A guide to Kubernetes for SREs and sysadmins

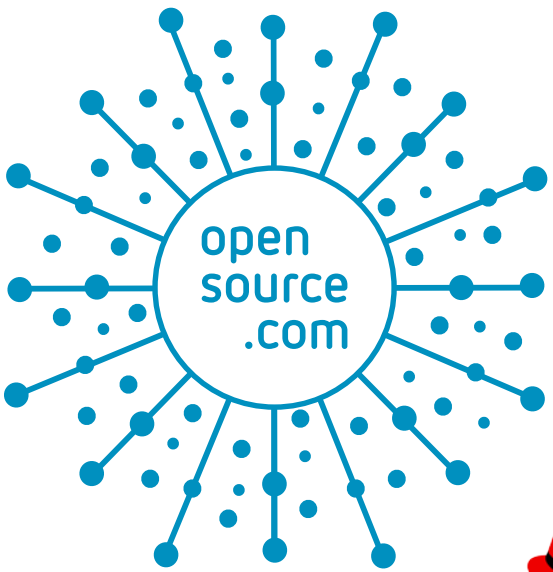
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

JESS CHERRY

Tech nomad, working in about anything I can find.

Evangelist of silo prevention in the IT space, the importance of information sharing with all teams.

Believer in educating all and open source development.

Lover of all things tech.All about K8s, chaos and anything new and shiny I can find!

Follow me at @alynderthered1

CONTENTS

Explore the Kubernetes Ecosystem

BY CHRIS COLLINS

SINCE ITS INTRODUCTION, KUBERNETES, has grown to become the defacto standard for container orchestration and development. It is complex and sometimes complicated, but also wonderfully extensible and open source, allowing it to play to its core strength—container orchestration—while allowing a vibrant community of enthusiasts, professionals, and everyone in between to build tools to extend Kubernetes to fit their needs.

This eBook is a great resource for jumping into Kubernetes. Ben Finkel provides a concise and helpful introduction to Kubernetes concepts and how to get started with Kubernetes at home. It's also a great series of articles by Jessica Cherry, focusing on getting to know Kubernetes more thoroughly, additional tools that help make working with Kubernetes easier, and exploration of several Kubernetes extensions, that open up and extend Kubernetes and take advantage of its power and flexibility for building and deploying applications and add new functionality.

Overall, this collection is an excellent overview of the ecosystem available around the most popular and well-known container orchestration system available, helpful for both beginners new to the Kubernetes world, and veterans who are interested in learning more about some of the available tools out there.

Getting started with Kubernetes

BY BEN FINKEL

Learn the basics of using the open source container management system with this easy tutorial.

ONE OF TODAY'S MOST promising emerging technologies is paring containers with cluster management software such as Docker Swarm [1], Apache Mesos [2], and the popular Kubernetes [3]. Kubernetes allows you to create a portable and scalable application deployment that can be scheduled, managed, and maintained easily. As an open source project, Kubernetes is continually being updated and improved, and it leads the way among container cluster management software.

Kubernetes uses various architectural components to describe the deployments it manages.

- Pods [4] are a group of one or more containers that share network and storage. The containers in a pod are considered "tightly coupled," and they are managed and deployed as a single unit. If an application were deployed in a more traditional model, the contents of the pod would always be deployed together on the same machine.
- Nodes [5] represents a worker machine in a Kubernetes cluster. The worker machine can be either physical or (more likely) virtual. A node contains all the required services to host a pod.
- A cluster always requires a master [6] node, where the controlling services (known as the master components) are installed. These services can be distributed on a single machine or across multiple machines for redundancy. They control communications, workload, and scheduling.
- Deployments [7] are a way to declaratively set a state for your pods or ReplicaSets (groups of pods to be deployed together). Deployments use a "desired state" format to describe how the deployment should look, and Kubernetes handles the actual deployment tasks. Deployments can be updated, rolled back, scaled, and paused at will.

The following tutorial will explain the basics of creating a cluster, deploying an app, and creating a proxy, then send you on your way to learning even more about Kubernetes.

Create a cluster

Begin by using the Kubernetes-provided tutorial [8] to create a cluster and deploy an app. This cluster will consist of a master and one or more nodes. In the first scenario, you'll create a cluster using a utility called "Minkube," which creates and runs a cluster on a local machine. Minikube is great for testing and development. You will also use the **kubectl** command, which is installed as part of the Kubernetes API.

In the interactive terminal, start the Minikube software with the command:

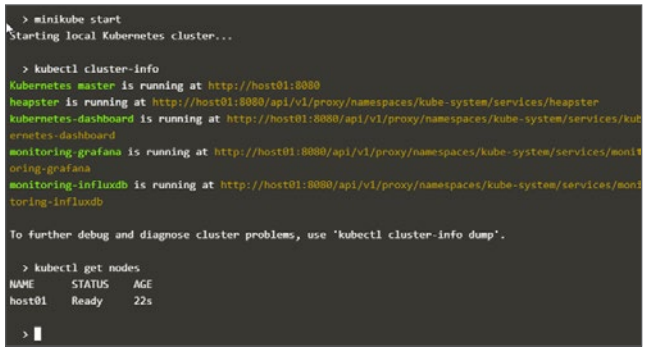
```
minikube start
```

View the cluster information with the command:

```
kubectl cluster-info
```

List the available nodes with the command:

```
kubectl get nodes
```



The screenshot above shows the output from these commands. Note the only available node is host01, which is operating as the master (as seen in the cluster-info output).

Deploy an app

In the next step [9] in the interactive tutorial, you'll deploy a containerized application to your cluster with a deployment configuration. This describes how to create instances of your app, and the master will schedule those instances onto nodes in the cluster.

In the interactive terminal, create a new deployment with the **kubectl** run command:

```
kubectl run kubernetes-bootcamp \
--image=docker.io/jocatalin/kubernetes-bootcamp:v1 --port=8080
```

This creates a new deployment with the name **kubernetes-bootcamp** from a public repository at docker.io and overrides the default port to 8080.

View the deployment with the command:

```
kubectl get deployments
```

The deployment is currently on a single node (host01), because only that node is available.

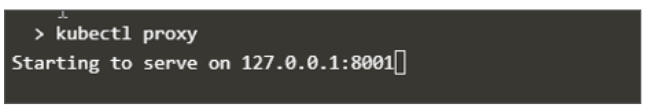
Create a proxy

In the third part [9] of the tutorial, you will create a proxy into your deployed app. A pod runs on an isolated private network that cannot be accessed from outside. The **kubectl** command uses an API to communicate with the application, and a proxy is needed to expose the application for use by other services.

Open a new terminal window and start the proxy server with the command:

```
kubectl proxy
```

This creates a connection between your cluster and the virtual terminal window. Notice it's running on port 8001 on the local host.



Return to the first terminal window and run a **curl** command to see this in action:

```
curl http://localhost:8001/version
```

The JSON output, shown in the screenshot above, displays the version information from the cluster itself.

Follow the online tutorial to find the internal name of the deployed pod and then query that pod directly. You can also get a detailed output of your pod by using the command:

```
kubectl describe pods
```

This output includes very important information, like the pod name, local IP address, state, and restart count.

Moving forward

Kubernetes is a full-fledged deployment, scheduling, and scaling manager and is capable of deciding all of the myriad details of how to deploy an app on your cluster. The few commands explored here are just the beginning of interacting with and understanding a Kubernetes deployment. The crucial takeaway is how fast and easy it is to do and how few details you need to provide to use Kubernetes.

Follow the online interactive tutorial [10] to learn more about how Kubernetes works and all that you can do with it.

Links

[1] https://docs.docker.com/engine/swarm/
[2] http://mesos.apache.org/
[3] https://kubernetes.io/
[4] https://kubernetes.io/docs/concepts/workloads/pods/pod/
[5] https://kubernetes.io/docs/concepts/architecture/nodes/
[6] https://kubernetes.io/docs/concepts/overview/components/
[7] https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[8] https://kubernetes.io/docs/tutorials/kubernetes-basics/
[9] https://kubernetes.io/docs/tutorials/kubernetes-basics/deploy-interactive/
[10] https://kubernetes.io/docs/tutorials/kubernetes-basics/explore-intro/

Set up Minishift and run Jenkins on Linux

Install, configure, and use Minishift to create your first pipeline.

MINISHIFT [1] is a tool that helps you run OKD [2] (Red Hat's open source OpenShift container platform) locally by launching a single-node OKD cluster inside a virtual machine. It is powered by Kubernetes [3], which is one of my favorite things to talk about.

In this article, I will demonstrate how to get started with Minishift on Linux. This was written for Ubuntu 18.04, and you'll need sudo access [4] on your Linux machine to run some commands.

Prerequisites

Before starting the installation, your Linux machine must have either KVM for Linux or VirtualBox [5], which runs on every platform. This demo uses KVM, which you can install along with all the required dependencies:

```
$ sudo apt install qemu-kvm \
libvirt-clients libvirt-daemon-system \
bridge-utils virt-manager
```

After installing KVM, you must make some modifications to allow your user to use it. Specifically, you must add your user name to the libvirt group:

```
$ sudo usermod --append --groups libvirt $(whoami)
$ newgrp libvirt
```

Next, install the Docker KVM driver, which is needed to run containers on Minishift. I downloaded the Docker machine driver directly to `/usr/local/bin`. You don't have to save it to `/usr/local/bin`, but you must ensure that its location is in your PATH [6]:

```
$ curl -L https://github.com/dhiltgen/docker-machine-kvm/
  releases/download/v0.10.0/docker-machine-driver-kvm-
  ubuntu16.04 \
-o /usr/local/bin/docker-machine-driver-kvm

$ sudo chmod +x /usr/local/bin/docker-machine-driver-kvm
```

Install Minishift

Now that the prerequisites are in place, visit the Minishift releases page [7] and determine which version of Minishift you want to install. I used Minishift v1.34.3 [8].

Download the Linux .tar file [9] to a directory you will be able to find easily. I used the minishift directory:

```
$ ls
Minishift-1.34.3-linux-amd64.tgz
```

Next, untar your new file using the `tar` command [10]:

```
$ tar zxvf minishift-1.34.3-linux-amd64.tgz
minishift-1.34.3-linux-amd64/
minishift-1.34.3-linux-amd64/LICENSE
minishift-1.34.3-linux-amd64/README.adoc
minishift-1.34.3-linux-amd64/minishift
```

By using the `v` (for *verbose*) option in your command, you can see all the files and their locations in your directory structure.

Run the `ls` command to confirm that the new directory was created:

```
$ ls
minishift-1.34.3-linux-amd64
```

Next, change to the new directory and find the binary file you need; it is named `minishift`:

```
$ cd minishift-1.34.3-linux-amd64
$ ls
LICENSE  minishift  README.adoc
$
```

Move the `minishift` binary file to your PATH, which you can find by running the following and looking at the output:

```
$ echo $PATH
/home/jess/.local/bin:/usr/local/sbin:/usr/local/bin
```

I used `/usr/local/bin` as the minishift binary file's location:

```
$ sudo mv minishift /usr/local/bin
[sudo] password for jess:
$ ls /usr/local/bin
minishift
```

Run the `minishift` command and look at the output:

```
$ minishift
Minishift is a command-line tool that provisions and manages single-
  node OpenShift clusters optimized for development workflows.

Usage:
  minishift [command]

Available Commands:
  addons      Manages Minishift add-ons.
  completion  Outputs minishift shell completion for the given shell
  config      Modifies Minishift configuration properties.
  console     Opens or displays the OpenShift Web Console URL.
[...]

Use "minishift [command] --help" for more information about a
  command.
```

Log into Minishift's web console

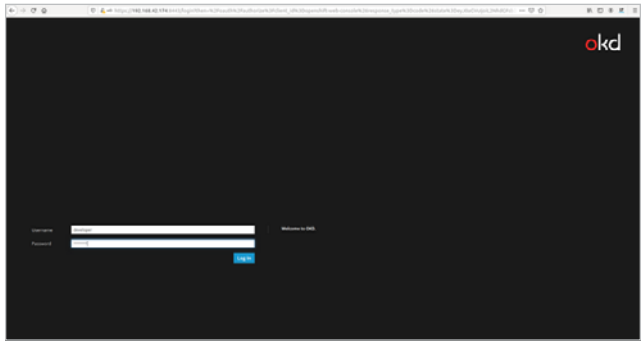
Now that Minishift is installed, you can walk through it and play with some cool new software. Begin with `minishift start`. This, as you might guess, starts Minishift—specifically, it starts a one-node cluster on your computer:

```
$ minishift start
  Starting profile 'minishift'
  Check if deprecated options are used ... OK
  Checking if https://github.com is reachable ... OK
[...]
  Minishift will be configured with...
  Memory: 4GB
  vCPUs : 2GB
  Disk size: 20 GB
Starting Minishift VM ..........OK
```

This process can take a long time, depending on your hardware, so be patient. When it ends, you'll get information about where to find your imaginary cluster on your virtualized network:

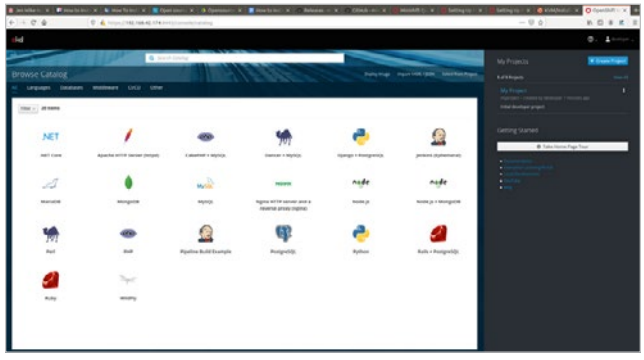
```
Server Information ...
MiniShift server started.
The server is accessible via web console at:
https://192.168.42.66:8443/console
```

Now, MiniShift is running, complete with a web console. You can log into the OKD console using **developer** as the user name and any password you want. I chose **developer / developer**.



