:

```
minikube delete
```

What's next

- Learn more about Deployment objects.

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

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Updating Your App

To interact with the Terminal, please use the desktop/tablet version

Create an Issue Edit this Page

Edit This Page

Overview

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

Start the tutorial›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Creating a Cluster

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 2›

Create an Issue Edit this Page

Edit This Page

Using Minikube to Create a Cluster

Objectives

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

Kubernetes Clusters

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

A Kubernetes cluster consists of two types of resources:

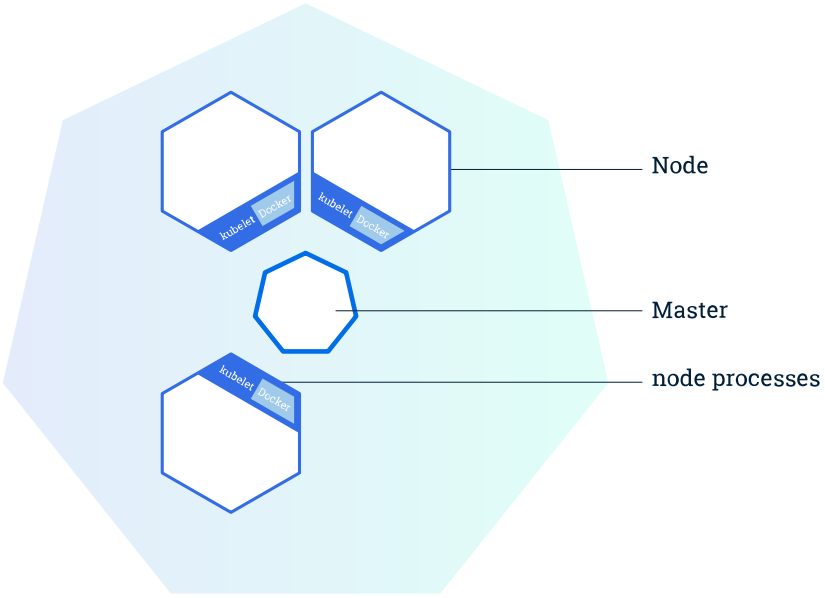
- The **Master** coordinates the cluster
- **Nodes** are the workers that run applications

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 ›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Creating a Cluster

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 2›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Deploying an App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 3›

Create an Issue Edit this Page

Edit This Page

Using kubectl to Create a Deployment

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 it. **This provides a self-healing mechanism to address machine failure or maintenance.**

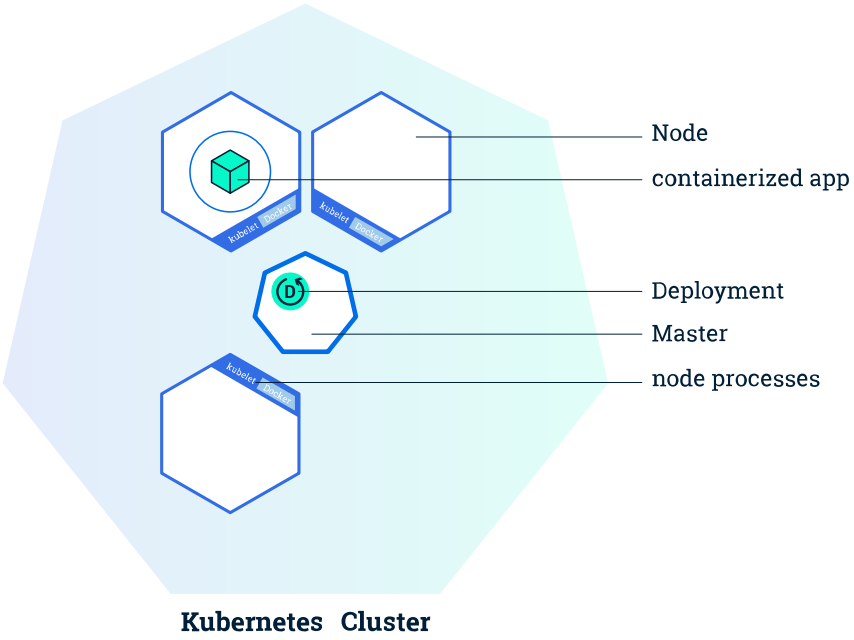
In a pre-orchestration world, installation scripts would often be used to start applications, but they did not allow recovery from machine failure. By both creating your application instances and keeping them running across Nodes, Kubernetes Deployments provide a fundamentally different approach to application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

Summary:

- Deployments
- Kubectl

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

Deploying your first app on Kubernetes



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 our first Deployment, we'll use a Node.js application packaged in a Docker container. The source code and the Dockerfile are available in the GitHub repository for the Kubernetes Basics.

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

Start Interactive Tutorial ›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Deploying an App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 3›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Exploring Your App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 4›

Create an Issue Edit this Page

Edit This Page

Viewing Pods and Nodes

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.

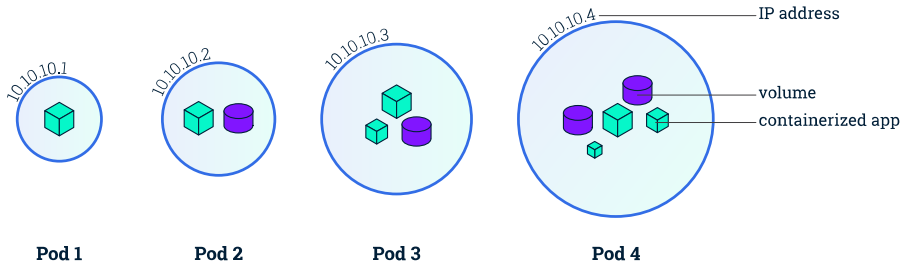
Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly). Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to 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



Nodes

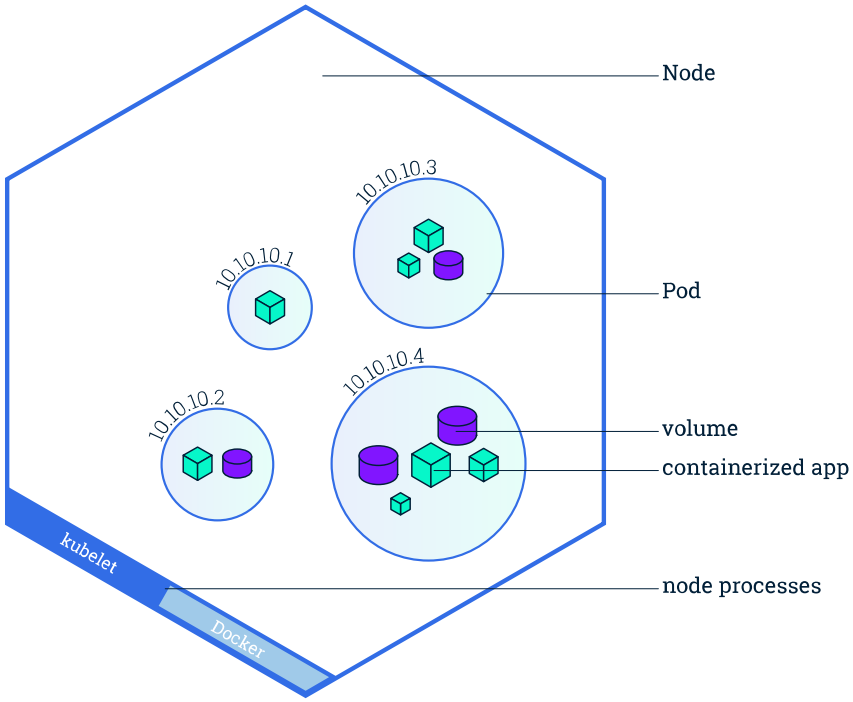
A Pod always runs on a **Node**. A Node is a worker machine in Kubernetes and may be either a virtual or a physical machine, depending on the cluster. Each Node is managed by the Master. A Node can have multiple pods, and the Kubernetes master automatically handles scheduling the pods across the Nodes 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 ›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Exploring Your App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 4›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Exposing Your App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 5›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Exposing Your App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 5›

Create an Issue Edit this Page

Edit This Page

Running Multiple Instances of Your App

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

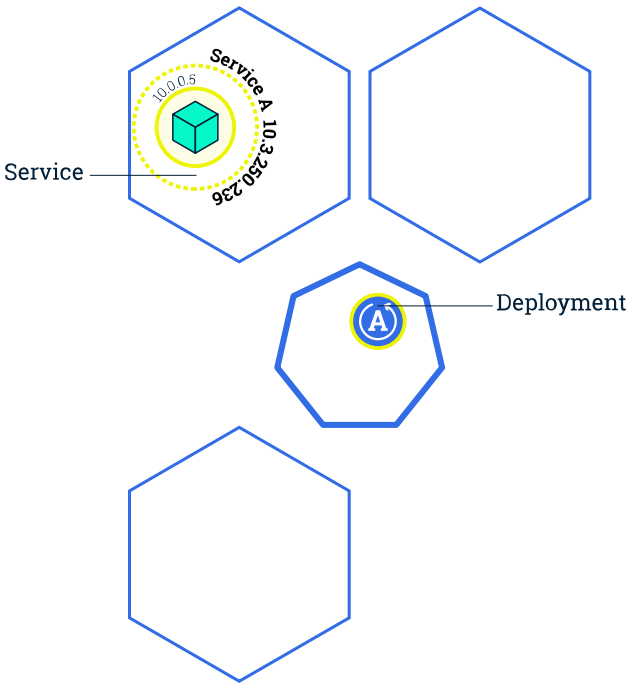
Summary:

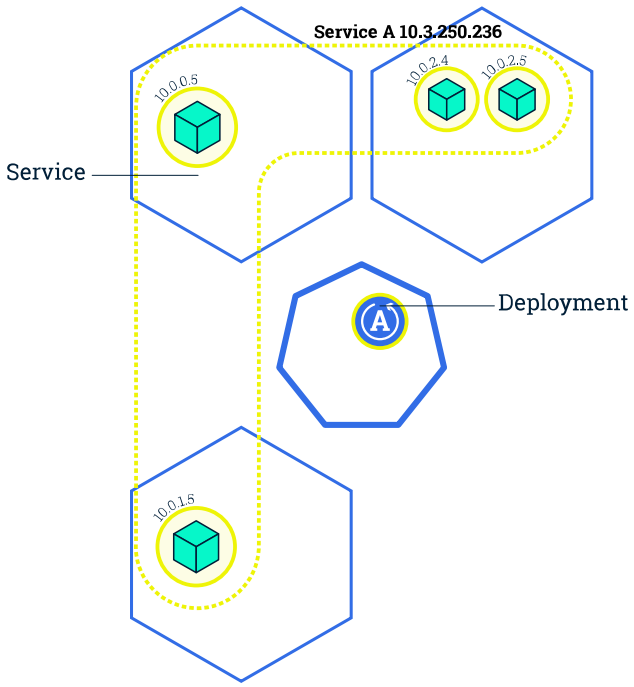
- Scaling a Deployment

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



Previous Next

Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling in will reduce 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 ›