(Jess Cherry, CC BY-SA 4.0)

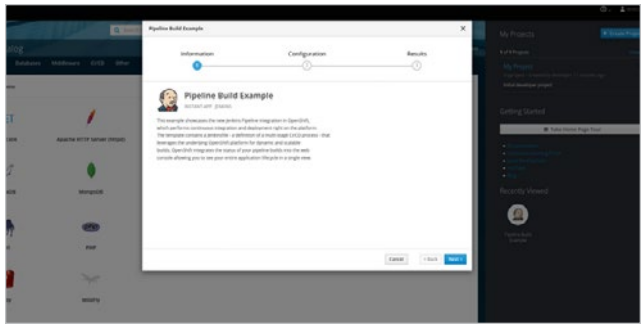
The web console is an easy control panel you can use to administer your humble cluster. It's a place for you to create and load container images, add and monitor pods, and ensure your instance is healthy.



(Jess Cherry, CC BY-SA 4.0)

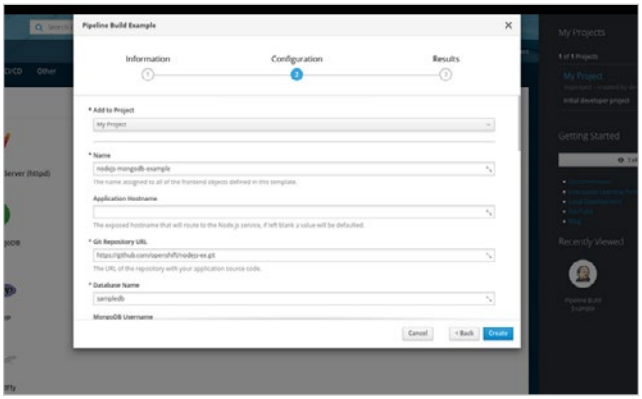
Build a pipeline

To start building your first pipeline, click **Pipeline Build Example** on the console. Click **Next** to show the parameters available to create the pipeline project.



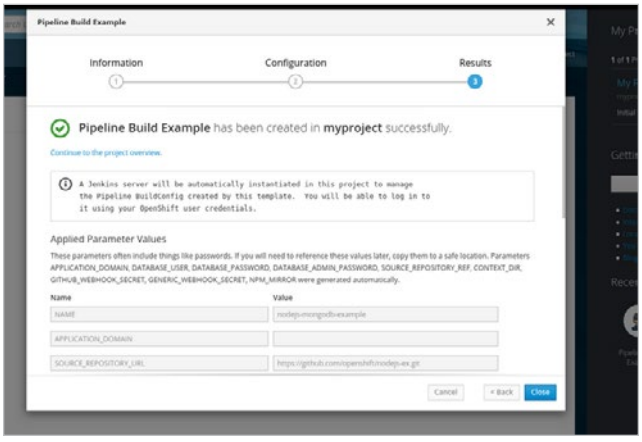
(Jess Cherry, CC BY-SA 4.0)

A window appears with parameters to fill in if you want; you can use what's already there for this example. Walk through the rest of the screen choices to create a sample pipeline.



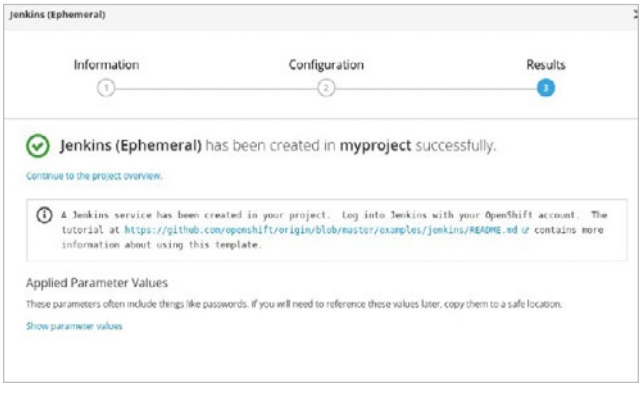
(Jess Cherry, CC BY-SA 4.0)

Click **Create**, and let Minishift create the project for you. It shows your success (or failure).

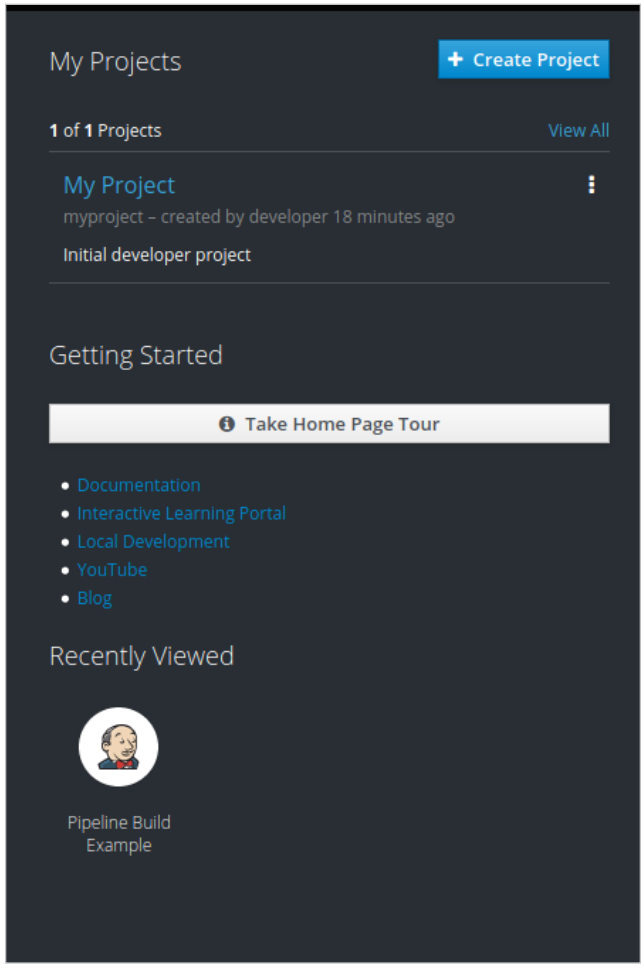


(Jess Cherry, CC BY-SA 4.0)

You can also click **Show Parameters** and scroll through the list of parameters configured for this project. Click **Close** and look for a confirmation message on the left.

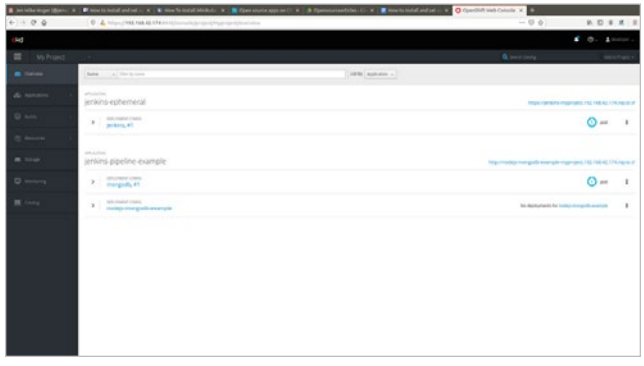


(Jess Cherry, CC BY-SA 4.0)



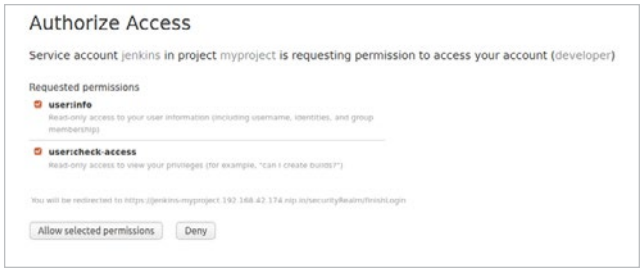
(Jess Cherry, CC BY-SA 4.0)

When you click on **My Project**, you can see the details and pods created for the project to run.



(Jess Cherry, CC BY-SA 4.0)

Open the `jenkins-ephemeral` link that was generated. Log in again with the **developer** credentials and allow access to run a pipeline in Jenkins.



(Jess Cherry, CC BY-SA 4.0)

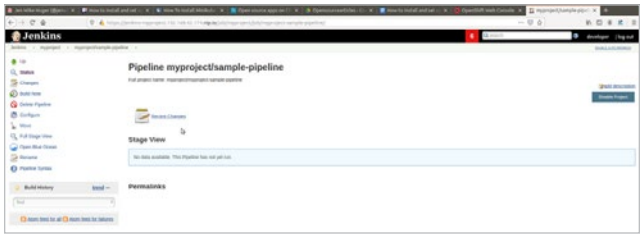
Now you can look through the Jenkins interface to get a feel for what it has to offer.

(Jess Cherry, CC BY-SA 4.0)

Find your project.

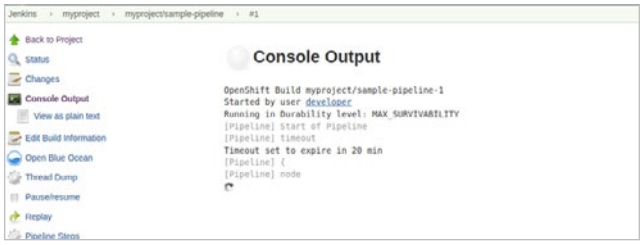
(Jess Cherry, CC BY-SA 4.0)

When you're ready, click **Build Now**.



(Jess Cherry, CC BY-SA 4.0)

Then you can view the job's output in the console output.



(Jess Cherry, CC BY-SA 4.0)

Once the job completes successfully, you will see a success message at the bottom of the console.

What did this pipeline do? It updated the deployment manually.

(Jess Cherry, CC BY-SA 4.0)

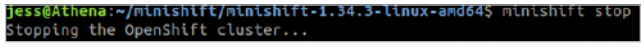
Congratulations, you successfully created an example automated deployment using Minishift!

Clean it up

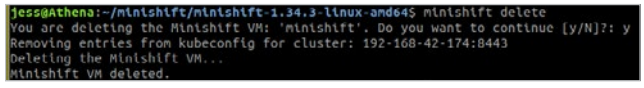
The last thing to do is to clean up everything by running two commands:

```
$ minishift stop
$ minishift delete
```

Why `stop` and then `delete`? Well, I like to make sure nothing is running before I run a delete command of any kind. This results in a cleaner delete without the possibility of having any leftover or hung processes. Here are the commands' output.



(Jess Cherry, CC BY-SA 4.0)



(Jess Cherry, CC BY-SA 4.0)

Final notes

Minishift is a great tool with great built-in automation. The user interface is comfortable to work with and easy on the eyes. I found it a fun new tool to play with at home, and if you want to dive in deeper, just look over the great documentation [11] and many online tutorials [12]. I recommend exploring this application in depth. Have a happy time Minishifting!

Links

[1] https://www.okd.io/minishift/
[2] https://www.redhat.com/sysadmin/learn-openshift-minishift
[3] https://opensource.com/resources/what-is-kubernetes
[4] https://en.wikipedia.org/wiki/Sudo
[5] https://www.virtualbox.org/wiki/Downloads
[6] https://opensource.com/article/17/6/set-path-linux
[7] https://github.com/minishift/minishift/releases
[8] https://github.com/minishift/minishift/releases/tag/v1.34.3
[9] https://github.com/minishift/minishift/releases/download/v1.34.3/minishift-1.34.3-linux-amd64.tgz
[10] https://opensource.com/article/17/7/how-unzip-targz-file
[11] https://docs.okd.io/3.11/minishift/using/index.html
[12] https://www.redhat.com/sysadmin/learn-openshift-minishift

Kubernetes namespaces for beginners

What is a namespace and why do you need it?

WHAT IS IN A KUBERNETES NAMESPACE? As Shakespeare once wrote, which we call a namespace, by any other name, would still be a virtual cluster. By virtual cluster, I mean Kubernetes can offer multiple Kubernetes clusters on a single cluster, much like a virtual machine is an abstraction of its host. According to the Kubernetes docs [1]:

> Kubernetes supports multiple virtual clusters backed by the same physical cluster. These virtual clusters are called namespaces.

Why do you need to use namespaces? In one word: isolation.

Isolation has many advantages, including that it supports secure and clean environments. If you are the owner of the infrastructure and are supporting developers, isolation is fairly important. The last thing you need is someone who is unfamiliar with how your cluster is built going and changing the system configuration—and possibly disabling everyone's ability to log in.

The namespaces that start it all

The first three namespaces created in a cluster are always **default**, **kube-system**, and **kube-public**. While you can technically deploy within these namespaces, I recommend leaving these for system configuration and not for your projects.

- **Default** is for deployments that are not given a namespace, which is a quick way to create a mess that will be hard to clean up if you do too many deployments without the proper information. I leave this alone because it serves that one purpose and has confused me on more than one occasion.
- **Kube-system** is for all things relating to, you guessed it, the Kubernetes system. Any deployments to this namespace are playing a dangerous game and can accidentally cause irreparable damage to the system itself. Yes, I have done it; I do not recommend it.
- **Kube-public** is readable by everyone, but the namespace is reserved for system usage.

Using namespaces for isolation

I have used namespaces for isolation in a couple of ways. I use them most often to split many users' projects into separate environments. This is useful in preventing cross-project contamination since namespaces provide independent environments. Users can install multiple versions of Jenkins, for example, and their environmental variables won't collide if they are in different namespaces.

This separation also helps with cleanup. If development groups are working on various projects that suddenly become obsolete, you can delete the namespace and remove everything in one swift movement with **kubectl delete ns <$NAMESPACENAME>**. (Please make sure it's the right namespace. I deleted the wrong one in production once, and it's not pretty.)

Be aware that this can cause damage across teams and problems for you if you are the infrastructure owner. For example, if you create a namespace with some special, extra-secure DNS functions and the wrong person deletes it, all of your pods and their running applications will be removed with the namespace. Any use of **delete** should be reviewed by a peer (through GitOps [2]) before hitting the cluster.

While the official documentation suggests not using multiple namespaces with 10 or fewer users [3], I still use them in my own cluster for architectural purposes. The cleaner the cluster, the better.

What admins need to know about namespaces

For starters, namespaces cannot be nested in other namespaces. There can be only one namespace with deployments in it. You don't have to use namespaces for versioned projects, but you can always use the labels to separate versioned apps with the same name. Namespaces divide resources between users using resource quotas;

for example, *this namespace can only have x number of nodes*. Finally, all namespaces scope down to a unique name for the resource type.

Namespace commands in action

To try out the following namespace commands, you need to have Minikube [4], Helm [5], and the kubectl [6] command line installed. For information about installing them, see my article *Security scanning your DevOps pipeline* [7] or each project's homepage. I am using the most recent release of Minikube. The manual installation is fast and has consistently worked correctly the first time.

To get your first set of namespaces:

```
jess@Athena:~$ kubectl get namespace
NAME              STATUS   AGE
default           Active   5m23s
kube-public       Active   5m24s
kube-system       Active   5m24s
```

To create a namespace:

```
jess@Athena:~$ kubectl create namespace athena
namespace/athena created
```

Now developers can deploy to the namespace you created; for example, here's a small and easy Helm chart:

```
jess@Athena:~$ helm install teset-deploy stable/redis --namespace
  athena
NAME: teset-deploy
LAST DEPLOYED: Sat Nov 23 13:47:43 2019
NAMESPACE: athena
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
** Please be patient while the chart is being deployed **
Redis can be accessed via port 6379 on the following DNS names from
  within your cluster:

teset-deploy-redis-master.athena.svc.cluster.local for read/write
  operations
teset-deploy-redis-slave.athena.svc.cluster.local for read-only
  operations
```

To get your password:

```
export REDIS_PASSWORD=$(kubectl get secret --namespace athena
  teset-deploy-redis -o jsonpath="{.data.redis-password}" |
  base64 --decode)
```

To connect to your Redis server:

1. Run a Redis pod that you can use as a client:

```
kubectl run --namespace athena teset-deploy-redis-client --rm
--tty -i --restart='Never' \
        --env REDIS_PASSWORD=$REDIS_PASSWORD \
--image docker.io/bitnami/redis:5.0.7-debian-9-r0 -- bash
```

2. Connect using the Redis CLI:

```
redis-cli -h teset-deploy-redis-master -a $REDIS_PASSWORD
redis-cli -h teset-deploy-redis-slave -a $REDIS_PASSWORD
```

To connect to your database from outside the cluster:

```
kubectl port-forward --namespace athena svc/teset-deploy-redis-
  master 6379:6379 &
redis-cli -h 127.0.0.1 -p 6379 -a $REDIS_PASSWORD
```

Now that this deployment is out, you have a chart deployed in your namespace named **test-deploy**.

To look at what pods are in your namespace:

```
jess@Athena:~$ kubectl get pods --namespace athena
NAME                             READY   STATUS  RESTARTS    AGE
teset-deploy-redis-master-0      1/1     Running   0         2m38s
teset-deploy-redis-slave-0       1/1     Running   0         2m38s
teset-deploy-redis-slave-1       1/1     Running   0           90s
```

At this point, you have officially isolated your application to a single namespace and created one virtual cluster that talks internally only to itself.

Delete everything with a single command:

```
jess@Athena:~$ kubectl delete namespace athena
namespace "athena" deleted
```

Because this deletes the application's entire internal configuration, the delete may take some time, depending on how large your deployment is.

Double-check that everything has been removed:

```
jess@Athena:~$ kubectl get pods --all-namespaces
NAMESPACE      NAME                              READY STATUS  RESTARTS  AGE
kube-system   coredns-5644d7b6d9-4vxv6           1/1   Running    0      32m
kube-system   coredns-5644d7b6d9-t5wn7           1/1   Running    0      32m
kube-system   etcd-minikube                      1/1   Running    0      31m
kube-system   kube-addon-manager-minikube        1/1   Running    0      32m
kube-system   kube-apiserver-minikube            1/1   Running    0      31m
kube-system   kube-controller-manager-minikube 1/1   Running    0      31m
kube-system   kube-proxy-5tdmh                   1/1   Running    0      32m
kube-system   kube-scheduler-minikube            1/1   Running    0      31m
kube-system   storage-provisioner                1/1   Running    0      27m
```

This is a list of all the pods and all the known namespaces where they live. As you can see, the application and namespace you previously made are now gone.

Namespaces in practice

I currently use namespaces for security purposes, including reducing the privileges of users with limitations. You can limit everything—from which roles can access a namespace to their quota levels for cluster resources, like CPUs. For example, I use resource quotas and role-based access control (RBAC) configurations to confirm that a namespace is accessible only by the appropriate service accounts.

On the isolation side of security, I don't want my home Jenkins application to be accessible over a trusted local network as secure images that have public IP addresses (and thus, I have to assume, could be compromised).

Namespaces can also be helpful for budgeting purposes if you have a hard budget on how much you can use in your cloud platform for nodes (or, in my case, how much I can deploy before segfaulting [8] my home server). Although this is out of scope for this article, and it's complicated, it is worth researching and taking advantage of to prevent overextending your cluster.

Conclusion

Namespaces are a great way to isolate projects and applications. This is just a quick introduction to the topic, so I encourage you to do more advanced research on namespaces and use them more in your work.

Links

[1] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
[2] https://www.weave.works/blog/gitops-operations-by-pull-request
[3] https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/
[4] https://kubernetes.io/docs/tasks/tools/install-minikube/
[5] https://helm.sh/
[6] https://kubernetes.io/docs/tasks/tools/install-kubectl/
[7] https://opensource.com/article/19/7/security-scanning-your-devops-pipeline
[8] https://en.wikipedia.org/wiki/Segmentation_fault

9 kubectl commands sysadmins need to know

Download our new kubectl cheat sheet to learn helpful commands for the Kubernetes command-line utility.

KUBERNETES [1] is the dominant technology for infrastructure today, and that means sysadmins need to be familiar with administering it. I have been managing Kubernetes clusters every day for years now, and I picked up a few tricks that I highly recommend for others looking to simplify their admin experience.

I created this cheat sheet [2] to share the key notes about kubectl and the commands I use daily to keep clusters up and running. It's broken up into sections to help you gauge whether or not you should use them for certain tasks. I also included some flags in both long-form and shorthand to help get you fluent with them more quickly.

Get, create, edit, and delete resources with kubectl

The safest place to start with a command-line utility is to ask questions (read operations) rather than give commands (write operations). The helpful get commands can **get** you rolling.

Kubectl get

Use **get** to pull a list of resources you have currently on your cluster. The types of resources you can **get** include:

- Namespace
- Pod
- Node
- Deployment
- Service
- ReplicaSets

Each of these provides details about the available resources in the cluster. As an example, here's the output of the **get nodes** command, which provides a version of Kubernetes in usage and status.

```
$ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   9d    v1.18.0
```

Most of these commands have shortened versions. To get the namespaces, you can run **kubectl get namespaces** or **kubectl get ns** (see the cheat sheet for the full list):

```
$ kubectl get ns
NAME              STATUS   AGE
charts            Active   8d
default           Active   9d
kube-node-lease   Active   9d
kube-public       Active   9d
kube-system       Active   9d
```

Each **get** command can focus in on a given namespace with the **–namespace** or **-n** flag. I use especially help when you want to review the pods in **kube-system**, which are the services needed to run Kubernetes itself.

```
$ kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS   AGE
coredns-66bff467f8-mjptx           1/1     Running   2          9d
coredns-66bff467f8-t2xcz           1/1     Running   2          9d
etcd-minikube                      1/1     Running   1          9d
kube-apiserver-minikube            1/1     Running   1          9d
kube-controller-manager-minikube   1/1     Running   2          9d
kube-proxy-rpc9d                   1/1     Running   1          9d
kube-scheduler-minikube            1/1     Running   2          9d
storage-provisioner                1/1     Running   1          9d
```

Kubectl create

Now that we've gathered some resources, let's create some more. With kubectl, you can create nearly any type of resource in a cluster. Some of these resources do require con-

figuration files and namespaces to set the resource to, as well as names. Resources you can create include:

- service
- cronjob
- deployment
- job
- namespace (ns)

So, for example, **create namespace** requires another parameter to name the namespace.

```
$ kubectl create ns hello-there
namespace/hello-there created
```

We can also create continuously running jobs with cron like many Linux friends will be familiar with [3]. Here we use **cronjob** to echoes "hello" every five seconds.

```
$ kubectl create cronjob my-cron --image=busybox
  --schedule="*/5 * * * *" -- echo hello
cronjob.batch/my-namespaced-cron created
```

You can also use the shortened version, **cj**, rather than **cronjob**.

```
$ kubectl create cj my-existing-cron --image=busybox
  --schedule="*/15 * * * *" -- echo hello
cronjob.batch/my-existing-cron created
```

Kubectl edit

So, what happens when we've created something, and we want to update? That's where **kubectl edit** comes in.

You can edit any resource in your cluster when you run this command. It will open your default text editor. So we'll edit our existing cron job, can we run:

```
$ kubectl edit cronjob/my-existing-cron
```

This shows our configuration to edit.

```
# Please edit the object below. Lines beginning with a '#' will
# be ignored, and an empty file will abort the edit. If an error
# occurs while saving this file will be reopened with the relevant
# failures.
#
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  creationTimestamp: "2020-04-19T16:06:06Z"
  managedFields:
  - apiVersion: batch/v1beta1
    fieldsType: FieldsV1
    fieldsV1:
      f:spec:
        f:concurrencyPolicy: {}
        f:failedJobsHistoryLimit: {}
        f:jobTemplate:
          f:metadata:
            f:name: {}
          f:spec:
            f:template:
              f:spec:
                f:containers:
                  k:{"name":"my-new-cron"}:
                    .: {}
                    f:command: {}
                    f:image: {}
                    f:imagePullPolicy: {}
```

The schedule is set to every 15 seconds:

```
schedule: '*/15 * * * *'
successfulJobsHistoryLimit: 3
suspend: false
```

We will change it to every 25 seconds and write to the resource:

```
  schedule: '*/25 * * * *'
  successfulJobsHistoryLimit: 3
  suspend: false
status: {}
:wq
```

Once we write it, we can see it was changed.

```
$ kubectl edit cronjob/my-existing-cron
cronjob.batch/my-existing-cron edited
```

If we want to use a different editor, we can override that by adding by using this **KUBE_EDITOR** syntax.

```
$ KUBE_EDITOR="nano" kubectl edit cronjob/my-existing-cron
```

Kubectl delete

So far, we have done everything short of removing it altogether, so that is what we are going to do next. The **cronjob** we just edited was one of two **cronjobs**, so now we're just going to delete the whole resource.

```
$ kubectl delete cronjob my-existing-cron
cronjob.batch "my-existing-cron" deleted
```

As a warning, you should never just delete something you don't know all the information about. Once the resource is

deleted, there's no recovering it; you will have to recreate it, so think twice before you run this command.

Kubectl apply

Earlier, I mentioned that some commands will require configuration files. The **apply** command allows you to apply configurations via files for resources within your cluster. This can also be done through **standard in (STDIN)** on the command line, but the suggestion is always by file.

I consider this command to be a little more advanced, as you need to know how to use your cluster and what kind of configuration file to apply to it. For this example, I am using the role-based access control (RBAC) config from Helm [4] for a service account.

```
$ kubectl apply -f commands.yaml
serviceaccount/tiller created
clusterrolebinding.rbac.authorization.k8s.io/tiller created
```

You can apply just about any configuration you want, but you will always need to know for sure what it is you're applying, or you may see unintended outcomes.

Troubleshooting Kubernetes with kubectl

Kubectl describe

Describe shows the details of the resource you're looking at. The most common use case is describing a pod or node to check if there's an error in the events, or if resources are too limited to use.