Create an Issue Edit this Page

Edit This Page

Running Multiple Instances of Your App

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

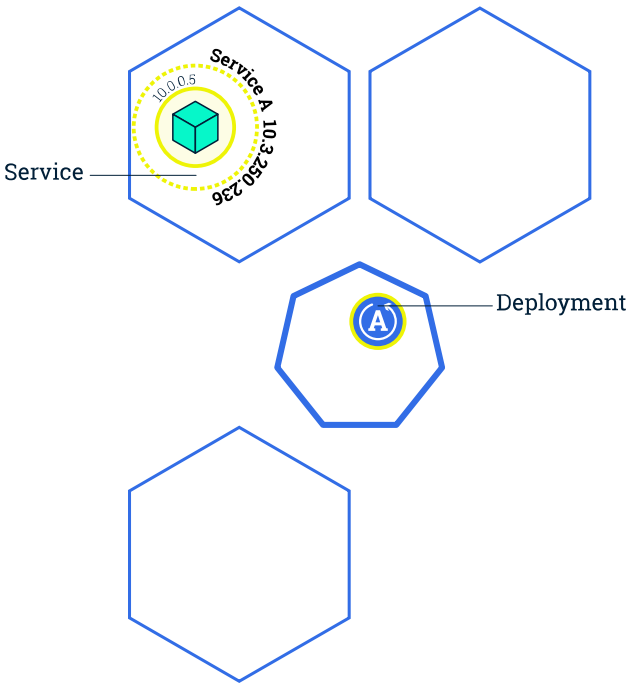
Summary:

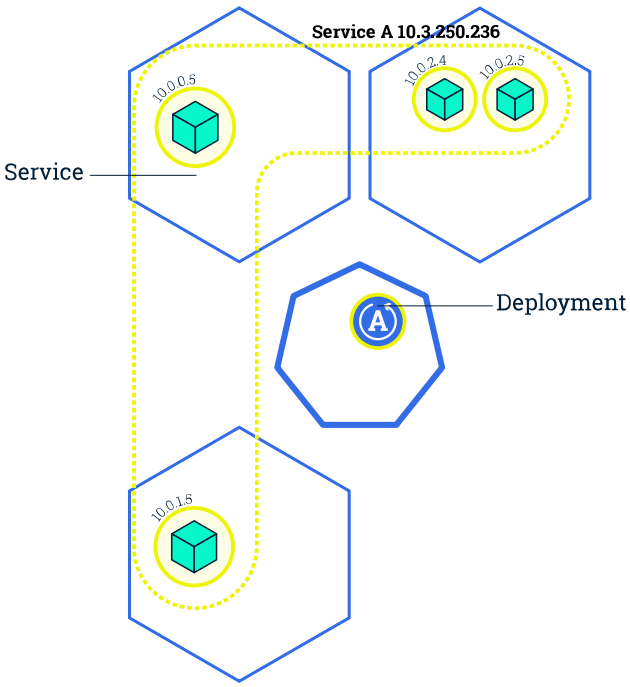
- Scaling a Deployment

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



Previous Next

Scaling out a Deployment will ensure new Pods are created and scheduled to Nodes with available resources. Scaling in will reduce 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 ›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Scaling Your App

To interact with the Terminal, please use the desktop/tablet version

Continue to Module 6›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Updating Your App

To interact with the Terminal, please use the desktop/tablet version

Create an Issue Edit this Page

Edit This Page

Performing a Rolling Update

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 previous (stable) version.

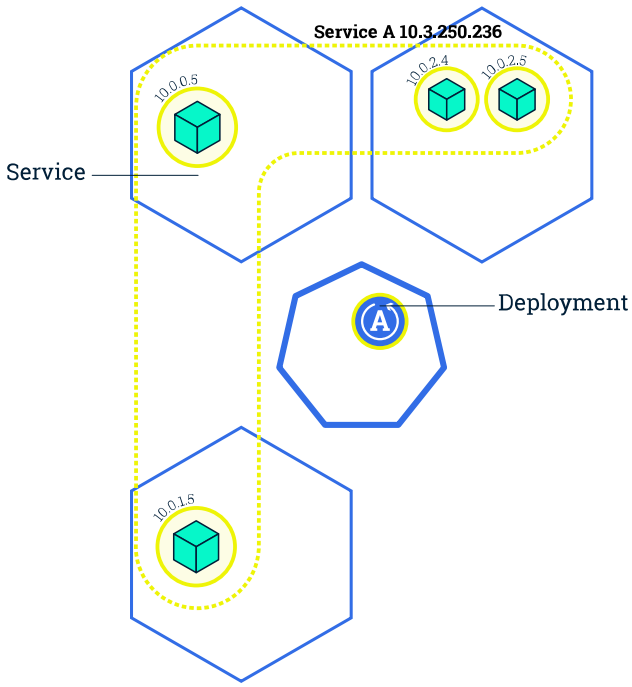
Summary:

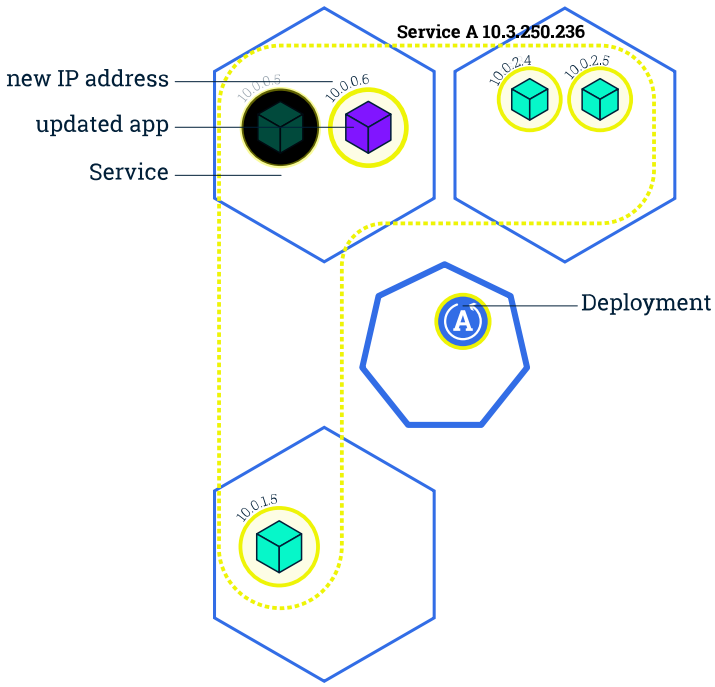
- Updating an app

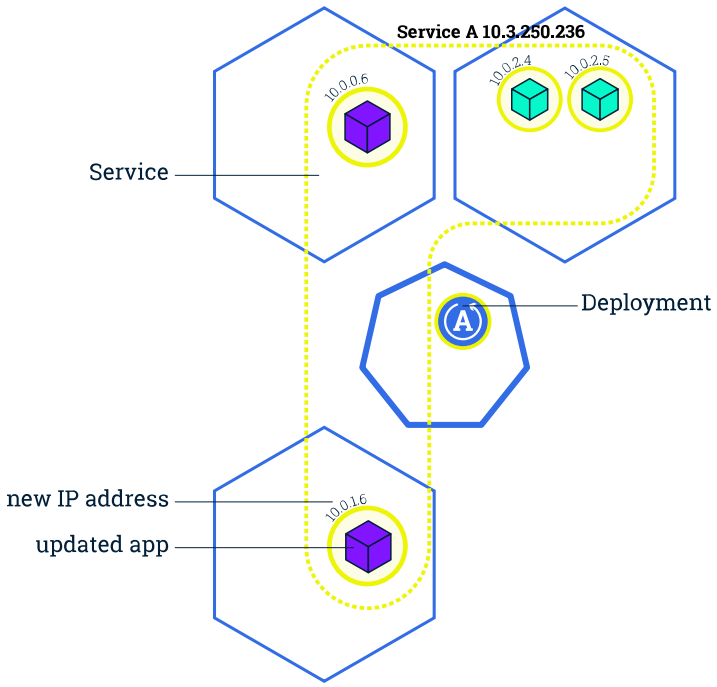
Rolling updates allow Deployments' update to take place with zero downtime by incrementally updating Pods instances with new ones.

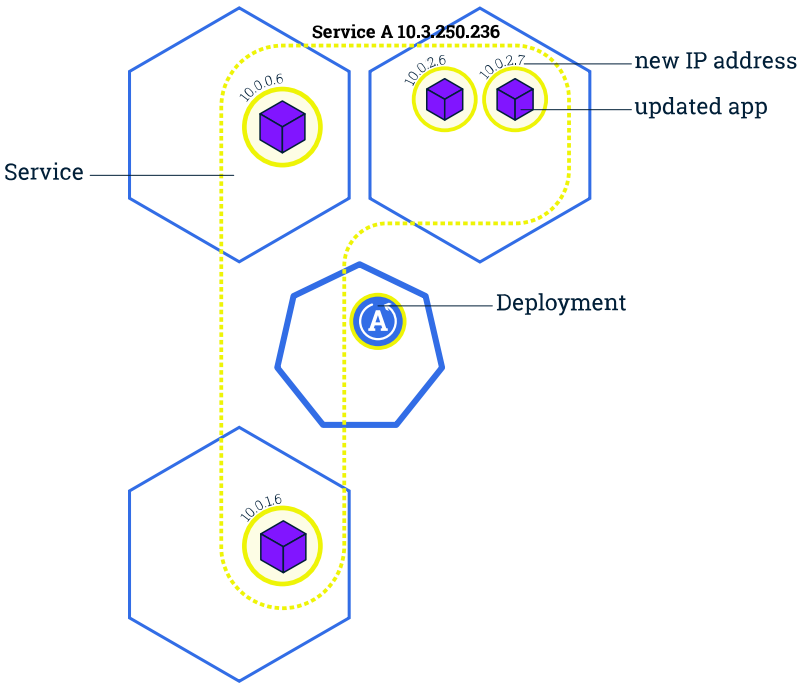
Rolling updates overview

1.
2.
3.
4.


Service A 10.3.250.236
10.0.0.5
10.0.2.4
10.0.2.5
Service
Deployment
10.0.1.5


Service A 10.3.250.236
new IP address
updated app
Service
10.0.0.6
10.0.2.4
10.0.2.5
Deployment
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
updated app
10.0.1.6



Previous Next

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 ›

Create an Issue Edit this Page

Edit This Page

Interactive Tutorial - Updating Your App

To interact with the Terminal, please use the desktop/tablet version

Create an Issue Edit this Page

Edit This Page

Overview of Kubernetes Online Training

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

- Scalable Microservices with Kubernetes (Udacity)
- Introduction to Kubernetes (edX)

Create an Issue Edit this Page

Edit This Page

Overview of Kubernetes Online Training

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

- Scalable Microservices with Kubernetes (Udacity)
- Introduction to Kubernetes (edX)

Create an Issue Edit this Page

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

- Create a ConfigMap.
- Create a pod specification using the ConfigMap.
- Create the pod.
- 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`.

- 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.

1. Create a ConfigMap from the `docs/tutorials/configuration/configmap/redis/redis-config` file:

```
kubectl create configmap example-redis-config --from-file=https://k8s.io/docs/tutorials/c
kubectl get configmap example-redis-config -o yaml
```

```
apiVersion: v1
data:
  redis-config: |
```

```
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-03-30T18:14:41Z
  name: example-redis-config
  namespace: default
  resourceVersion: "24686"
  selfLink: /api/v1/namespaces/default/configmaps/example-redis-config
  uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002
```

1. Create a pod specification that uses the config data stored in the ConfigMap:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
  containers:
  - name: redis
    image: kubernetes/redis:v1
    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
```

1. Create the pod:

```
kubectl create -f https://k8s.io/tutorials/configuration/configmap/redis/redis-pod.yaml
```

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.

1. 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"
```

What's next

- Learn more about ConfigMaps.

Create an Issue Edit this Page

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

- Create a ConfigMap.
- Create a pod specification using the ConfigMap.
- Create the pod.
- 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`.

- 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.

1. Create a ConfigMap from the `docs/tutorials/configuration/configmap/redis/redis-config` file:

```
kubectl create configmap example-redis-config --from-file=https://k8s.io/docs/tutorials/c
kubectl get configmap example-redis-config -o yaml
```

```
apiVersion: v1
data:
  redis-config: |
    maxmemory 2mb
    maxmemory-policy allkeys-lru
kind: ConfigMap
metadata:
  creationTimestamp: 2016-03-30T18:14:41Z
  name: example-redis-config
  namespace: default
  resourceVersion: "24686"
  selfLink: /api/v1/namespaces/default/configmaps/example-redis-config
  uid: 460a2b6e-f6a3-11e5-8ae5-42010af00002
```

1. Create a pod specification that uses the config data stored in the ConfigMap:

```
apiVersion: v1
kind: Pod
metadata:
  name: redis
spec:
```

```
containers:
- name: redis
  image: kubernetes/redis:v1
  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
```

1. Create the pod:

```
kubectl create -f https://k8s.io/tutorials/configuration/configmap/redis/redis-pod.yaml
```

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.

1. 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"
```

What's next

- Learn more about ConfigMaps.

Create an Issue Edit this Page

Edit This Page

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`.

Download the following configuration files:

1. redis-master-deployment.yaml
2. redis-master-service.yaml
3. redis-slave-deployment.yaml
4. redis-slave-service.yaml
5. frontend-deployment.yaml
6. frontend-service.yaml

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.

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 redis-master-deployment.yaml`

guestbook/redis-master-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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. 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
   ```

2. 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.

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

`guestbook/redis-master-service.yaml`

`docs/tutorials/stateless-application/guestbook`

```
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
```

> **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.

1. 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           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1     <none>        443/TCP    1m
redis-master   10.0.0.151   <none>        6379/TCP   8s
```

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.

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

```
kubectl apply -f redis-slave-deployment.yaml
```

guestbook/redis-slave-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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:v1
        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. 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.

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

   ```
   kubectl apply -f redis-slave-service.yaml
   ```

`guestbook/redis-slave-service.yaml`

`docs/tutorials/stateless-application/guestbook`

```
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. 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           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1     <none>        443/TCP    2m
redis-master   10.0.0.151   <none>        6379/TCP   1m
redis-slave    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

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

```
kubectl apply -f frontend-deployment.yaml
```

guestbook/frontend-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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. 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`.

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

```
kubectl apply -f frontend-service.yaml
```

guestbook/frontend-service.yaml

docs/tutorials/stateless-application/guestbook

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

```
kubectl get services
```

The response should be similar to this:

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

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

1. 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
```

1. 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
```

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

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

1. 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 "redis-master" deleted
deployment "redis-slave" deleted
service "redis-master" deleted
service "redis-slave" deleted
deployment "frontend" deleted
service "frontend" deleted
```

1. 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

- 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

Create an Issue Edit this Page

Edit This Page

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

- 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:

   ```
   kubectl run hello-world --replicas=5 --labels="run=load-balancer-example" --image=gcr.i
   ```

   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.

2. Display information about the Deployment:

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

3. Display information about your ReplicaSet objects:

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

4. Create a Service object that exposes the deployment:

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

5. Display information about the Service:

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

   The output is similar to this:

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

   Note: If the external IP address is shown as <pending>, wait for a minute and enter the same command again.

6. Display detailed information about the Service:

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

   The output is similar to this:

   ```
   Name:           my-service
   Namespace:      default
   Labels:         run=load-balancer-example
   Annotations:    <none>
   Selector:       run=load-balancer-example
   Type:           LoadBalancer
   IP:             10.3.245.137
   LoadBalancer Ingress:   104.198.205.71
   Port:           <unset> 8080/TCP
   NodePort:       <unset> 32377/TCP
   Endpoints:      10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more...
   Session Affinity:   None
   Events:         <none>
   ```

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

7. In the preceding output, you can see that the service has several endpoints: 10.0.0.6:8080,10.0.1.6:8080,10.0.1.7:8080 + 2 more. These are internal ad-

dresses 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
```

8. Use the external IP address (`LoadBalancer Ingress`) to access the Hello World application:

   `curl http://<external-ip>:<port>`

   where `<external-ip>` is the external IP address (`LoadBalancer Ingress`) of your Service, and `<port>` is the value of `Port` in your Service description. If you are using minikube, typing `minikube service my-service` will automatically open the Hello World application in a browser.

   The response to a successful request is a hello message:

   `Hello Kubernetes!`

Cleaning up

To delete the Service, enter this command:

```
kubectl delete services my-service
```

To delete the Deployment, the ReplicaSet, and the Pods that are running the Hello World application, enter this command:

```
kubectl delete deployment hello-world
```

What's next

Learn more about connecting applications with services.

Create an Issue Edit this Page

Edit This Page

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`.

Download the following configuration files:

1. redis-master-deployment.yaml
2. redis-master-service.yaml
3. redis-slave-deployment.yaml
4. redis-slave-service.yaml

5. frontend-deployment.yaml
6. frontend-service.yaml

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.

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 redis-master-deployment.yaml`

guestbook/redis-master-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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. 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
   ```

2. 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.

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

`guestbook/redis-master-service.yaml`

`docs/tutorials/stateless-application/guestbook`

```
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
```

> **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.

1. 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           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1     <none>        443/TCP    1m
redis-master   10.0.0.151   <none>        6379/TCP   8s
```

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.

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

```
kubectl apply -f redis-slave-deployment.yaml
```

guestbook/redis-slave-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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:v1
        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. 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.

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

```
kubectl apply -f redis-slave-service.yaml
```

`guestbook/redis-slave-service.yaml`

`docs/tutorials/stateless-application/guestbook`

```
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. 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           CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
kubernetes     10.0.0.1     <none>        443/TCP    2m
redis-master   10.0.0.151   <none>        6379/TCP   1m
redis-slave    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

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

```
kubectl apply -f frontend-deployment.yaml
```

guestbook/frontend-deployment.yaml

docs/tutorials/stateless-application/guestbook

```
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. 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`.

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

```
kubectl apply -f frontend-service.yaml
```

guestbook/frontend-service.yaml

docs/tutorials/stateless-application/guestbook

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

```
kubectl get services
```

The response should be similar to this:

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

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

1. 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
```

1. 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
```

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

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

1. 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 "redis-master" deleted
deployment "redis-slave" deleted
service "redis-master" deleted
service "redis-slave" deleted
deployment "frontend" deleted
service "frontend" deleted
```

1. 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

- 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

Create an Issue Edit this Page

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

- Create PersistentVolumeClaims and PersistentVolumes
- Create a Secret
- Deploy MySQL
- Deploy WordPress
- 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`.

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 Secret for MySQL Password

A Secret is an object that stores a piece of sensitive data like a password or key. The manifest files are already configured to use a Secret, but you have to create your own Secret.

1. Create the Secret object from the following command. You will need to replace `YOUR_PASSWORD` with the password you want to use.

   ```
   kubectl create secret generic mysql-pass --from-literal=password=YOUR_PASSWORD
   ```

2. 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            Opaque                1         42s
   ```

   Note: To protect the Secret from exposure, neither `get` nor `describe` show its contents.

Deploy MySQL

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.

mysql-wordpress-persistent-volume/mysql-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

```
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
```

```
mysql-wordpress-persistent-volume/mysql-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume
```

1. Deploy MySQL from the `mysql-deployment.yaml` file:

   ```
   kubectl create -f mysql-deployment.yaml
   ```

2. Verify that a PersistentVolume got dynamically provisioned. Note that it can It can take up to a few minutes for the PVs to be provisioned and bound.

   ```
   kubectl get pvc
   ```

   The response should be like this:

   ```
   NAME             STATUS    VOLUME                                     CAPACITY ACCESS
   mysql-pv-claim   Bound     pvc-91e44fbf-d477-11e7-ac6a-42010a800002   20Gi     RWO
   ```

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
   ```

Deploy WordPress

The following manifest describes a single-instance WordPress Deployment and Service. It uses many of the same features like a PVC for persistent storage and a Secret for the password. But it also uses a different setting: `type: LoadBalancer`. This setting exposes WordPress to traffic from outside of the cluster.

mysql-wordpress-persistent-volume/wordpress-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

```
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
```

mysql-wordpress-persistent-volume/wordpress-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

1. Create a WordPress Service and Deployment from the `wordpress-deployment.yaml` file:

   ```
   kubectl create -f wordpress-deployment.yaml
   ```

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
   wp-pv-claim      Bound     pvc-e69d834d-d477-11e7-ac6a-42010a800002   20Gi     RWO
   ```

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

   ```
   kubectl get services wordpress
   ```

   The response should be like this:

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

   Note: Minikube can only expose Services through `NodePort`.