Resources you can **describe** include:

- Nodes
- Pods
- Services
- Deployments
- Replica sets
- Cronjobs

In this example, we can **describe** the **cronjob** currently in the cluster from our previous examples.

```
$ kubectl describe cronjob my-cron
```

Snippet:

```
Name:                          my-cron
Namespace:                     default
Labels:                        <none>
Annotations:                   <none>
Schedule:                      */5 * * * *
Concurrency Policy:            Allow
Suspend:                       False
Successful Job History Limit:  3
Failed Job History Limit:      1
Starting Deadline Seconds:     <unset>
Selector:                      <unset>
Parallelism:                   <unset>
Completions:                   <unset>
Pod Template:
  Labels:  <none>
  Containers:
   my-cron:
    Image:      busybox
    Port:       <none>
    Host Port:  <none>
```

Kubectl logs

While the **describe** command gives you the events occurring for the applications inside a pod, **logs** offer detailed insights into what's happening inside Kubernetes in relation to the pod. Understanding this distinction allows you to troubleshoot issues happening inside the application and inside Kubernetes because they are not always the same problem.

```
$ kubectl logs cherry-chart-88d49478c-dmcfv -n charts
```

Snippet:

```
172.17.0.1 - - [19/Apr/2020:16:01:15 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:20 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [/Apr/2020:16:01:25 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:30 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:35 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:40 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:45 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:50 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
172.17.0.1 - - [19/Apr/2020:16:01:55 +0000] "GET / HTTP/1.1"
               200 612 "-" "kube-probe/1.18" "-"
```

You can also remove extra noise or look for another event by using **grep** with this command. The **kube-probe** can be noisy, so let's filter it out with **grep**.

```
$ kubectl logs cherry-chart-88d49478c-dmcfv -n charts | grep
  -vie kube-probe
127.0.0.1 - - [10/Apr /2020:23:01:55 +0000] "GET / HTTP/1.1"
  200 612 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:75.0)
  Gecko/20100101 Firefox/75.0" "-"
```

Since some deployments have multiple containers within a pod, you can also use the **-c <container name>** with your logs to only look in one specific container for **logs**.

Kubectl exec

Much like the **docker exec** command, you can also **exec** into a container to troubleshoot an application directly. This is useful when the **logs** from the pod haven't provided you an answer to the issues you may be debugging. When using the **exec** command, the end of the line must always provide which shell you are using within the pod.

```
$ kubectl exec -it cherry-chart-88d49478c-dmcfv -n charts -- /
  bin/bash
root@cherry-chart-88d49478c-dmcfv:/#
```

Kubectl cp

This command is for copying files and directories to and from containers, much like the Linux cp command. It is not something you will use every day, but it is my personal favorite for pulling or restoring backups in an emergency when automation is failing.

Here's an example of copying a local file to a container. The syntax follows a **kubectl cp <filename> <namespace/podname:/path/tofile>** format:

```
$ kubectl cp commands_copy.txt charts/cherry-chart-88d49478c-
  dmcfv:commands.txt
$ kubectl exec -it cherry-chart-88d49478c-dmcfv -n charts -- /
  bin/bash
root@cherry-chart-88d49478c-dmcfv:/# ls
bin  boot  commands.txt  dev  etc  home  lib  lib64  media  mnt
  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

Here is another example, but this time pulling a file to our local machine from a container. The syntax is **kubectl cp <namespace/podname:/path/tofile>** format:

```
$ kubectl cp charts/cherry-chart-88d49478c-dmcfv:commands.txt
  commands_copy.txt
$ ls
commands_copy.txt
```

Download the kubectl cheat sheet

There are a lot of little commands that are helpful to have around as a Kubernetes administrator. I hope this cheat sheet comes in handy for you!

Links

[1] https://opensource.com/resources/what-is-kubernetes
[2] https://opensource.com/downloads/kubectl-cheat-sheet
[3] https://opensource.com/article/17/11/how-use-cron-linux
[4] https://github.com/venezia/k8s-helm/blob/master/docs/service_accounts.md

Kubectl Cheat Sheet

BY JESSICA CHERRY

kubectl is a powerful command-line tool to maintain your Kubernetes cluster. Here are commonly used commands to take you above and beyond average cluster administration.

Basic Commands

kubectl get

```
kubectl get <resource> --output wide
```

List all information about the select resource type. Common resources include:

- Pods (`kubectl get pods`)
- Namespaces (`kubectl get ns`)
- Nodes (`kubectl get node`)
- Deployments (`kubectl get deploy`)
- Service (`kubectl get svc`)
- ReplicaSets (`kubectl get rs`)

Call resources using singular (pod), plural (pods), or with shortcuts.

Get pod by namespace:

`-n, --namespace`

Wait for a resource to finish:

`-w, --watch`

Query multiple resources (comma-separated values):

```
kubectl get rs,services -o wide
```

kubectl edit

```
kubectl edit <resource-type>/<name>
```

Edit resources in a cluster. The default editor opens unless KUBE_EDITOR is specificed:

```
KUBE_EDITOR="nano" kubectl edit \
svc/container-registry
```

kubectl create

```
kubectl create --filename ./pod.yaml
```

Some resources require a name parameter. A small list of resources you can create include:

- Services (`svc`)
- Cronjobs (`cj`)
- Deployments (`deploy`)
- Quotas (`quota`)

See required parameters:

```
kubectl create cronjobs --help
```

kubectl delete

```
kubectl delete <resource>
```

Remove one more our resources by name, label, or by filename.

If you want to delete pods by label in mass you have to describe the pod and gather the app=”name” from the label section. This makes it easier to cycle multiple containers at once.

Add --grace-period=5 to give yourself a few seconds to cancel before deleting:

```
kubectl delete pod foo --grace-period=5
```

Kubectl Cheat Sheet

BY JESSICA CHERRY

kubectl is a powerful command-line tool to maintain your Kubernetes cluster. Here are commonly used commands to take you above and beyond average cluster administration.

Basic Commands

kubectl get

```
kubectl get <resource> --output wide
```

List all information about the select resource type. Common resources include:

- Pods (`kubectl get pods`)
- Namespaces (`kubectl get ns`)
- Nodes (`kubectl get node`)
- Deployments (`kubectl get deploy`)
- Service (`kubectl get svc`)
- ReplicaSets (`kubectl get rs`)

Call resources using singular (pod), plural (pods), or with shortcuts.

Get pod by namespace:

`-n, --namespace`

Wait for a resource to finish:

`-w, --watch`

Query multiple resources (comma-separated values):

```
kubectl get rs,services -o wide
```

kubectl edit

```
kubectl edit <resource-type>/<name>
```

Edit resources in a cluster. The default editor opens unless KUBE_EDITOR is specificed:

```
KUBE_EDITOR="nano" kubectl edit \
svc/container-registry
```

kubectl create

```
kubectl create --filename ./pod.yaml
```

Some resources require a name parameter. A small list of resources you can create include:

- Services (`svc`)
- Cronjobs (`cj`)
- Deployments (`deploy`)
- Quotas (`quota`)

See required parameters:

```
kubectl create cronjobs --help
```

kubectl delete

```
kubectl delete <resource>
```

Remove one more our resources by name, label, or by filename.

If you want to delete pods by label in mass you have to describe the pod and gather the app="name" from the label section. This makes it easier to cycle multiple containers at once.

Add --grace-period=5 to give yourself a few seconds to cancel before deleting:

```
kubectl delete pod foo --grace-period=5
```

Speed up administration of Kubernetes clusters with k9s

Check out this cool terminal UI for Kubernetes administration.

USUALLY, MY ARTICLES ABOUT KUBERNETES ADMINISTRATION are full of kubectl commands for administration for your clusters. Recently, however, someone pointed me to the k9s [1] project for a fast way to review and resolve day-to-day issues in Kubernetes. It's been a huge improvement to my workflow and I'll show you how to get started in this tutorial.

Installation can be done on a Mac, in Windows, and Linux. Instructions for each operating system can be found here [1]. Be sure to complete installation to be able to follow along.

I will be using Linux and Minikube, which is a lightweight way to run Kubernetes on a personal computer. Install it following this tutorial [2] or by using the documentation [3].

Setting the k9s configuration file

Once you've installed the k9s app, it's always good to start with the help command.

```
$ k9s help
K9s is a CLI to view and manage your Kubernetes clusters.

Usage:
  k9s [flags]
  k9s [command]

Available Commands:
  help        Help about any command
  info        Print configuration info
  version     Print version/build info

Flags:
  -A, --all-namespaces             Launch K9s in all namespaces
      --as string                  Username to impersonate or the
                                   operation
      --as-group stringArray       Group to impersonate for the
                                   operation
      --certificate-authority string Path to a cert file for the
                                   certificate authority
      --client-certificate string  Path to a client certificate file
                                   for TLS
      --client-key string          Path to a client key file for TLS
      --cluster string             The name of the kubeconfig
                                   cluster to use
  -c, --command string             Specify the default command to
                                   view when the application launches
      --context string             The name of the kubeconfig
                                   context to use
      --demo                       Enable demo mode to show
                                   keyboard commands
      --headless                   Turn K9s header off
  -h, --help                       help for k9s
      --insecure-skip-tls-verify   If true, the server's caCertFile
                                   will not be checked for validity
      --kubeconfig string          Path to the kubeconfig file to use
                                   for CLI requests
  -l, --logLevel string            Specify a log level (info, warn,
                                   debug, error, fatal, panic,
                                   trace) (default "info")
  -n, --namespace string           If present, the namespace scope
                                   for this CLI request
      --readonly                   Disable all commands that modify
                                   the cluster
  -r, --refresh int                Specify the default refresh rate
                                   as an integer (sec) (default 2)
      --request-timeout string     The length of time to wait
                                   before giving up on a single
                                   server request
      --token string               Bearer token for authentication to
                                   the API server
      --user string                The name of the kubeconfig user
                                   to use

Use "k9s [command] --help" for more information about a command.
```

As you can see, there is a lot of functionality we can configure with k9s. The only step we need to take place to get off the ground is to write a configuration file. The **info** command will point us to where the application is looking for it.

```
$ k9s info
 ____  __.________
|    |/ _/   __   \______
|      < \____    /  ___/
|    |  \   /    /\___ \
|____|__ \ /____//____  >
        \/            \/

Configuration:   /Users/jess/.k9s/config.yml
Logs:            /var/folders/5l/c1y1gcw97szdywgf9rk1100m0000gn/T/
                 k9s-jess.log
Screen Dumps:    /var/folders/5l/c1y1gcw97szdywgf9rk1100m0000gn/T/
                 k9s-screens-jess
```

To add a file, make the directory if it doesn't already exist and then add one.

```
$ mkdir -p ~/.k9s/
$ touch ~/.k9s/config.yml
```

For this introduction, we will use the default config.yml recommendations from the k9s repository. The maintainers note that this format is subject to change, so we can check here [4] for the latest version.

```
k9s:
  refreshRate: 2
  headless: false
  readOnly: false
  noIcons: false
  logger:
    tail: 200
    buffer: 500
    sinceSeconds: 300
    fullScreenLogs: false
    textWrap: false
    showTime: false
  currentContext: minikube
  currentCluster: minikube
  clusters:
    minikube:
      namespace:
        active: ""
        favorites:
        - all
        - kube-system
        - default
      view:
        active: dp
  thresholds:
    cpu:
      critical: 90
      warn: 70
    memory:
      critical: 90
      warn: 70
```

We set k9s to look for a local minikube configuration, so I'm going to confirm minikube is online and ready to go.

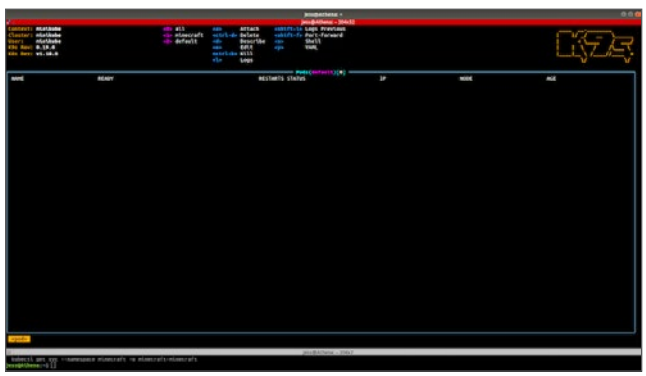
```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Running k9s to explore a Kubernetes cluster

With a configuration file set and pointing at our local cluster, we can now run the k9s command.

```
$ k9s
```

Once you start it up, the k9s text-based user interface (UI) will pop up. With no flag for a namespace, it will show you the pods in the default namespace.



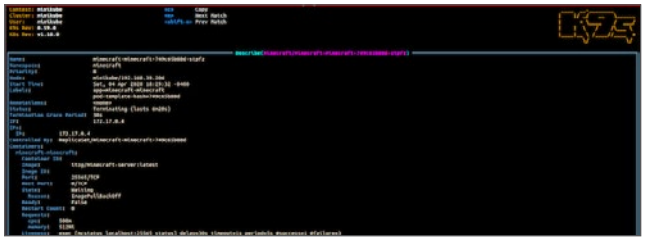
If you run in an environment with a lot of pods, the default view can be overwhelming. Alternatively, we can focus on a given namespace. Exit the application and run **k9s -n <namespace>** where *<namespace>* is an existing namespace. In the picture below, I ran **k9s -n minecraft**, and it shows my broken pod

So once you have k9s up and running, there are a bunch of things you can do quickly.

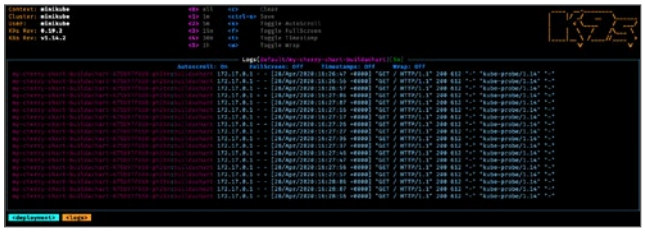
Navigating k9s happens through shortcut keys. We can always use arrow keys and the enter key to choose items listed. There are quite a few other universal keystrokes to navigate to different views:

- **0**—Show all pods in all namespaces

- **d**—Describe the selected pod



- **l**—Show logs for the selected pod pod

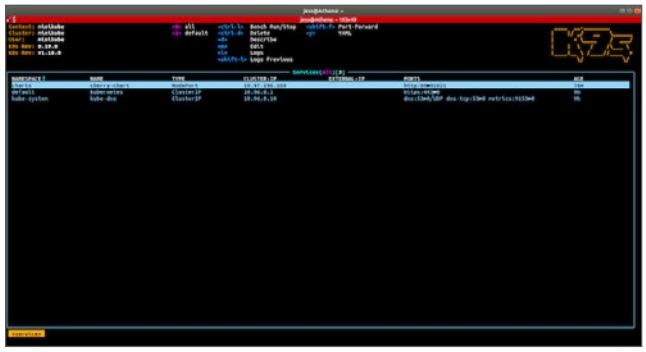


You may notice that k9s is set to use Vim command keys [5], including moving up and down using **J** and **K** keys. Good luck exiting, emacs users :)

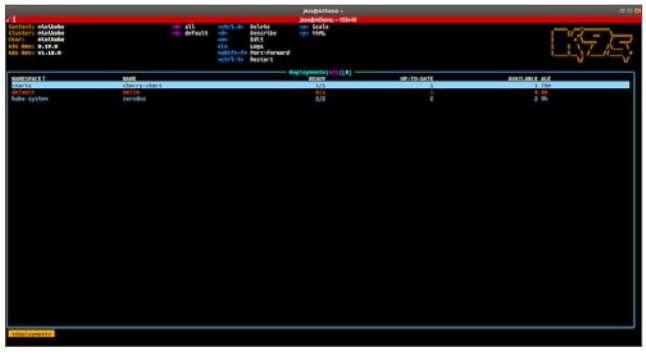
Viewing different Kubernetes resources quickly

Need to get to something that's not a pod? Yea I do too. There are a number of shortcuts that are available when we enter a colon (":") key. From there, you can use the following commands to navigate around there.

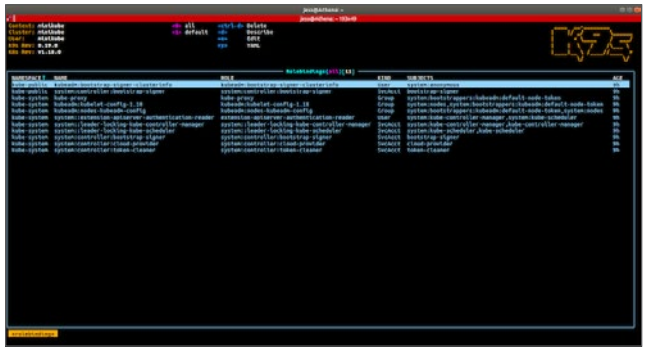
- **:svc**—Jump to a services view.



- **:deploy**—Jump to a deployment view.

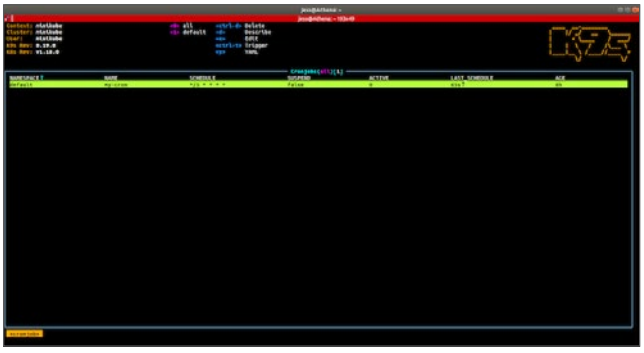


- **:rb**—Jump to a Rolebindings view for role-based access control (RBAC) [6] management.



- **:namespace**—Jump back to the namespaces view.

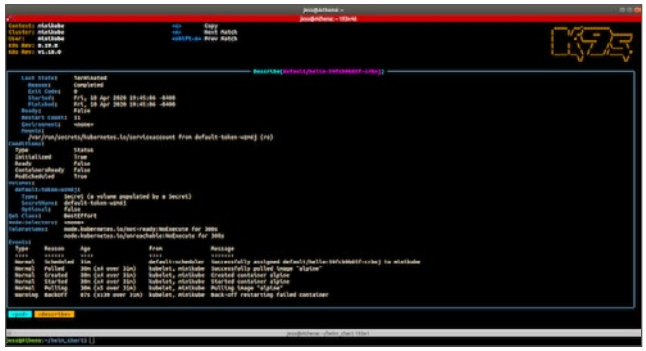
- **:cj**—Jump to the cronjobs view to see what jobs are scheduled in the cluster.



The most used tool for this application will be the keyboard; to go up or down on any page, use the arrow keys. If you need to quit, remember to use Vim keybindings. Type **:q** and hit enter to leave.

Example of troubleshooting Kubernetes with k9s

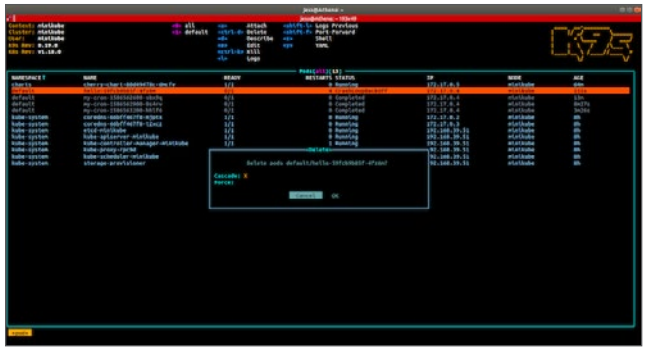
How does k9s help when something goes wrong? To walk through an example, I let several pods die due to misconfiguration. Below you can see my terrible hello deployment that's crashing. Once we highlight it, we press **d** to run a *describe* command to see what is causing the failure.



Skimming the events does not tell us a reason for the failure. Next, I hit the **esc** key and go check the logs by highlighting the pod and entering **<shift-l>**.

Unfortunately, the logs don't offer anything helpful either (probably because the deployment was never correctly configured), and the pod will not come up.

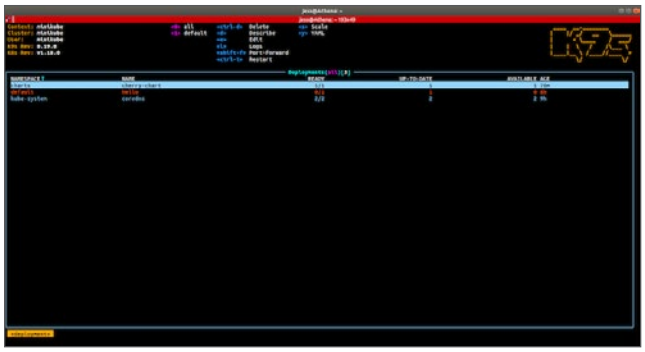
I then **esc** to step back out, and I will see if deleting the pod will take care of this issue. To do so, I highlight the pod and use **<ctrl-d>**. Thankfully, k9s prompts users before deletion.

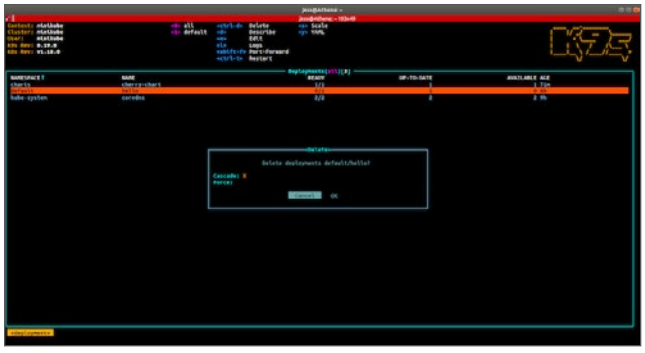


While I did delete the pod, the deployment resource still exists, so a new pod will come back up. It will also continue to restart and crash for whatever reason (we don't know yet).

Here is the point where I would repeat reviewing logs, describing resources, and use the **e** shortcut to even edit a running pod to troubleshoot the behavior. In this particular case, the failing pod is not configured to run in this environment. So let's delete the deployment to stop crash-then-reboot loop we are in.

We can get to deployments by typing **:deploy** and clicking enter. From there we highlight and press **<ctrl-d>** to delete.





And poof the deployment is gone! It only took a few keystrokes to clean up this failed deployment.

k9s is incredibly customizable

So this application has a ton of customization options, down to the color scheme of the UI. Here are a few editable options you may be interested in:

- Adjust where you put the config.yml file (so you can store it in version control [7])
- Add custom aliases [8] to an **alias.yml** file
- Create custom hotkeys [9] in a **hotkey.yml** file
- Explore available plugins [10] or write your own

The entire application is configured in YAML files, so customization will feel familiar to any Kubernetes administrator.

Simplify your life with k9s

I'm prone to administrating over my team's systems in a very manual way, more for brain training than anything else. When I first heard about k9s, I thought, "This is just lazy Kubernetes," so I dismissed it and went back to doing my manual intervention everywhere. I actually started using it daily while working through my backlog, and I was blown away at how much faster it was to use than kubectl alone. Now I'm a convert.

It's important to know your tools and master the "hard way" of doing something. It is also important to remember, as far as administration goes, it's important to work smarter, not harder. Using k9s is the way I live up to that objective. I guess we can call it lazy Kubernetes administration, and that's okay.

Links

[1] https://github.com/derailed/k9s
[2] https://opensource.com/article/18/10/getting-started-minikube
[3] https://kubernetes.io/docs/tasks/tools/install-minikube/
[4] https://github.com/derailed/k9s#k9s-configuration
[5] https://opensource.com/article/19/3/getting-started-vim
[6] https://kubernetes.io/docs/reference/access-authn-authz/rbac/
[7] https://opensource.com/article/19/3/move-your-dotfiles-version-control
[8] https://k9scli.io/topics/aliases/
[9] https://k9scli.io/topics/hotkeys/
[10] https://github.com/derailed/k9s/tree/master/plugins

Level up your use of Helm on Kubernetes with Charts

Configuring known apps using the Helm package manager.

APPLICATIONS ARE COMPLEX COLLECTIONS of code and configuration that have a lot of nuance to how they are installed. Like all open source software, they can be installed from source code, but most of the time users want to install something simply and consistently. That's why package managers exist in nearly every operating system, which manages the installation process.

Similarly, Kubernetes depends on package management to simplify the installation process. In this article, we'll be using the Helm package manager and its concept of stable charts to create a small application.

What is Helm package manager?

Helm [1] is a package manager for applications to be deployed to and run on Kubernetes. It is maintained by the Cloud Native Computing Foundation (CNCF) [2] with collaboration with the largest companies using Kubernetes. Helm can be used as a command-line utility, which I cover how to use here [3].

Installing Helm

Installing Helm is quick and easy for Linux and macOS. There are two ways to do this, you can go to the release page [4], download your preferred version, untar the file, and move the Helm executable to your **/usr/local/bin** or your **/usr/bin** whichever you are using.

Alternatively, you can use your operating system package manage (**dnf**, **snap**, **brew**, or otherwise) to install it. There are instructions on how to install on each OS on this GitHub page [5].

What are Helm Charts?

We want to be able to repeatably install applications, but also to customize them to our environment. That's where Helm Charts comes into play. Helm coordinates the deployment of applications using standardized templates called Charts. Charts are used to define, install, and upgrade your applications at any level of complexity.

> A *Chart* is a Helm package. It contains all of the resource definitions necessary to run an application, tool, or service inside of a Kubernetes cluster. Think of it like the Kubernetes equivalent of a Homebrew formula, an Apt dpkg, or a Yum RPM file.
>
> Using Helm [6]

Charts are quick to create, and I find them straightforward to maintain. If you have one that is accessible from a public version control site, you can publish it to the stable repository [7] to give it greater visibility. In order for a Chart to be added to stable, it must meet a number of technical requirements [8]. In the end, if it is considered properly maintained by the Helm maintain, it can then be published to Helm Hub [9].

Since we want to use the community-curated stable charts, we will make that easier by adding a shortcut:

```
$ helm repo add stable https://kubernetes-charts.storage.
  googleapis.com
"stable" has been added to your repositories
```

Running our first Helm Chart

Since I've already covered the basic Helm usage in this article [10], I'll focus on how to edit and use charts in this article. To follow along, you'll need Helm installed and access to some Kubernetes environment, like minikube (which you can walk through here [11] or here [12]).

Starting I will be picking one chart. Usually, in my article I use Jenkins as my example, and I would gladly do this if the chart wasn't really complex. This time I'll be using a basic chart and will be creating a small wiki, using mediawiki and its chart [13].

So how do I get this chart? Helm makes that as easy as a pull.

By default, charts are compressed in a .tgz file, but we can unpack that file to customize our wiki by using the **--untar** flag.

```
$ helm pull stable/mediawiki --untar
$ ls
mediawiki/
$ cd mediawiki/
$ ls
Chart.yaml          README.md       requirements.lock  templates/
OWNERS              charts/         requirements.yaml  values.yaml
```

Now that we have this we can begin customizing the chart.

Editing your Helm Chart

When the file was untared there was a massive amount of files that came out. While it does look frightening, there really is only one file we should be working with and that's the **values.yaml** file.

Everything that was unpacked was a list of template files that has all the information for the basic application configurations. All the template files actually depend on what is configured in the values.yaml file. Most of these templates and chart files actually are for creating service accounts in the cluster and the various sets of required application configurations that would usually be put together if you were to build this application on a regular server.