   ```
   The EXTERNAL-IP is always <pending>.
   ```

4. Run the following command to get the IP Address for the WordPress Service:

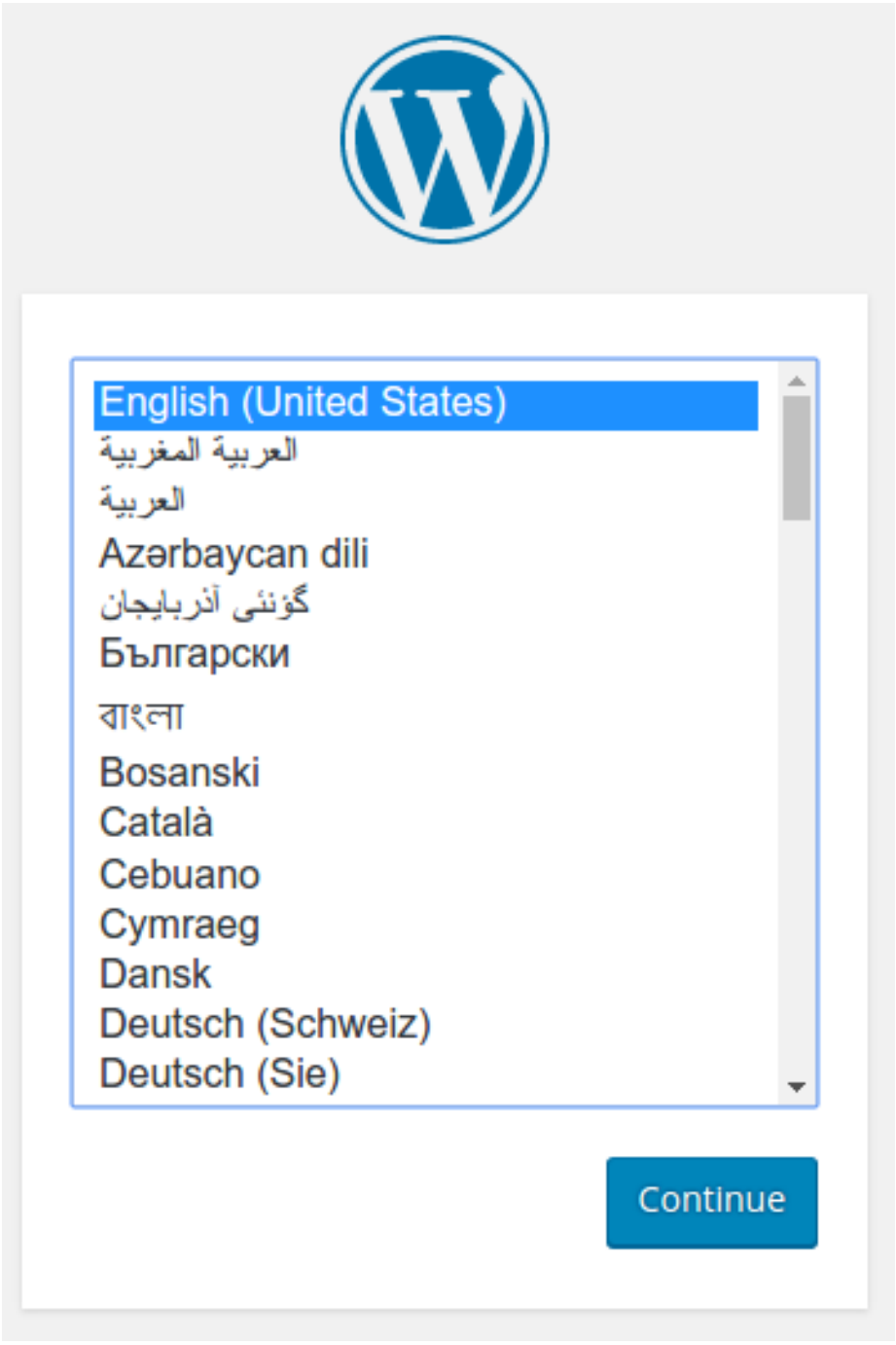
   ```
   minikube service wordpress --url
   ```

   The response should be like this:

   ```
   http://1.2.3.4:32406
   ```

5. Copy the IP address, and load the page in your browser to view your site.

You should see the WordPress set up page similar to the following screenshot.



Warning: Do not leave your WordPress installation on this page. If another user finds it, they can set up a website on your instance

and use it to serve malicious content.

Either install WordPress by creating a username and password or delete your instance.

Cleaning up

1. Run the following command to delete your Secret:

   ```
   kubectl delete secret mysql-pass
   ```

2. Run the following commands to delete all Deployments and Services:

   ```
   kubectl delete deployment -l app=wordpress
   kubectl delete service -l app=wordpress
   ```

3. Run the following commands to delete the PersistentVolumeClaims. The dynamically provisioned PersistentVolumes will be automatically deleted.

   ```
   kubectl delete pvc -l app=wordpress
   ```

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

Create an Issue Edit this Page

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`.

web.yaml docs/tutorials/stateful-application

```
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
```

`web.yaml docs/tutorials/stateful-application`

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 create` to create the Headless Service and StatefulSet defined in `web.yaml`.

```
kubectl create -f web.yaml
service "nginx" created
statefulset "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      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
nginx     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 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: 10.244.1.6

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 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   AC
www-web-0   Bound     pvc-15c268c7-b507-11e6-932f-42010a800002   1Gi        RWO           48
www-web-1   Bound     pvc-15c79307-b507-11e6-932f-42010a800002   1Gi        RWO           48
```

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/inde

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

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

Updating StatefulSets

In Kubernetes 1.7 and later, the StatefulSet controller supports automated updates. The strategy used is determined by the `spec.updateStrategy` field of the StatefulSet API Object. This feature can be used to upgrade the container images, resource requests and/or limits, labels, and annotations of the Pods in a StatefulSet. There are two valid update strategies, `RollingUpdate` and `OnDelete`.

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 "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/s
statefulset "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, $c := .spec.containers}}{{$
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","rolling
statefulset "web" patched
```

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

```
kubectl patch statefulset web --type='json' -p='[{"op": "replace", "path": "/spec/template/s
statefulset "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","rolling
statefulset "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","rolling
statefulset "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, $c := .spec.containers}}{{$
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 "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 create -f web.yaml
statefulset "web" created
Error from server (AlreadyExists): error when creating "web.yaml": services "nginx" already
```

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 "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 create -f web.yaml
service "nginx" created
statefulset "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.

webp.yaml docs/tutorials/stateful-application

```
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
```

webp.yaml docs/tutorials/stateful-application

Download the example above, and save it to a file named `webp.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 create -f webp.yaml
service "nginx" created
statefulset "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 "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.

Create an Issue Edit this Page

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

- Create PersistentVolumeClaims and PersistentVolumes
- Create a Secret
- Deploy MySQL
- Deploy WordPress
- 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`.

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 Secret for MySQL Password

A Secret is an object that stores a piece of sensitive data like a password or key. The manifest files are already configured to use a Secret, but you have to create your own Secret.

1. Create the Secret object from the following command. You will need to replace `YOUR_PASSWORD` with the password you want to use.

   ```
   kubectl create secret generic mysql-pass --from-literal=password=YOUR_PASSWORD
   ```

2. 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            Opaque                1         42s
   ```

   Note: To protect the Secret from exposure, neither `get` nor `describe` show its contents.

Deploy MySQL

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.

mysql-wordpress-persistent-volume/mysql-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

```
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
```

```
mysql-wordpress-persistent-volume/mysql-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume
```

1. Deploy MySQL from the `mysql-deployment.yaml` file:

   ```
   kubectl create -f mysql-deployment.yaml
   ```

2. Verify that a PersistentVolume got dynamically provisioned. Note that it can It can take up to a few minutes for the PVs to be provisioned and bound.

   ```
   kubectl get pvc
   ```

   The response should be like this:

   ```
   NAME             STATUS    VOLUME                                     CAPACITY ACCESS
   mysql-pv-claim   Bound     pvc-91e44fbf-d477-11e7-ac6a-42010a800002   20Gi     RWO
   ```

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
   ```

Deploy WordPress

The following manifest describes a single-instance WordPress Deployment and Service. It uses many of the same features like a PVC for persistent storage and a Secret for the password. But it also uses a different setting: `type: LoadBalancer`. This setting exposes WordPress to traffic from outside of the cluster.

mysql-wordpress-persistent-volume/wordpress-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

```
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
```

mysql-wordpress-persistent-volume/wordpress-deployment.yaml
docs/tutorials/stateful-application/mysql-wordpress-persistent-volume

1. Create a WordPress Service and Deployment from the `wordpress-deployment.yaml` file:

   ```
   kubectl create -f wordpress-deployment.yaml
   ```

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
   wp-pv-claim      Bound     pvc-e69d834d-d477-11e7-ac6a-42010a800002   20Gi     RWO
   ```

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

   ```
   kubectl get services wordpress
   ```

   The response should be like this:

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

   Note: Minikube can only expose Services through `NodePort`.

   ```
   The EXTERNAL-IP is always <pending>.
   ```

4. Run the following command to get the IP Address for the WordPress Service:

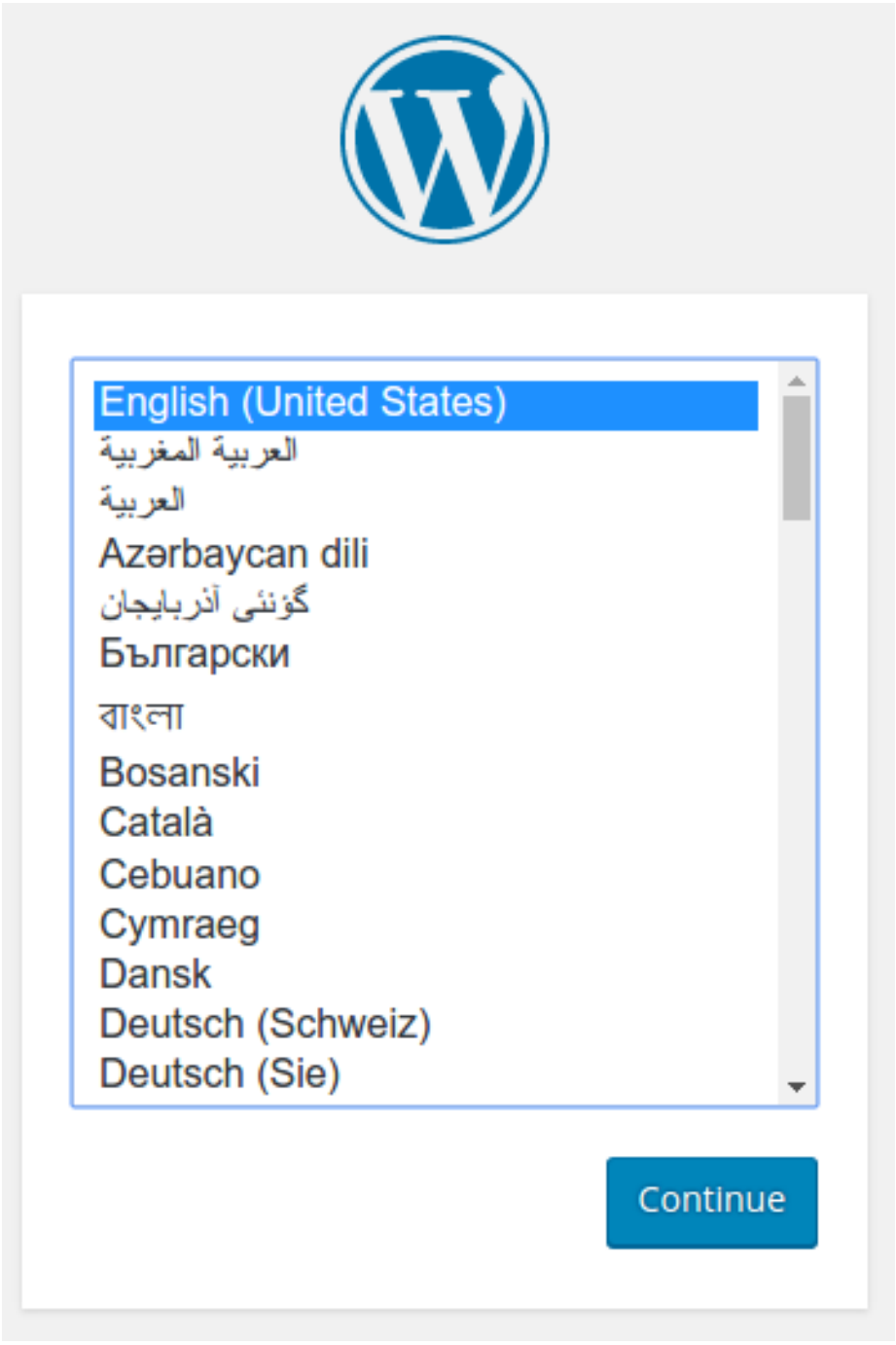
   ```
   minikube service wordpress --url
   ```

   The response should be like this:

   ```
   http://1.2.3.4:32406
   ```

5. Copy the IP address, and load the page in your browser to view your site.

You should see the WordPress set up page similar to the following screenshot.



Warning: Do not leave your WordPress installation on this page. If another user finds it, they can set up a website on your instance

and use it to serve malicious content.

Either install WordPress by creating a username and password or delete your instance.

Cleaning up

1. Run the following command to delete your Secret:

   ```
   kubectl delete secret mysql-pass
   ```

2. Run the following commands to delete all Deployments and Services:

   ```
   kubectl delete deployment -l app=wordpress
   kubectl delete service -l app=wordpress
   ```

3. Run the following commands to delete the PersistentVolumeClaims. The dynamically provisioned PersistentVolumes will be automatically deleted.

   ```
   kubectl delete pvc -l app=wordpress
   ```

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

Create an Issue Edit this Page

Edit This Page

Example: Deploying Cassandra with Stateful Sets

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

Deploying stateful distributed applications, like Cassandra, within a clustered environment can be challenging. StatefulSets greatly simplify this process. Please read about StatefulSets for more information about the features used in this tutorial.

Cassandra Docker

The Pods 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 Services.
- 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
- Download cassandra-service.yaml and cassandra-statefulset.yaml
- Have a supported Kubernetes Cluster running

  Note: Please read the getting started guides if you do not already have a cluster.

Additional Minikube Setup Instructions

> **Caution:** Minikube defaults to 1024MB of memory and 1 CPU which results in an insufficient resource errors during this tutorial.

To avoid these errors, run minikube with:

```
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.

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 create -f cassandra-service.yaml

`cassandra/cassandra-service.yaml`

`docs/tutorials/stateful-application/cassandra`

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

Validating (optional)

Get the Cassandra `Service`.

```
kubectl get svc cassandra
```

The response should be

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

If anything else returns, the service was not successfully created. 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.

1. Update the StatefulSet if necessary.
2. Create the Cassandra StatefulSet from the `cassandra-statefulset.yaml` file:

   kubectl create -f cassandra-statefulset.yaml

cassandra/cassandra-statefulset.yaml

docs/tutorials/stateful-application/cassandra

```
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
```

```
cassandra/cassandra-statefulset.yaml

docs/tutorials/stateful-application/cassandra
```

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.

1. 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
```

Note: It can take up to ten minutes for all three Pods to deploy.

```
Once all Pods 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
```

1. Run the Cassandra utility nodetool to display the status of the ring.

   kubectl exec cassandra-0 – nodetool status

The response is:

```
Datacenter: DC1-K8Demo
======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID
UN  172.17.0.5  83.57 KiB  32           74.0%             e2dd09e6-d9d3-477e-96c5-45094c0
UN  172.17.0.4  101.04 KiB  32           58.8%             f89d6835-3a42-4419-92b3-0e62ca
UN  172.17.0.6  84.74 KiB  32           67.1%             a6a1e8c2-3dc5-4417-b1a0-26507af
```

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.

Note: 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
# 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"
 selfLink: /apis/apps/v1/namespaces/default/statefulsets/cassandra
 uid: 7a219483-6185-11e6-a910-42010a8a0fc0
spec:
 replicas: 3
```

1. Change the number of replicas to 4, and then save the manifest.

The StatefulSet now contains 4 Pods.

1. 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 ensures safety first: your data is more valuable than an auto purge of all related StatefulSet resources.

Warning: Depending on the storage class and reclaim policy, deleting the Persistent Volume Claims 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 to delete everything in a `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

Create an Issue Edit this Page

Edit This Page

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
- Tolerating Node Failure
- Surviving Maintenance
- Cleaning up

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 their 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 PodDisruption-Budget, and a StatefulSet.

zookeeper.yaml docs/tutorials/stateful-application

```
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: Parallel
  template:
```

```
zookeeper.yaml docs/tutorials/stateful-application
```

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

```
kubectl apply -f https://k8s.io/docs/tutorials/stateful-application/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 "zk-pdb" created
statefulset "zk" created
```

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

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

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

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

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

Facilitating Leader Election

Because there is no terminating algorithm for electing a leader in an anonymous network, Zab requires explicit membership configuration to perform leader election. Each server in the ensemble needs to have a unique identifier, all servers

need to know the global set of identifiers, and each identifier needs to be associated with a network address.

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

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

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

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

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

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

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

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 "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/docs/tutorials/stateful-application/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
datadir-zk-0   Bound     pvc-bed742cd-bcb1-11e6-994f-42010a800002   20Gi       RWO
datadir-zk-1   Bound     pvc-bedd27d2-bcb1-11e6-994f-42010a800002   20Gi       RWO
datadir-zk-2   Bound     pvc-bee0817e-bcb1-11e6-994f-42010a800002   20Gi       RWO
```

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

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

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

Ensuring Consistent Configuration

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

Get the `zk` StatefulSet.

```
kubectl get sts zk -o yaml
…
command:
```

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

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

Configuring Logging

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

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

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

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

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

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

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

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

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

```
2016-12-06 19:34:16,236 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:16,237 [myid:1] - INFO  [Thread-1136:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:26,155 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:26,156 [myid:1] - INFO  [Thread-1137:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:26,222 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:26,226 [myid:1] - INFO  [Thread-1138:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:36,151 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:36,152 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:36,152 [myid:1] - INFO  [Thread-1139:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:36,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:36,231 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:36,231 [myid:1] - INFO  [Thread-1140:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:46,149 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:46,149 [myid:1] - INFO  [Thread-1141:NIOServerCnxn@1008] - Closed socket co
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:46,230 [myid:1] - INFO  [NIOServerCxn.Factory:0.0.0.0/0.0.0.0:2181:NIOServe
2016-12-06 19:34:46,230 [myid:1] - INFO  [Thread-1142:NIOServerCnxn@1008] - Closed socket co
```

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
0 S zookeep+    27     1  0  80   0 - 1155556 -    20:46 ?        00:00:19 /usr/lib/jvm/java
```

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/conta

statefulset "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 "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 23.

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

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-minion-group-cxpk
kubernetes-minion-group-a5aq
kubernetes-minion-group-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-hs
        topologyKey: "kubernetes.io/hostname"
```

The `requiredDuringSchedulingIgnoredDuringExecution` field tells the Kubernetes Scheduler that it should never co-locate two Pods from the `zk-hs` Service 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 StatefulSet`.

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

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

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


kubernetes-minion-group-pb41
kubernetes-minion-group-ixsl
kubernetes-minion-group-i4c4
```

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

```
kubectl drain $(kubectl get pod zk-0 --template {{.spec.nodeName}}) --ignore-daemonsets --fo
node "kubernetes-minion-group-pb41" cordoned


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

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

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

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

```
kubectl drain $(kubectl get pod zk-1 --template {{.spec.nodeName}}) --ignore-daemonsets --fo


WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet:
pod "zk-1" deleted
node "kubernetes-minion-group-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.nodeName}}) --ignore-daemonsets --fo
node "kubernetes-minion-group-i4c4" cordoned


WARNING: Deleting pods not managed by ReplicationController, ReplicaSet, Job, or DaemonSet:
WARNING: Ignoring DaemonSet-managed pods: node-problem-detector-v0.1-dyrog; Deleting pods no
There are pending pods when an error occurred: Cannot evict pod as it would violate the pod'
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-minion-group-pb41
```

```
node "kubernetes-minion-group-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.nodeName}}) --ignore-daemonsets --fo
```

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

This time `kubectl drain` succeeds.

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

```
kubectl uncordon kubernetes-minion-group-ixsl
```

```
node "kubernetes-minion-group-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.

Create an Issue Edit this Page

Edit This Page

AppArmor

FEATURE STATE: `Kubernetes v1.4` beta

This feature is currently in a *beta* state, meaning:

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.
- Support for the overall feature will not be dropped, though details may change.

- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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

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

- Objectives
- Before you begin
- Securing a Pod
- Example
- Administration
- Authoring Profiles
- API Reference
- What's next

Objectives

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

Before you begin

Make sure:

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

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.status.nodeInfo.
gke-test-default-pool-239f5d02-gyn2: v1.4.0
gke-test-default-pool-239f5d02-x1kf: v1.4.0
gke-test-default-pool-239f5d02-xwux: v1.4.0
```

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

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

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

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

1. Container runtime is Docker – Currently the only Kubernetes-supported container runtime that also supports AppArmor is Docker. As more runtimes add AppArmor support, the options will be expanded. You can verify that your nodes are running docker with:

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.status.nodeInfo.
gke-test-default-pool-239f5d02-gyn2: docker://1.11.2
gke-test-default-pool-239f5d02-x1kf: docker://1.11.2
gke-test-default-pool-239f5d02-xwux: docker://1.11.2
```

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

1. Profile is loaded – AppArmor is applied to a Pod by specifying an AppArmor profile that each container should be run with. If any of the specified

profiles is not already loaded in the kernel, the Kubelet (>= v1.4) will reject the Pod. You can view which profiles are loaded on a node by checking the `/sys/kernel/security/apparmor/profiles` file. For example:

```
$ ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/security/apparmor/profile
apparmor-test-deny-write (enforce)
apparmor-test-audit-write (enforce)
docker-default (enforce)
k8s-nginx (enforce)
```

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

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

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {.status.conditions[?(
gke-test-default-pool-239f5d02-gyn2: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-x1kf: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-xwux: kubelet is posting ready status. AppArmor enabled
```

Securing a Pod

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

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

```
container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_ref>
```

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

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

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

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

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

```
$ kubectl get events | grep Created
22s        22s         1         hello-apparmor     Pod       spec.containers{hello}   Norma
```

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

```
$ kubectl exec <pod_name> cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

Example

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

First, we need to load the profile we want to use onto our nodes. The profile we'll use simply denies all file writes:

deny-write.profile docs/tutorials/clusters

```
#include <tunables/global>

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

  file,

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

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

```
$ NODES=(
    # The SSH-accessible domain names of your nodes
    gke-test-default-pool-239f5d02-gyn2.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-x1kf.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-xwux.us-central1-a.my-k8s)
$ for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_parser -q <<EOF
#include <tunables/global>

profile k8s-apparmor-example-deny-write flags=(attach_disconnected) {
```

```
  #include <abstractions/base>

  file,

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

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

hello-apparmor-pod.yaml docs/tutorials/clusters

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    # Tell Kubernetes to apply the AppArmor profile "k8s-apparmor-example-deny-write".
    # Note that this is ignored if the Kubernetes node is not running version 1.4 or greate
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-de
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

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

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

```
$ kubectl get events | grep hello-apparmor
14s        14s         1         hello-apparmor   Pod                                Normal
14s        14s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
```

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

```
$ kubectl exec hello-apparmor cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

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

```
$ kubectl exec hello-apparmor touch /tmp/test
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero exit code: Error exe
```

To wrap up, let's look at what happens if we try to specify a profile that hasn't been loaded:

```
$ kubectl create -f /dev/stdin <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor-2
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-all
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
EOF
pod "hello-apparmor-2" created

$ kubectl describe pod hello-apparmor-2
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kubernetes.io/hello=localhost/k8s-apparmor-e
Status:        Pending
Reason:        AppArmor
Message:       Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" is no
IP:
Controllers:   <none>
Containers:
  hello:
    Container ID:
    Image:     busybox
    Image ID:
    Port:
    Command:
      sh
      -c
      echo 'Hello AppArmor!' && sleep 1h
```

```
    State:              Waiting
      Reason:           Blocked
    Ready:              False
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-dnz7v (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-dnz7v:
    Type:    Secret (a volume populated by a Secret)
    SecretName:    default-token-dnz7v
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen    LastSeen    Count    From                          SubobjectPath    Type
  ---------    --------    -----    ----                          -------------    --------
  23s          23s         1        {default-scheduler }                           Normal
  23s          23s         1        {kubelet e2e-test-stclair-minion-group-t1f5}
```

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

Administration

Setting up nodes with profiles

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

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

The scheduler is not aware of which profiles are loaded onto which node, so the

full set of profiles must be loaded onto every node. An alternative approach is to add a node label for each profile (or class of profiles) on the node, and use a node selector to ensure the Pod is run on a node with the required profile.

Restricting profiles with the PodSecurityPolicy

If the PodSecurityPolicy extension is enabled, cluster-wide AppArmor restrictions can be applied. To enable the PodSecurityPolicy, the following flag must be set on the `apiserver`:

```
--enable-admission-plugins=PodSecurityPolicy[,others...]
```

The AppArmor options can be specified as annotations on the PodSecurityPolicy:

```
apparmor.security.beta.kubernetes.io/defaultProfileName: <profile_ref>
apparmor.security.beta.kubernetes.io/allowedProfileNames: <profile_ref>[,others...]
```

The default profile name option specifies the profile to apply to containers by default when none is specified. The allowed profile names option specifies a list of profiles that Pod containers are allowed to be run with. If both options are provided, the default must be allowed. The profiles are specified in the same format as on containers. See the API Reference for the full specification.

Disabling AppArmor

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

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

When disabled, any Pod that includes an AppArmor profile will fail validation with a "Forbidden" error. Note that by default docker always enables the "docker-default" profile on non-privileged pods (if the AppArmor kernel module is enabled), and will continue to do so even if the feature-gate is disabled. The option to disable AppArmor will be removed when AppArmor graduates to general availability (GA).

Upgrading to Kubernetes v1.4 with AppArmor

No action is required with respect to AppArmor to upgrade your cluster to v1.4. However, if any existing pods had an AppArmor annotation, they will not go through validation (or PodSecurityPolicy admission). If permissive profiles are loaded on the nodes, a malicious user could pre-apply a permissive profile to escalate the pod privileges above the docker-default. If this is a concern, it is recommended to scrub the cluster of any pods containing an annotation with `apparmor.security.beta.kubernetes.io`.

Upgrade path to General Availability

When AppArmor is ready to be graduated to general availability (GA), the options currently specified through annotations will be converted to fields. Supporting all the upgrade and downgrade paths through the transition is very nuanced, and will be explained in detail when the transition occurs. We will commit to supporting both fields and annotations for at least 2 releases, and will explicitly reject the annotations for at least 2 releases after that.

Authoring Profiles

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

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

It is recommended to run your application through Docker on a development workstation to generate the profiles, but there is nothing preventing running the tools on the Kubernetes node where your Pod is running.

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

API Reference

Pod Annotation

Specifying the profile a container will run with:

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

Profile Reference

- `runtime/default`: Refers to the default runtime profile.
  - Equivalent to not specifying a profile (without a PodSecurityPolicy default), except it still requires AppArmor to be enabled.

    - For Docker, this resolves to the `docker-default` profile for non-privileged containers, and unconfined (no profile) for privileged containers.
- `localhost/<profile_name>`: Refers to a profile loaded on the node (localhost) by name.
    - The possible profile names are detailed in the core policy reference.
- `unconfined`: This effectively disables AppArmor on the container.

Any other profile reference format is invalid.

PodSecurityPolicy Annotations

Specifying the default profile to apply to containers when none is provided:

- **key**: `apparmor.security.beta.kubernetes.io/defaultProfileName`
- **value**: a profile reference, described above

Specifying the list of profiles Pod containers is allowed to specify:

- **key**: `apparmor.security.beta.kubernetes.io/allowedProfileNames`
- **value**: a comma-separated list of profile references (described above)
    - Although an escaped comma is a legal character in a profile name, it cannot be explicitly allowed here.

What's next

Additional resources:

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

Create an Issue Edit this Page

Edit This Page

AppArmor

FEATURE STATE: `Kubernetes v1.4` beta

This feature is currently in a *beta* state, meaning:

- The version names contain beta (e.g. v2beta3).
- Code is well tested. Enabling the feature is considered safe. Enabled by default.

- Support for the overall feature will not be dropped, though details may change.
- The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable release. When this happens, we will provide instructions for migrating to the next version. This may require deleting, editing, and re-creating API objects. The editing process may require some thought. This may require downtime for applications that rely on the feature.
- Recommended for only non-business-critical uses because of potential for incompatible changes in subsequent releases. If you have multiple clusters that can be upgraded independently, you may be able to relax this restriction.
- **Please do try our beta features and give feedback on them! After they exit beta, it may not be practical for us to make more changes.**

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

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

- Objectives
- Before you begin
- Securing a Pod
- Example
- Administration
- Authoring Profiles
- API Reference
- What's next

Objectives

- See an example of how to load a profile on a node
- Learn how to enforce the profile on a Pod
- Learn how to check that the profile is loaded

- See what happens when a profile is violated
- See what happens when a profile cannot be loaded

Before you begin

Make sure:

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

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.status.nodeInfo.
gke-test-default-pool-239f5d02-gyn2: v1.4.0
gke-test-default-pool-239f5d02-x1kf: v1.4.0
gke-test-default-pool-239f5d02-xwux: v1.4.0
```

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

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

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

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

1. Container runtime is Docker – Currently the only Kubernetes-supported container runtime that also supports AppArmor is Docker. As more runtimes add AppArmor support, the options will be expanded. You can verify that your nodes are running docker with:

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {@.status.nodeInfo.
gke-test-default-pool-239f5d02-gyn2: docker://1.11.2
gke-test-default-pool-239f5d02-x1kf: docker://1.11.2
gke-test-default-pool-239f5d02-xwux: docker://1.11.2
```

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

1. Profile is loaded – AppArmor is applied to a Pod by specifying an AppArmor profile that each container should be run with. If any of the specified profiles is not already loaded in the kernel, the Kubelet (>= v1.4) will reject the Pod. You can view which profiles are loaded on a node by checking the `/sys/kernel/security/apparmor/profiles` file. For example:

```
$ ssh gke-test-default-pool-239f5d02-gyn2 "sudo cat /sys/kernel/security/apparmor/profile
apparmor-test-deny-write (enforce)
apparmor-test-audit-write (enforce)
docker-default (enforce)
k8s-nginx (enforce)
```

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

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

```
$ kubectl get nodes -o=jsonpath=$'{range .items[*]}{@.metadata.name}: {.status.conditions[?
gke-test-default-pool-239f5d02-gyn2: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-x1kf: kubelet is posting ready status. AppArmor enabled
gke-test-default-pool-239f5d02-xwux: kubelet is posting ready status. AppArmor enabled
```

Securing a Pod

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

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

```
container.apparmor.security.beta.kubernetes.io/<container_name>: <profile_ref>
```

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

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

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

Kubernetes AppArmor enforcement works by first checking that all the prerequisites have been met, and then forwarding the profile selection to the container

runtime for enforcement. If the prerequisites have not been met, the Pod will be rejected, and will not run.