But on to the values.yaml file and what we should be changing in it. Open it in your favorite text editor or IDE. We see a YAML [14] file with a ton of configuration. If we zoom in just on the container image file, we see its repository, registry, and tags amongst other details.

```
## Bitnami DokuWiki image version
## ref: https://hub.docker.com/r/bitnami/mediawiki/tags/
##
image:
  registry: docker.io
  repository: bitnami/mediawiki
  tag: 1.34.0-debian-10-r31
  ## Specify a imagePullPolicy
  ## Defaults to 'Always' if image tag is 'latest', else set to
     'IfNotPresent'
  ## ref: http://kubernetes.io/docs/user-guide/images/
          #pre-pulling-images
  ##
  pullPolicy: IfNotPresent
  ## Optionally specify an array of imagePullSecrets.
  ## Secrets must be manually created in the namespace.
  ## ref: https://kubernetes.io/docs/tasks/configure-pod-
          container/pull-image-private-registry/
  ##
  # pullSecrets:
  #   - myRegistryKeySecretName
```

As you can see in the file each configuration for the values is well-defined. Our pull policy is set to **IfNotPresent**. This means if I run a **helm pull** command, it will not overwrite my existing version. If it's set to always, the image will default to the latest version of the image on every pull. I'll be using the default in this case, as in the past I have run into images being broken if it goes to the latest version without me expecting it (remember to version control your software, folks).

Customizing our Helm Chart

So let's configure this values file with some basic changes and make it our own. I'll be changing some naming conventions, the wiki username, and the mediawiki site name. *Note: This is another snippet from values.yaml. All of this customization happens in that one file.*

```
## User of the application
## ref: https://github.com/bitnami/bitnami-docker-
        mediawiki#environment-variables
##
mediawikiUser: cherrybomb

## Application password
## Defaults to a random 10-character alphanumeric string if not set
## ref: https://github.com/bitnami/bitnami-docker-
        mediawiki#environment-variables
##
# mediawikiPassword:

## Admin email
## ref: https://github.com/bitnami/bitnami-docker-
        mediawiki#environment-variables
##
mediawikiEmail: root@example.com

## Name for the wiki
## ref: https://github.com/bitnami/bitnami-docker-
        mediawiki#environment-variables
##
mediawikiName: Jess's Home of Helm
```

After this, I'll make some small modifications to our database name and user account. I changed the defaults to "jess" so you can see where changes were made.

```
externalDatabase:
 ## Database host
  host:

  ## Database port
  port: 3306

  ## Database user
  user: jess_mediawiki
```

```
  ## Database password
  password:

  ## Database name
  database: jess_mediawiki

##
## MariaDB chart configuration
##
## https://github.com/helm/charts/blob/master/stable/mariadb/
   values.yaml
##
mariadb:
 ## Whether to deploy a mariadb server to satisfy the applications
    database requirements. To use an external database set this to
    false and configure the externalDatabase parameters
  enabled: true
  ## Disable MariaDB replication
  replication:
    enabled: false

  ## Create a database and a database user
  ## ref: https://github.com/bitnami/bitnami-docker-mariadb/blob/
         master/README.md#creating-a-database-user-on-first-run
  ##
  db:
    name: jess_mediawiki
    user: jess_mediawiki
```

And finally, I'll be adding some ports in our load balancer to allow traffic from the local host. I'm running on minikube and find the **LoadBalancer** option works well.

```
service:
 ## Kubernetes svc type
  ## For minikube, set this to NodePort, elsewhere use LoadBalancer
  ##
  type: LoadBalancer
  ## Use serviceLoadBalancerIP to request a specific static IP,
  ## otherwise leave blank
  ##
  # loadBalancerIP:
  # HTTP Port
  port: 80
  # HTTPS Port
  ## Set this to any value (recommended: 443) to enable the https
service port
  # httpsPort: 443
  ## Use nodePorts to requets some specific ports when usin NodePort
  ## nodePorts:
  ##   http: <to set explicitly, choose port between 30000-32767>
  ##   https: <to set explicitly, choose port between 30000-32767>
  ##
  # nodePorts:
  #  http: "30000"
  #  https: "30001"
  ## Enable client source IP preservation
  ## ref http://kubernetes.io/docs/tasks/access-application-cluster/
     create-external-load-balancer/#preserving-the-client-source-ip
  ##
  externalTrafficPolicy: Cluster
```

Now that we have made the configurations to allow traffic and create the database, we know that we can go ahead and deploy our chart.

Deploy and enjoy!

Now that we have our custom version of the wiki, it's time to create a deployment. Before we get into that, let's first confirm that nothing else is installed with Helm, to make sure my cluster has available resources to run our wiki.

```
$ helm ls
NAME    NAMESPACE    REVISION    UPDATED STATUS  CHART    APP VERSION
```

There are no other deployments through Helm right now, so let's proceed with ours.

```
$ helm install jesswiki -f values.yaml stable/mediawiki
NAME: jesswiki
LAST DEPLOYED: Thu Mar  5 12:35:31 2020
NAMESPACE: default
STATUS: deployed
REVISION: 2
NOTES:
1. Get the MediaWiki URL by running:

  NOTE: It may take a few minutes for the LoadBalancer IP to be
        available.
        Watch the status with: 'kubectl get svc --namespace
        default -w jesswiki-mediawiki'

  export SERVICE_IP=$(kubectl get svc --namespace default
    jesswiki-mediawiki --template "{{ range (index .status.
    loadBalancer.ingress 0) }}{{.}}{{ end }}")
  echo "Mediawiki URL: http://$SERVICE_IP/"

2. Get your MediaWiki login credentials by running:

    echo Username: user
    echo Password: $(kubectl get secret --namespace default
jesswiki-mediawiki -o jsonpath="{.data.mediawiki-password}" |
  base64 --decode)
$
```

Perfect! Now we will navigate to the wiki, which is accessible at the cluster IP address. To confirm that address:

```
kubectl get svc --namespace default -w jesswiki-mediawiki
NAME                TYPE           CLUSTER-IP     EXTERNAL-IP PORT(S)      AGE
jesswiki-mediawiki LoadBalancer 10.103.180.70 <pending>   80:30220/TCP 17s
```

Now that we have the IP, we go ahead and check to see if it's up:

Now we have our new wiki up and running, and we can enjoy our new application with our personal edits. Use the command from the output above to get the password and start to fill in your wiki.

Conclusion

Helm is a powerful package manager that makes installing and uninstalling applications on top of Kubernetes as simple as a single command. Charts add to the experience by giving us curated and tested templates to install applications with our unique customizations. Keep exploring what Helm and Charts have to offer and let me know what you do with them in the comments.

Links

[1] https://helm.sh/
[2] https://www.cncf.io/
[3] https://opensource.com/article/20/2/kubectl-helm-commands
[4] https://github.com/helm/helm/releases/tag/v3.1.1
[5] https://github.com/helm/helm
[6] https://helm.sh/docs/intro/using_helm/
[7] https://github.com/helm/charts
[8] https://github.com/helm/charts/blob/master/CONTRIBUTING.md#technical-requirements
[9] https://artifacthub.io/
[10] https://opensource.com/article/20/2/kubectl-helm-commands
[11] https://opensource.com/article/18/10/getting-started-minikube
[12] https://opensource.com/article/19/7/security-scanning-your-devops-pipeline
[13] https://github.com/helm/charts/tree/master/stable/mediawiki
[14] https://en.wikipedia.org/wiki/YAML

How to make a Helm chart in 10 minutes

Write a simple Helm chart for Kubernetes in about 10 minutes.

A GOOD AMOUNT OF MY DAY-TO-DAY involves creating, modifying, and deploying Helm charts to manage the deployment of applications. Helm is an application package manager for Kubernetes, which coordinates the download, installation, and deployment of apps. Helm charts [1] are the way we can define an application as a collection of related Kubernetes resources.

So why would anyone use Helm? Helm makes managing the deployment of applications easier inside Kubernetes through a templated approach. All Helm charts follow the same structure while still having a structure flexible enough to represent any type of application you could run on Kubernetes. Helm also supports versioning since deployment needs are guaranteed to change with time. The alternative is to use multiple configuration files that you manually apply to your Kubernetes cluster to bring an application up. If we've learned anything from seeing infrastructure as code [2], it's that manual processes inevitably lead to errors. Helm charts give us a chance to apply that same lesson to the world of Kubernetes.

In this example, we'll be walking through using Helm with minikube, a single-node testing environment for Kubernetes. We will make a small Nginx web server application. For this example, I have minikube version 1.9.2 and Helm version 3.0.0 installed on my Linux laptop. To get set up, do the following.

- Download and configure minikube by following the excellent documentation here [3].
- Download and configure Helm using your favorite package manager listed here [4] or manually from the releases [5].

Create a Helm chart

Start by confirming we have the prerequisites installed:

```
$ which helm ## this can be in any folder as long as it returns
               in the path
/usr/local/bin/helm
$ minikube status ## if it shows Stopped, run `minikube start`
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

Starting a new Helm chart requires one simple command:

```
$ helm create mychartname
```

For the purposes of this tutorial, name the chart **buildachart**:

```
$ helm create buildachart
Creating buildachart
$ ls buildachart/
Chart.yaml   charts/      templates/   values.yaml
```

Examine the chart's structure

Now that you have created the chart, take a look at its structure to see what's inside. The first two files you see—**Chart.yaml** and **values.yaml**—define what the chart is and what values will be in it at deployment.

Look at **Chart.yaml**, and you can see the outline of a Helm chart's structure:

```
apiVersion: v2
name: buildachart
description: A Helm chart for Kubernetes

# A chart can be either an 'application' or a 'library' chart.
#
# Application charts are a collection of templates that can be
# packaged into versioned archives to be deployed.
#
# Library charts provide useful utilities or functions for the
# chart developer. They're included as a dependency of
# application charts to inject those utilities and functions
# into the rendering pipeline. Library charts do not define any
# templates and therefore cannot be deployed.
type: application
```

```
# This is the chart version. This version number should be
# incremented each time you make changes to the chart and its
# templates, including the app version.
version: 0.1.0

# This is the version number of the application being deployed.
# This version number should be incremented each time you make
# changes to the application.
appVersion: 1.16.0
```

The first part includes the API version that the chart is using (this is required), the name of the chart, and a description of the chart. The next section describes the type of chart (an application by default), the version of the chart you will deploy, and the application version (which should be incremented as you make changes).

The most important part of the chart is the template directory. It holds all the configurations for your application that will be deployed into the cluster. As you can see below, this application has a basic deployment, ingress, service account, and service. This directory also includes a test directory, which includes a test for a connection into the app. Each of these application features has its own template files under **templates/**:

```
$ ls templates/
NOTES.txt      _helpers.tpl      deployment.yaml      ingress.yaml
  service.yaml      serviceaccount.yaml  tests/
```

There is another directory, called **charts**, which is empty. It allows you to add dependent charts that are needed to deploy your application. Some Helm charts for applications have up to four extra charts that need to be deployed with the main application. When this happens, the **values** file is updated with the values for each chart so that the applications will be configured and deployed at the same time. This is a far more advanced configuration (which I will not cover in this introductory article), so leave the **charts/** folder empty.

Understand and edit values

Template files are set up with formatting that collects deployment information from the **values.yaml** file. Therefore, to customize your Helm chart, you need to edit the values file. By default, the **values.yaml** file looks like:

```
# Default values for buildachart.
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

image:
  repository: nginx
  pullPolicy: IfNotPresent

imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""

serviceAccount:
 # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using the
  # fullname template
  name:

podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000

service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
  annotations: {}
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  hosts:
    - host: chart-example.local
      paths: []
  tls: []
  #  - secretName: chart-example-tls
  #    hosts:
  #      - chart-example.local

resources: {}
  # We usually recommend not to specify default resources and to
  # leave this as a conscious choice for the user. This also
  # increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify
  # resources, uncomment the following lines, adjust them as
  # necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
```

```
  #   memory: 128Mi

nodeSelector: {}

tolerations: []

affinity: {}
```

Basic configurations

Beginning at the top, you can see that the **replicaCount** is automatically set to 1, which means that only one pod will come up. You only need one pod for this example, but you can see how easy it is to tell Kubernetes to run multiple pods for redundancy.

The **image** section has two things you need to look at: the **repository** where you are pulling your image and the **pullPolicy**. The pullPolicy is set to **IfNotPresent**; which means that the image will download a new version of the image if one does not already exist in the cluster. There are two other options for this: **Always**, which means it will pull the image on every deployment or restart (I always suggest this in case of image failure), and **Latest**, which will always pull the most up-to-date version of the image available. Latest can be useful if you trust your image repository to be compatible with your deployment environment, but that's not always the case.

Change the value to **Always**.

Before:

```
image:
  repository: nginx
  pullPolicy: IfNotPresent
```

After:

```
image:
  repository: nginx
  pullPolicy: Always
```

Naming and secrets

Next, take a look at the overrides in the chart. The first override is **imagePullSecrets**, which is a setting to pull a secret, such as a password or an API key you've generated as credentials for a private registry. Next are **nameOverride** and **fullnameOverride**. From the moment you ran **helm create**, its name (buildachart) was added to a number of configuration files—from the YAML ones above to the **templates/helper.tpl** file. If you need to rename a chart after you create it, this section is the best place to do it, so you don't miss any configuration files.

Change the chart's name using overrides.

Before:

```
imagePullSecrets: []
nameOverride: ""
fullnameOverride: ""
```

After:

```
imagePullSecrets: []
nameOverride: "cherry-awesome-app"
fullnameOverride: "cherry-chart"
```

Accounts

Service accounts provide a user identity to run in the pod inside the cluster. If it's left blank, the name will be generated based on the full name using the **helpers.tpl** file. I recommend always having a service account set up so that the application will be directly associated with a user that is controlled in the chart.

As an administrator, if you use the default service accounts, you will have either too few permissions or too many permissions, so change this.

Before:

```
serviceAccount:
 # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using
  # the fullname template
  Name:
```

After:

```
serviceAccount:
 # Specifies whether a service account should be created
  create: true
  # Annotations to add to the service account
  annotations: {}
  # The name of the service account to use.
  # If not set and create is true, a name is generated using
  # the fullname template
  Name: cherrybomb
```

Security

You can configure pod security to set limits on what type of filesystem group to use or which user can and cannot be used. Understanding these options is important to securing a Kubernetes pod, but for this example, I will leave this alone.