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

```
$ kubectl get events | grep Created
22s        22s         1         hello-apparmor     Pod       spec.containers{hello}   Norma
```

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

```
$ kubectl exec <pod_name> cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

Example

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

First, we need to load the profile we want to use onto our nodes. The profile we'll use simply denies all file writes:

deny-write.profile docs/tutorials/clusters

```
#include <tunables/global>

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

  file,

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

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

```
$ NODES=(
    # The SSH-accessible domain names of your nodes
    gke-test-default-pool-239f5d02-gyn2.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-x1kf.us-central1-a.my-k8s
    gke-test-default-pool-239f5d02-xwux.us-central1-a.my-k8s)
$ for NODE in ${NODES[*]}; do ssh $NODE 'sudo apparmor_parser -q <<EOF
```

```
#include <tunables/global>

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

  file,

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

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

hello-apparmor-pod.yaml docs/tutorials/clusters

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor
  annotations:
    # Tell Kubernetes to apply the AppArmor profile "k8s-apparmor-example-deny-write".
    # Note that this is ignored if the Kubernetes node is not running version 1.4 or greate
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-de
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
```

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

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

```
$ kubectl get events | grep hello-apparmor
14s        14s         1         hello-apparmor   Pod                                Normal
14s        14s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
13s        13s         1         hello-apparmor   Pod       spec.containers{hello}   Normal
```

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

```
$ kubectl exec hello-apparmor cat /proc/1/attr/current
k8s-apparmor-example-deny-write (enforce)
```

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

```
$ kubectl exec hello-apparmor touch /tmp/test
touch: /tmp/test: Permission denied
error: error executing remote command: command terminated with non-zero exit code: Error exe
```

To wrap up, let's look at what happens if we try to specify a profile that hasn't been loaded:

```
$ kubectl create -f /dev/stdin <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: hello-apparmor-2
  annotations:
    container.apparmor.security.beta.kubernetes.io/hello: localhost/k8s-apparmor-example-all
spec:
  containers:
  - name: hello
    image: busybox
    command: [ "sh", "-c", "echo 'Hello AppArmor!' && sleep 1h" ]
EOF
pod "hello-apparmor-2" created

$ kubectl describe pod hello-apparmor-2
Name:          hello-apparmor-2
Namespace:     default
Node:          gke-test-default-pool-239f5d02-x1kf/
Start Time:    Tue, 30 Aug 2016 17:58:56 -0700
Labels:        <none>
Annotations:   container.apparmor.security.beta.kubernetes.io/hello=localhost/k8s-apparmor-e
Status:        Pending
Reason:        AppArmor
Message:       Pod Cannot enforce AppArmor: profile "k8s-apparmor-example-allow-write" is no
IP:
Controllers:   <none>
Containers:
  hello:
    Container ID:
    Image:     busybox
    Image ID:
    Port:
    Command:
      sh
```

```
      -c
      echo 'Hello AppArmor!' && sleep 1h
    State:              Waiting
      Reason:           Blocked
    Ready:              False
    Restart Count:      0
    Environment:        <none>
    Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-dnz7v (ro)
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-dnz7v:
    Type:    Secret (a volume populated by a Secret)
    SecretName:    default-token-dnz7v
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen    LastSeen    Count    From                          SubobjectPath    Type
  ---------    --------    -----    ----                          -------------    --------
  23s          23s         1        {default-scheduler }                           Normal
  23s          23s         1        {kubelet e2e-test-stclair-minion-group-t1f5}
```

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

Administration

Setting up nodes with profiles

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

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

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

Restricting profiles with the PodSecurityPolicy

If the PodSecurityPolicy extension is enabled, cluster-wide AppArmor restrictions can be applied. To enable the PodSecurityPolicy, the following flag must be set on the `apiserver`:

```
--enable-admission-plugins=PodSecurityPolicy[,others...]
```

The AppArmor options can be specified as annotations on the PodSecurityPolicy:

```
apparmor.security.beta.kubernetes.io/defaultProfileName: <profile_ref>
apparmor.security.beta.kubernetes.io/allowedProfileNames: <profile_ref>[,others...]
```

The default profile name option specifies the profile to apply to containers by default when none is specified. The allowed profile names option specifies a list of profiles that Pod containers are allowed to be run with. If both options are provided, the default must be allowed. The profiles are specified in the same format as on containers. See the API Reference for the full specification.

Disabling AppArmor

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

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

When disabled, any Pod that includes an AppArmor profile will fail validation with a "Forbidden" error. Note that by default docker always enables the "docker-default" profile on non-privileged pods (if the AppArmor kernel module is enabled), and will continue to do so even if the feature-gate is disabled. The option to disable AppArmor will be removed when AppArmor graduates to general availability (GA).

Upgrading to Kubernetes v1.4 with AppArmor

No action is required with respect to AppArmor to upgrade your cluster to v1.4. However, if any existing pods had an AppArmor annotation, they will not go through validation (or PodSecurityPolicy admission). If permissive profiles are loaded on the nodes, a malicious user could pre-apply a permissive profile to escalate the pod privileges above the docker-default. If this is a concern, it is

recommended to scrub the cluster of any pods containing an annotation with `apparmor.security.beta.kubernetes.io`.

Upgrade path to General Availability

When AppArmor is ready to be graduated to general availability (GA), the options currently specified through annotations will be converted to fields. Supporting all the upgrade and downgrade paths through the transition is very nuanced, and will be explained in detail when the transition occurs. We will commit to supporting both fields and annotations for at least 2 releases, and will explicitly reject the annotations for at least 2 releases after that.

Authoring Profiles

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

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

It is recommended to run your application through Docker on a development workstation to generate the profiles, but there is nothing preventing running the tools on the Kubernetes node where your Pod is running.

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

API Reference

Pod Annotation

Specifying the profile a container will run with:

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

Profile Reference

- `runtime/default`: Refers to the default runtime profile.
  - Equivalent to not specifying a profile (without a PodSecurityPolicy default), except it still requires AppArmor to be enabled.
  - For Docker, this resolves to the `docker-default` profile for non-privileged containers, and unconfined (no profile) for privileged containers.
- `localhost/<profile_name>`: Refers to a profile loaded on the node (localhost) by name.
  - The possible profile names are detailed in the core policy reference.
- `unconfined`: This effectively disables AppArmor on the container.

Any other profile reference format is invalid.

PodSecurityPolicy Annotations

Specifying the default profile to apply to containers when none is provided:

- **key**: `apparmor.security.beta.kubernetes.io/defaultProfileName`
- **value**: a profile reference, described above

Specifying the list of profiles Pod containers is allowed to specify:

- **key**: `apparmor.security.beta.kubernetes.io/allowedProfileNames`
- **value**: a comma-separated list of profile references (described above)
  - Although an escaped comma is a legal character in a profile name, it cannot be explicitly allowed here.

What's next

Additional resources:

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

Create an Issue Edit this Page

Edit This Page

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
- Before you begin
- Terminology
- Prerequisites
- Source IP for Services with Type=ClusterIP
- Source IP for Services with Type=NodePort
- Source IP for Services with Type=LoadBalancer
- Cleaning up
- What's next

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 run source-ip-app --image=k8s.gcr.io/echoserver:1.4
deployment "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
NAME                           STATUS     AGE     VERSION
kubernetes-minion-group-6jst   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-cx31   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-jj1t   Ready      2h      v1.6.0+fff5156

kubernetes-minion-group-6jst $ curl localhost:10249/proxyMode
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
service "clusterip" exposed

$ kubectl get svc clusterip
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
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
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
...
```

If the client pod and server pod are in the same node, the client_address is the client pod's IP address. However, if the client pod and server pod are in different nodes, the client_address is the client pod's node flannel IP address.

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 --typ
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
```

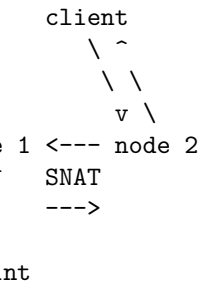
If you're running on a cloudprovider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

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



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"}}'
service "nodeport" patched
```

Now, re-run the test:

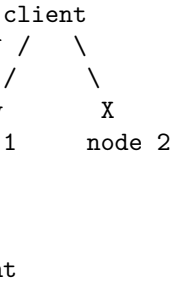
```
$ for node in $NODES; do curl --connect-timeout 1 -s $node:$NODEPORT | grep -i client_addres
client_address=104.132.1.79
```

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:



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

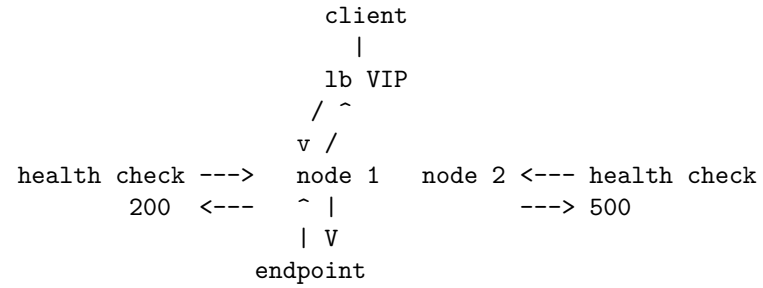
```
$ kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-port=8080 -
service "loadbalancer" exposed

$ kubectl get svc loadbalancer
NAME           CLUSTER-IP    EXTERNAL-IP       PORT(S)   AGE
loadbalancer   10.0.65.118   104.198.149.140   80/TCP    5m

$ curl 104.198.149.140
CLIENT VALUES:
client_address=10.240.0.5
...
```

However, if you're running on Google Kubernetes Engine/GCE, setting the same `service.spec.externalTrafficPolicy` field to `Local` forces nodes *without* Service endpoints to remove themselves from the list of nodes eligible for loadbalanced traffic by deliberately failing health checks.

Visually:



You can test this by setting the annotation:

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

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

```
$ kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort
  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
NAME                              READY     STATUS    RESTARTS   AGE       IP              NODE
source-ip-app-826191075-qehz4   1/1       Running   0          20h       10.180.1.136    kube

kubernetes-minion-group-6jst $ curl localhost:32122/healthz
1 Service Endpoints found

kubernetes-minion-group-jj1t $ curl localhost:32122/healthz
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
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

Create an Issue Edit this Page

Edit This Page

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 run source-ip-app --image=k8s.gcr.io/echoserver:1.4
deployment "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
NAME                           STATUS     AGE     VERSION
kubernetes-minion-group-6jst   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-cx31   Ready      2h      v1.6.0+fff5156
kubernetes-minion-group-jj1t   Ready      2h      v1.6.0+fff5156

kubernetes-minion-group-6jst $ curl localhost:10249/proxyMode
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
service "clusterip" exposed

$ kubectl get svc clusterip
NAME         CLUSTER-IP    EXTERNAL-IP   PORT(S)   AGE
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
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
...
```

If the client pod and server pod are in the same node, the client_address is the client pod's IP address. However, if the client pod and server pod are in different nodes, the client_address is the client pod's node flannel IP address.

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 --typ
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
```

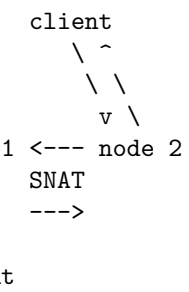
If you're running on a cloudprovider, you may need to open up a firewall-rule for the `nodes:nodeport` reported above. Now you can try reaching the Service from outside the cluster through the node port allocated above.