```
podSecurityContext: {}
  # fsGroup: 2000

securityContext: {}
  # capabilities:
  #   drop:
  #   - ALL
```

```
  # readOnlyRootFilesystem: true
  # runAsNonRoot: true
  # runAsUser: 1000
```

Networking

There are two different types of networking options in this chart. One uses a local service network with a **ClusterIP** address, which exposes the service on a cluster-internal IP. Choosing this value makes the service associated with your application reachable only from within the cluster (and through **ingress**, which is set to **false** by default). The other networking option is **NodePort**, which exposes the service on each Kubernetes node's IP address on a statically assigned port. This option is recommended for running minikube [6], so use it for this how-to.

Before:

```
service:
  type: ClusterIP
  port: 80

ingress:
  enabled: false
```

After:

```
service:
  type: NodePort
  port: 80

ingress:
  enabled: false
```

Resources

Helm allows you to explicitly allocate hardware resources. You can configure the maximum amount of resources a Helm chart can request and the highest limits it can receive. Since I'm using Minikube on a laptop, I'll set a few limits by removing the curly braces and the hashes to convert the comments into commands.

Before:

```
resources: {}
  # We usually recommend not to specify default resources and to
  # leave this as a conscious choice for the user. This also
  # increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify
  # resources, uncomment the following lines, adjust them as
  # necessary, and remove the curly braces after 'resources:'.
  # limits:
  #   cpu: 100m
  #   memory: 128Mi
  # requests:
  #   cpu: 100m
  #   memory: 128Mi
```

After:

```
resources:
  # We usually recommend not to specify default resources and
  # to leave this as a conscious choice for the user. This also
  # increases chances charts run on environments with little
  # resources, such as Minikube. If you do want to specify
  # resources, uncomment the following lines, adjust them as
  # necessary, and remove the curly braces after 'resources:'.
   limits:
     cpu: 100m
     memory: 128Mi
   requests:
     cpu: 100m
     memory: 128Mi
```

Tolerations, node selectors, and affinities

These last three values are based on node configurations. Although I cannot use any of them in my local configuration, I'll still explain their purpose.

nodeSelector comes in handy when you want to assign part of your application to specific nodes in your Kubernetes cluster. If you have infrastructure-specific applications, you set the node selector name and match that name in the Helm chart. Then, when the application is deployed, it will be associated with the node that matches the selector.

Tolerations, **tainting**, and **affinities** work together to ensure that pods run on separate nodes. Node affinity [7] is a property of *pods* that *attracts* them to a set of nodes (either as a preference or a hard requirement). Taints are the opposite—they allow a *node* to *repel* a set of pods.

In practice, if a node is tainted, it means that it is not working properly or may not have enough resources to hold the application deployment. Tolerations are set up as a key/value pair watched by the scheduler to confirm a node will work with a deployment.

Node affinity is conceptually similar to **nodeSelector**: it allows you to constrain which nodes your pod is eligible to be scheduled based on labels on the node. However, the labels differ because they match rules that apply to scheduling [8].

```
nodeSelector: {}

tolerations: []

affinity: {}
```

Deploy ahoy!

Now that you've made the necessary modifications to create a Helm chart, you can deploy it using a Helm command, add a name point to the chart, add a values file, and send it to a namespace:

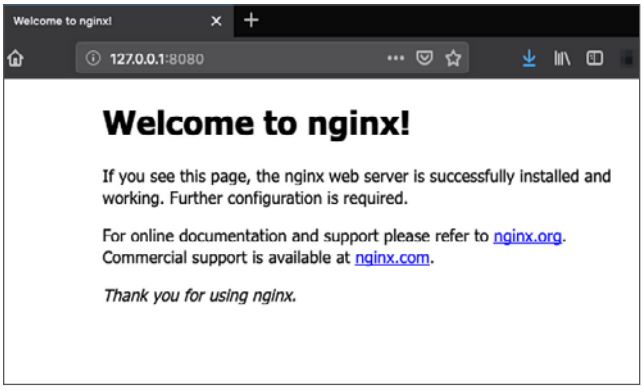
```
$ helm install my-cherry-chart buildachart/ --values
  buildachart/values.yaml
Release "my-cherry-chart" has been upgraded. Happy Helming!
```

The command's output will give you the next steps to connect to the application, including setting up port forwarding so you can reach the application from your localhost. To follow those instructions and connect to an Nginx load balancer:

```
$ export POD_NAME=$(kubectl get pods -l "app.kubernetes.io/
  name=buildachart,app.kubernetes.io/instance=my-cherry-chart"
  -o jsonpath="{.items[0].metadata.name}")
$ echo "Visit http://127.0.0.1:8080 to use your application"
Visit http://127.0.0.1:8080 to use your application
$ kubectl port-forward $POD_NAME 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
```

View the deployed application

To view your application, open your web browser:



Congratulations! You've deployed an Nginx web server by using a Helm chart!

There is a lot to learn as you explore what Helm charts can do. If you want to double-check your work, visit my example repository on GitHub [9].

Links

[1] https://helm.sh/docs/topics/charts/
[2] https://opensource.com/article/19/7/infrastructure-code
[3] https://kubernetes.io/docs/tasks/tools/install-minikube/
[4] https://github.com/helm/helm#install
[5] https://github.com/helm/helm/releases
[6] https://kubernetes.io/docs/setup/learning-environment/minikube/
[7] https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#node-affinity-beta-feature
[8] https://kubernetes.io/docs/concepts/configuration/assign-pod-node/#node-affinity
[9] https://github.com/Alynder/build_a_chart

Basic kubectl and Helm commands for beginners

Take a trip to the grocery store to shop for the commands you'll need to get started with these Kubernetes tools.

RECENTLY, my husband was telling me about an upcoming job interview where he would have to run through some basic commands on a computer. He was anxious about the interview, but the best way for him to learn and remember things has always been to equate the thing he doesn't know to something very familiar to him. Because our conversation happened right after I was roaming the grocery store trying to decide what to cook that evening, it inspired me to write about kubectl and Helm commands by equating them to an ordinary trip to the grocer.

Helm [1] is a tool to manage applications within Kubernetes. You can easily deploy charts with your application information, allowing them to be up and preconfigured in minutes within your Kubernetes environment. When you're learning something new, it's always helpful to look at chart examples to see how they are used, so if you have time, take a look at these stable charts [2].

Kubectl [3] is a command line that interfaces with Kubernetes environments, allowing you to configure and manage your cluster. It does require some configuration to work within environments, so take a look through the documentation [4] to see what you need to do.

I'll use namespaces in the examples, which you can learn about in my article Kubernetes namespaces for beginners [5].

Now that we have that settled, let's start shopping for basic kubectl and Helm commands!

Helm list

What is the first thing you do before you go to the store? Well, if you're organized, you make a **list**. LIkewise, this is the first basic Helm command I will explain.

In a Helm-deployed application, **list** provides details about an application's current release. In this example, I have one deployed application—the Jenkins CI/CD application. Running the basic **list** command always brings up the default namespace. Since I don't have anything deployed in the default namespace, nothing shows up:

```
$ helm list
NAME    NAMESPACE  REVISION    UPDATED    STATUS  CHART  APP VERSION
```

However, if I run the command with an extra flag, my application and information appear:

```
$ helm list --all-namespaces
NAME    NAMESPACE REVISION UPDATED                STATUS   CHART  APP VERSION
jenkins jenkins      1  2020-01-18 16:18:07 EST deployed jenkins-1.9.4 lts
```

Finally, I can direct the list command to check only the namespace I want information from:

```
$ helm list --namespace jenkins
NAME    NAMESPACE REVISION UPDATED                STATUS   CHART  APP VERSION
jenkins jenkins      1  2020-01-18 16:18:07 EST deployed jenkins-1.9.4  lts
```

Now that I have a list and know what is on it, I can go and get my items with **get** commands! I'll start with the Kubernetes cluster; what can I get from it?

Kubectl get

The **kubectl get** command gives information about many things in Kubernetes, including pods, nodes, and namespaces. Again, without a namespace flag, you'll always land in the default. First, I'll get the namespaces in the cluster to see what's running:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   53m
jenkins           Active   44m
kube-node-lease   Active   53m
```

```
kube-public      Active   53m
kube-system      Active   53m
```

Now that I have the namespaces running in my environment, I'll get the nodes and see how many are running:

```
$ kubectl get nodes
NAME       STATUS   ROLES       AGE   VERSION
minikube   Ready    master   55m   v1.16.2
```

I have one node up and running, mainly because my Minikube is running on one small server. To get the pods running on my one node:

```
$ kubectl get pods
No resources found in default namespace.
```

Oops, it's empty. I'll get what's in my Jenkins namespace with:

```
$ kubectl get pods --namespace jenkins
NAME                      READY   STATUS    RESTARTS   AGE
jenkins-7fc688c874-mh7gv  1/1     Running   0          40m
```

Good news! There's one pod, it hasn't restarted, and it has been running for 40 minutes. Well, since I know the pod is up, I want to see what I can get from Helm.

Helm get

Helm get is a little more complicated because this **get** command requires more than an application name, and you can request multiple things from applications. I'll begin by getting the values used to make the application, and then I'll show a snip of the **get all** action, which provides all the data related to the application.

```
$ helm get values jenkins -n jenkins
USER-SUPPLIED VALUES:
null
```

Since I did a very minimal stable-only install, the configuration didn't change. If I run the **all** command, I get everything out of the chart:

```
$ helm get all jenkins -n jenkins
```

```
jess@Athena:~$ helm get all jenkins -n jenkins
NAME: jenkins
LAST DEPLOYED: Sat Jan 18 16:18:07 2020
NAMESPACE: jenkins
STATUS: deployed
REVISION: 1
USER-SUPPLIED VALUES:
null

COMPUTED VALUES:
agent:
  TTYEnabled: false
  alwaysPullImage: false
  args: null
  command: null
  componentName: jenkins-slave
  containerCap: 10
  customJenkinsLabels: []
  enabled: true
  envVars: []
  idleMinutes: 0
  image: jenkins/jnlp-slave
  imagePullSecretName: null
  nodeSelector: {}
  podName: default
  podRetention: Never
  privileged: false
  resources:
    limits:
      cpu: 512m
      memory: 512Mi
    requests:
      cpu: 512m
      memory: 512Mi
  sideContainerName: jnlp
  tag: 3.27-1
  volumes: []
  yamlTemplate: ""
backup:
  annotations: {}
```

This produces a ton of data, so I always recommend keeping a copy of a Helm chart so you can look over the templates in the chart. I also create my own values to see what I have in place.

Now that I have all my goodies in my shopping cart, I'll check the labels that **describe** what's in them. These examples pertain only to kubectl, and they describe what I've deployed through Helm.

Kubectl describe

As I did with the **get** command, which can describe just about anything in Kubernetes, I'll limit my examples to namespaces, pods, and nodes. Since I know I'm working with one of each, this will be easy.

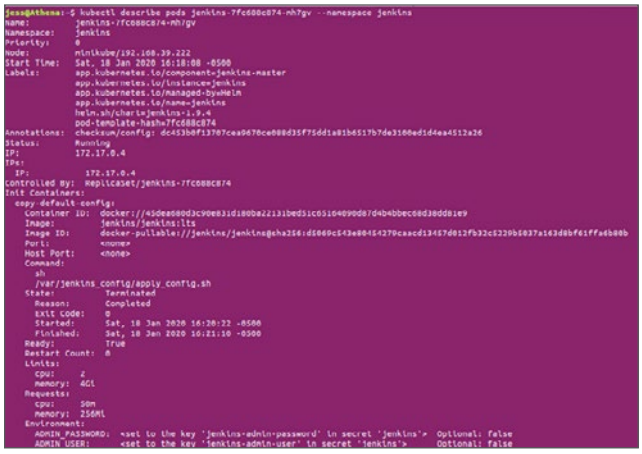
```
$ kubectl describe ns jenkins
Name:         jenkins
Labels:       <none>
Annotations:  <none>
Status:       Active
No resource quota.
No resource limits.
```

I can see my namespace's name and that it is active and has no resource nor quote limits.