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



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"}}'
service "nodeport" patched
```

Now, re-run the test:

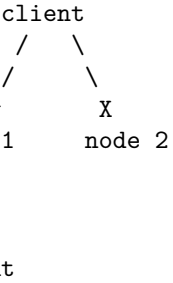
```
$ for node in $NODES; do curl --connect-timeout 1 -s $node:$NODEPORT | grep -i client_addres
client_address=104.132.1.79
```

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:



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

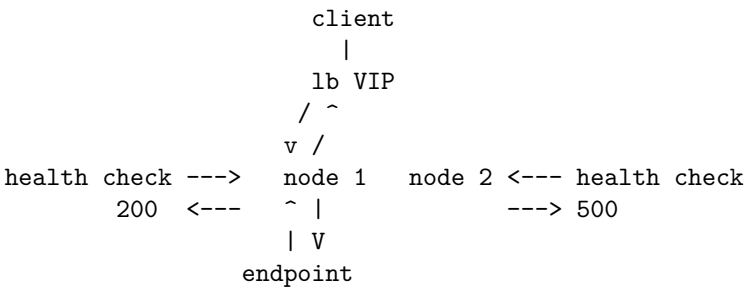
```
$ kubectl expose deployment source-ip-app --name=loadbalancer --port=80 --target-port=8080 -
service "loadbalancer" exposed

$ kubectl get svc loadbalancer
NAME           CLUSTER-IP    EXTERNAL-IP       PORT(S)   AGE
loadbalancer   10.0.65.118   104.198.149.140   80/TCP    5m

$ curl 104.198.149.140
CLIENT VALUES:
client_address=10.240.0.5
...
```

However, if you're running on Google Kubernetes Engine/GCE, setting the same `service.spec.externalTrafficPolicy` field to `Local` forces nodes *without* Service endpoints to remove themselves from the list of nodes eligible for loadbalanced traffic by deliberately failing health checks.

Visually:



You can test this by setting the annotation:

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

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

```
$ kubectl get svc loadbalancer -o yaml | grep -i healthCheckNodePort
  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
NAME                              READY     STATUS    RESTARTS   AGE       IP              NODE
source-ip-app-826191075-qehz4   1/1       Running   0          20h       10.180.1.136   kube

kubernetes-minion-group-6jst $ curl localhost:32122/healthz
1 Service Endpoints found

kubernetes-minion-group-jj1t $ curl localhost:32122/healthz
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
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

Create an Issue Edit this Page

Edit This Page

Kubernetes 201

Labels, Deployments, Services and Health Checking

If you went through Kubernetes 101, you learned about kubectl, Pods, Volumes, and multiple containers. For Kubernetes 201, we will pick up where 101 left

off and cover some slightly more advanced topics in Kubernetes, related to application productionization, Deployment and scaling.

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`.

In order for the kubectl usage examples to work, make sure you have an examples directory locally, either from a release or the source.

- TOC
  - * Labels, Deployments, Services and Health Checking
    * Labels
    * Deployments
      · Deployment Management
    * Services
      · Service Management
    * Health Checking
      · Process Health Checking
      · Application Health Checking
    * What's Next?

Labels

Having already learned about Pods and how to create them, you may be struck by an urge to create many, many Pods. Please do! But eventually you will need a system to organize these Pods into groups. The system for achieving this in Kubernetes is Labels. Labels are key-value pairs that are attached to each object in Kubernetes. Label selectors can be passed along with a RESTful `list` request to the apiserver to retrieve a list of objects which match that label selector.

To add a label, add a labels section under metadata in the Pod definition:

```
labels:
  app: nginx
```

For example, here is the nginx Pod definition with labels (pod-nginx-with-label.yaml):

pod-nginx-with-label.yaml docs/tutorials

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Create the labeled Pod (pod-nginx-with-label.yaml):

```
kubectl create -f https://k8s.io/docs/tutorials/pod-nginx-with-label.yaml
```

List all Pods with the label `app=nginx`:

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

Delete the Pod by label:

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

For more information, see Labels. They are a core concept used by two additional Kubernetes building blocks: Deployments and Services.

Deployments

Now that you know how to make awesome, multi-container, labeled Pods and you want to use them to build an application, you might be tempted to just start building a whole bunch of individual Pods, but if you do that, a whole host of operational concerns pop up. For example: how will you scale the number of Pods up or down? How will you roll out a new release?

The answer to those questions and more is to use a Deployment to manage maintaining and updating your running *Pods*.

A Deployment object defines a Pod creation template (a "cookie-cutter" if you will) and desired replica count. The Deployment uses a label selector to identify the Pods it manages, and will create or delete Pods as needed to meet the replica count. Deployments are also used to manage safely rolling out changes to your running Pods.

Here is a Deployment that instantiates two nginx Pods:

deployment.yaml docs/tutorials

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template: # create pods using pod definition in this template
    metadata:
      # unlike pod-nginx.yaml, the name is not included in the meta data as a unique name i
      # generated from the deployment name
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Deployment Management

Create an nginx Deployment:

`kubectl create -f https://k8s.io/docs/tutorials/deployment.yaml`

List all Deployments:

`kubectl get deployment`

List the Pods created by the Deployment:

`kubectl get pods -l app=nginx`

Upgrade the nginx container from 1.7.9 to 1.8 by changing the Deployment and calling `apply`. The following config contains the desired changes:

```
deployment-update.yaml docs/tutorials
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.8 # Update the version of nginx from 1.7.9 to 1.8
        ports:
        - containerPort: 80
```

```
kubectl apply -f https://k8s.io/docs/tutorials/deployment-update.yaml
```

Watch the Deployment create Pods with new names and delete the old Pods:

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

Delete the Deployment by name:

```
kubectl delete deployment nginx-deployment
```

For more information, such as how to rollback Deployment changes to a previous version, see *Deployments*.

Services

Once you have a replicated set of Pods, you need an abstraction that enables connectivity between the layers of your application. For example, if you have a Deployment managing your backend jobs, you don't want to have to reconfigure your front-ends whenever you re-scale your backends. Likewise, if the Pods in your backends are scheduled (or rescheduled) onto different machines, you can't be required to re-configure your front-ends. In Kubernetes, the service abstraction achieves these goals. A service provides a way to refer to a set of

Pods (selected by labels) with a single static IP address. It may also provide load balancing, if supported by the provider.

For example, here is a service that balances across the Pods created in the previous nginx Deployment example (service.yaml):

service.yaml docs/tutorials

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  ports:
  - port: 8000 # the port that this service should serve on
    # the container on each pod to connect to, can be a name
    # (e.g. 'www') or a number (e.g. 80)
    targetPort: 80
    protocol: TCP
  # just like the selector in the deployment,
  # but this time it identifies the set of pods to load balance
  # traffic to.
  selector:
    app: nginx
```

Service Management

Create an nginx service (service.yaml):

```
kubectl create -f https://k8s.io/docs/tutorials/service.yaml
```

List all services:

```
kubectl get services
```

On most providers, the service IPs are not externally accessible. The easiest way to test that the service is working is to create a busybox Pod and exec commands on it remotely. See the command execution documentation for details.

Provided the service IP is accessible, you should be able to access its http endpoint with wget on the exposed port:

```
$ export SERVICE_IP=$(kubectl get service nginx-service -o go-template='{{.spec.clusterIP}}'
$ export SERVICE_PORT=$(kubectl get service nginx-service -o go-template='{{(index .spec.por
$ echo "$SERVICE_IP:$SERVICE_PORT"
$ kubectl run busybox  --generator=run-pod/v1 --image=busybox --restart=Never --tty -i --env
```

```
u@busybox$ wget -qO- http://$SERVICE_IP:$SERVICE_PORT # Run in the busybox container
u@busybox$ exit # Exit the busybox container
$ kubectl delete pod busybox # Clean up the pod we created with "kubectl run"
```

The service definition exposed the Nginx Service as port 8000 (`$SERVCE_PORT`).
We can also access the service from a host running Kubernetes using that port:

```
wget -qO- http://$SERVICE_IP:$SERVICE_PORT # Run on a Kubernetes host
```

(This works on AWS with Weave.)

To delete the service by name:

```
kubectl delete service nginx-service
```

When created, each service is assigned a unique IP address. This address is tied to the lifespan of the Service, and will not change while the Service is alive. Pods can be configured to talk to the service, and know that communication to the service will be automatically load-balanced out to some Pod that is a member of the set identified by the label selector in the Service.

For more information, see Services.

Health Checking

When I write code it never crashes, right? Sadly the Kubernetes issues list indicates otherwise…

Rather than trying to write bug-free code, a better approach is to use a management system to perform periodic health checking and repair of your application. That way a system outside of your application itself is responsible for monitoring the application and taking action to fix it. It's important that the system be outside of the application, since if your application fails and the health checking agent is part of your application, it may fail as well and you'll never know. In Kubernetes, the health check monitor is the Kubelet agent.

Process Health Checking

The simplest form of health-checking is just process level health checking. The Kubelet constantly asks the Docker daemon if the container process is still running, and if not, the container process is restarted. In all of the Kubernetes examples you have run so far, this health checking was actually already enabled. It's on for every single container that runs in Kubernetes.

Application Health Checking

However, in many cases this low-level health checking is insufficient. Consider, for example, the following code:

```
lockOne := sync.Mutex{}
lockTwo := sync.Mutex{}

go func() {
  lockOne.Lock();
  lockTwo.Lock();
  ...
}()

lockTwo.Lock();
lockOne.Lock();
```

This is a classic example of a problem in computer science known as "Deadlock". From Docker's perspective your application is still operating and the process is still running, but from your application's perspective your code is locked up and will never respond correctly.

To address this problem, Kubernetes supports user implemented application health-checks. These checks are performed by the Kubelet to ensure that your application is operating correctly for a definition of "correctly" that *you* provide.

Currently, there are three types of application health checks that you can choose from:

- HTTP Health Checks - The Kubelet will call a web hook. If it returns between 200 and 399, it is considered success, failure otherwise. See health check examples here.
- Container Exec - The Kubelet will execute a command inside your container. If it exits with status 0 it will be considered a success. See health check examples here.
- TCP Socket - The Kubelet will attempt to open a socket to your container. If it can establish a connection, the container is considered healthy, if it can't it is considered a failure.

In all cases, if the Kubelet discovers a failure the container is restarted.

The container health checks are configured in the `livenessProbe` section of your container config. There you can also specify an `initialDelaySeconds` that is a grace period from when the container is started to when health checks are performed, to enable your container to perform any necessary initialization.

Here is an example config for a Pod with an HTTP health check (pod-with-http-healthcheck.yaml):

```
pod-with-http-healthcheck.yaml docs/tutorials
```

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-http-healthcheck
spec:
  containers:
  - name: nginx
    image: nginx
    # defines the health checking
    livenessProbe:
      # an http probe
      httpGet:
        path: /_status/healthz
        port: 80
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
    - containerPort: 80
```

And here is an example config for a Pod with a TCP Socket health check (pod-with-tcp-socket-healthcheck.yaml):

pod-with-tcp-socket-healthcheck.yaml docs/tutorials

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-tcp-socket-healthcheck
spec:
  containers:
  - name: redis
    image: redis
    # defines the health checking
    livenessProbe:
      # a TCP socket probe
      tcpSocket:
        port: 6379
      # length of time to wait for a pod to initialize
      # after pod startup, before applying health checking
      initialDelaySeconds: 30
      timeoutSeconds: 1
    ports:
    - containerPort: 6379
```

For more information about health checking, see Container Probes.

What's Next?

For a complete application see the guestbook example.

Create an Issue Edit this Page