The **describe pods** command produces a large amount of information, so I'll provide a small snip of the output. If you run the command without the pod name, it will return infor-

mation for all of the pods in the namespace, which can be overwhelming. So, be sure you always include the pod name with this command. For example:

```
$ kubectl describe pods jenkins-7fc688c874-mh7gv --namespace
  jenkins
```

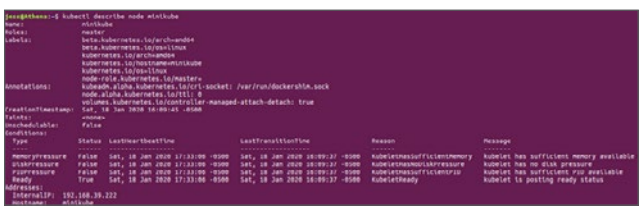


This provides (among many other things) the status of the container, how the container is managed, the label, and the image used in the pod. The data not in this abbreviated output includes resource requests and limits along with any conditions, init containers, and storage volume information applied in a Helm values file. This data is useful if your application is crashing due to inadequate resources, a configured init container that runs a prescript for configuration, or generated hidden passwords that shouldn't be in a plain text YAML file.

Finally, I'll use **describe node**, which (of course) describes the node. Since this example has just one, named Minikube, that is what I'll use; if you have multiple nodes in your environment, you must include the node name of interest.

As with pods, the node command produces an abundance of data, so I'll include just a snip of the output.

```
$ kubectl describe node minikube
```



Note that **describe node** is one of the more important basic commands. As this image shows, the command returns statistics that indicate when the node is running out of resources, and this data is excellent for alerting you when you need to scale up (if you do not have autoscaling in your environment). Other things not in this snippet of output include the percentages of requests made for all resources and limits, as well as the age and allocation of resources (e.g., for my application).

Checking out

With these commands, I've finished my shopping and gotten everything I was looking for. Hopefully, these basic commands can help you, too, in your day-to-day with Kubernetes.

I urge you to work with the command line often and learn the shorthand flags available in the Help sections, which you can access by running these commands:

```
$ helm --help
```

and

```
$ kubectl -h
```

Peanut butter and jelly

Some things just go together like peanut butter and jelly. Helm and kubectl are a little like that.

I often use these tools in my environment. Because they have many similarities in a ton of places, after using one, I usually need to follow up with the other. For example, I can do a Helm deployment and watch it fail using kubectl. Try them together, and see what they can do for you.

Links

[1] https://helm.sh/

[2] https://github.com/helm/charts/tree/master/stable

[3] https://kubernetes.io/docs/reference/kubectl/kubectl/

[4] https://kubernetes.io/docs/reference/kubectl/overview/

[5] https://opensource.com/article/19/12/kubernetes-namespaces

Create your first Knative app

Knative is a great way to get started quickly on serverless development with Kubernetes.

KNATIVE [1] is an open source community project that adds components to Kubernetes [2] for deploying, running, and managing serverless, cloud-native [3] applications. It enables more productive development with less interaction with Kubernetes' infrastructure.

There is a large amount of information out there about Knative, networking, and serverless deployments, and this introductory tutorial covers just a bite-size amount of it. In this walkthrough, I'll use Knative with Minikube [4] to create a Knative app—a simple container that prints messages in response to a curl command or in a web browser at a link provided by the deployment.

First, some background

Knative uses custom resource definitions (CRDs), a network layer, and a service core. For this walkthrough, I used Ubuntu 18.04, Kubernetes 1.19.0, Knative 0.17.2, and Kourier [5] 0.17.0 as the Knative networking layer, as well as the Knative command-line interface (CLI).

A CRD is a custom resource definition within Kubernetes. A resource is an endpoint in the Kubernetes API that stores a collection of API objects of a certain kind; for example, the built-in pod's resource contains a collection of pod objects. This allows an expansion of the Kubernetes API with new definitions. One example is the Knative serving core, which is defined to have internal autoscaling and rapid deployment of pods with the correct roles and access predefined.

Kourier is an Ingress [6] (a service to let in external network traffic) for Knative serving and a lightweight alternative for the Istio [7] ingress. Its deployment consists only of an Envoy proxy [8] and a control plane for it.

To understand the concepts in this tutorial, I recommend you are somewhat familiar with:

- Serverless, cloud-native applications
- Ingress with Envoy proxies, i.e., Istio
- DNS in Kubernetes
- Kubernetes patching configurations
- Custom resource definitions in Kubernetes
- Configuring YAML files for Kubernetes

Set up and installation

There are some prerequisites you must do before you can use Knative.

Configure Minikube

Before doing anything else, you must configure Minikube to run Knative locally in your homelab. Below are the configurations I suggest and the commands to set them:

```
$ minikube config set kubernetes-version v1.19.0
$ minikube config set memory 4000
$ minikube config set cpus 4
```

To make sure those configurations are set up correctly in your environment, run the Minikube commands to delete and start your cluster:

```
$ minikube delete
$ minikube start
```

Install the Knative CLI

You need the Knative CLI to make a deployment, and you need Go v1.14 [9] or later to work with the CLI. I created a separate directory to make it easier to find and install these tools. Use the following commands to set up the command line:

```
$ mkdir knative
$ cd knative/
$ git clone https://github.com/knative/client.git
$ cd client/
$ hack/build.sh -f
$ sudo cp kn /usr/local/bin
$ kn version
Version:        v20201018-local-40a84036
Build Date:   2020-10-18 20:00:37
Git Revision: 40a84036
Supported APIs:
* Serving
  - serving.knative.dev/v1 (knative-serving v0.18.0)
```

```
* Eventing
  - sources.knative.dev/v1alpha2 (knative-eventing v0.18.0)
  - eventing.knative.dev/v1beta1 (knative-eventing v0.18.0)
```

Once the CLI is installed, you can configure Knative in the Minikube cluster.

Install Knative

Since Knative is composed of CRDs, much of its installation uses YAML files with `kubectl` commands. To make this easier, set up some environment variables in the terminal so that you can get the needed YAML files a little faster and in the same version:

```
$ export KNATIVE="0.17.2"
```

First, apply the service resource definitions:

```
$ kubectl apply -f https://github.com/knative/serving/releases/
  download/v$KNATIVE/serving-crds.yaml
```

Then apply the core components to Knative:

```
$ kubectl apply -f https://github.com/knative/serving/releases/
  download/v$KNATIVE/serving-core.yaml
```

This deploys the services and deployments to the namespace `knative-serving`. You may have to wait a couple of moments for the deployment to finish.

To confirm the deployment finished, run the `kubectl` command to get the deployments from the namespace:

```
$ kubectl get deployments -n knative-serving
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
3scale-kourier-control  1/1     1            1           107m
activator               1/1     1            1           108m
autoscaler              1/1     1            1           108m
controller              1/1     1            1           108m
webhook                 1/1     1            1           108m
```

Install Kourier

Because you want to use a specific version and collect the correct YAML file, use another environment variable:

```
$ export KOURIER="0.17.0"
```

Then apply your networking layer YAML file:

```
$ kubectl apply -f https://github.com/knative/net-kourier/
  releases/download/v$KOURIER/kourier.yaml
```

You will find the deployment in the `kourier-system` namespace. To confirm the deployment is correctly up and functioning, use the `kubectl` command to get the deployments:

```
$ kubectl get deployments -n kourier-system
NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
3scale-kourier-gateway  1/1     1            1           110m
```

Next, configure the Knative serving to use Kourier as default. If you don't set this, the external networking traffic will not function. Set it with this `kubectl patch` command:

```
$ kubectl patch configmap/config-network \
  --namespace knative-serving \
  --type merge \
  --patch '{"data":{"ingress.class":"kourier.ingress.
           networking.knative.dev"}}'
```

Configure the DNS

Before you can access the load balancer, you need to run the `minikube tunnel` command in a separate terminal window. This command creates a route to services deployed with type `LoadBalancer` and sets their Ingress to their `ClusterIP`. Without this command, you will never get an `External-IP` from the load balancer. Your output will look like this:

```
Status:
    machine: minikube
    pid: 57123
    route: 10.96.0.0/12 -> 192.168.39.67
    minikube: Running
    services: [kourier]
        errors:
         minikube: no errors
         router: no errors
         loadbalancer emulator: no errors
Status:
    machine: minikube
    pid: 57123
    route: 10.96.0.0/12 -> 192.168.39.67
    minikube: Running
    services: [kourier]
        errors:
         minikube: no errors
         router: no errors
         loadbalancer emulator: no errors
```

Now that the services and deployments are complete, configure the DNS for the cluster. This enables your future deployable application to support DNS web addresses. To configure this, you need to get some information from your Kourier service by using the `kubectl get` command:

```
$ kubectl get service kourier -n kourier-system
NAME    TYPE           CLUSTER-IP     EXTERNAL-IP  PORT(S)
kourier LoadBalancer   10.103.12.15 10.103.12.15 80:32676/TCP,443:30558/TCP
```

Get the `CLUSTER-IP` address and save it for the next step. Next, configure the domain to determine your internal website on local DNS. (I ended mine in `nip.io`, and you can also use `xip.io`.) This requires another `kubectl patch` command:

```
$ kubectl patch configmap -n knative-serving config-domain -p
  "{\"data\": {\"10.103.12.15.nip.io\": \"\"}}"
```

Once it's patched, you will see this output:

```
configmap/config-domain patched
```

Use the Knative CLI

Now that your configurations are done, you can create an example application to see what happens.

Deploy a service

Earlier in this walkthrough, you installed the Knative CLI, which is used for Serving and Eventing resources in a Kubernetes cluster. This means you can deploy a sample application and manage services and routes. To bring up the command-line menu, type kn. Here is a snippet of the output:

```
$ kn
kn is the command line interface for managing Knative Serving
and Eventing resources

 Find more information about Knative at: https://knative.dev

Serving Commands:
  service       Manage Knative services
  revision      Manage service revisions
  route         List and describe service routes
```

Next, use the Knative CLI to deploy a basic "hello world" application with a web frontend. Knative provides some examples you can use; this one does a basic deployment:

```
kn service create hello --image gcr.io/knative-samples/helloworld-go
```

Your output should look something like this:

```
$ kn service create hello --image gcr.io/knative-samples/
  helloworld-go
Creating service 'hello' in namespace 'default':

  0.032s The Configuration is still working to reflect the latest
    desired specification.
  0.071s The Route is still working to reflect the latest
    desired specification.
  0.116s Configuration "hello" is waiting for a Revision to
    become ready.
 34.908s ...
 34.961s Ingress has not yet been reconciled.
 35.020s unsuccessfully observed a new generation
 35.208s Ready to serve.

Service 'hello' created to latest revision 'hello-dydlw-1' is
  available at URL:
http://hello.default.10.103.12.15.nip.io
```

This shows that the service was deployed with a URL into the namespace default. You can deploy to another namespace by running something like the following, then look at the output:

```
$ kn service create hello --image gcr.io/knative-samples/
  helloworld-go --namespace hello
Creating service 'hello' in namespace 'hello':

  0.015s The Configuration is still working to reflect the latest
    desired specification.
  0.041s The Route is still working to reflect the latest
    desired specification.
  0.070s Configuration "hello" is waiting for a Revision to
    become ready.
  5.911s ...
  5.958s Ingress has not yet been reconciled.
  6.043s unsuccessfully observed a new generation
  6.213s Ready to serve.

Service 'hello' created to latest revision 'hello-wpbwj-1' is
  available at URL:
http://hello.hello.10.103.12.15.nip.io
```

Test your new deployment

Check to see if the new service you deployed is up and running. There are two ways to check:

1. Check your web address in a browser
2. Run a `curl` command to see what returns

If you check the address in a web browser, you should see something like this:

(Jess Cherry, CC BY-SA 4.0)

Good! It looks like your application's frontend is up!

Next, test the `curl` command to confirm everything works from the command line. Here is an example of a `curl` to my application and the output:

```
$ curl http://hello.default.10.103.12.15.nip.io
Hello World!
```

Interact with the Knative app

From here, you can use the Knative CLI to make some basic changes and test the functionality. Describe the service and check the output:

```
$ kn service describe hello
Name:          hello
Namespace:  default
Age:           12h
URL:           http://hello.default.10.103.12.15.nip.io

Revisions:
  100%  @latest (hello-dydlw-1) [1] (12h)
        Image: gcr.io/knative-samples/helloworld-go (pinned to 5ea96b)

Conditions:
  OK TYPE                   AGE REASON
  ++ Ready                  12h
  ++ ConfigurationsReady    12h
  ++ RoutesReady            12h
```

It looks like everything is up and ready as you configured it. Some other things you can do with the Knative CLI (which won't show up now due to the minimal configuration in this example) are to describe and list the routes with the app:

```
$ kn route describe hello
Name:          hello
Namespace:  default
Age:           12h
URL:           http://hello.default.10.103.12.15.nip.io
Service:       hello

Traffic Targets:
  100%  @latest (hello-dydlw-1)

Conditions:
  OK TYPE                     AGE REASON
  ++ Ready                    12h
  ++ AllTrafficAssigned       12h
  ++ CertificateProvisioned   12h TLSNotEnabled
  ++ IngressReady             12h
jess@Athena:~/knative/client$ kn route list hello
NAME    URL                                   READY
hello   http://hello.default.10.103.12.15.nip.io   True
```

This can come in handy later when you need to troubleshoot issues with your deployments.

Clean up

Just as easily as you deployed your application, you can clean it up:

```
$ kn service delete hello
Service 'hello' successfully deleted in namespace 'default'.

jess@Athena:~/knative/client$ kn service delete hello
            --namespace hello
Service 'hello' successfully deleted in namespace 'hello'.
```

Make your own app

This walkthrough used an existing Knative example, but you are probably wondering about making something that *you* want. You are right, so I'll provide this example YAML then explain how you can apply it with kubectl and manage it with the Knative CLI.

Example YAML

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld
  namespace: default
spec:
 template:
  spec:
   containers:
     - image: gcr.io/knative-samples/helloworld-go
       ports:
             - containerPort: 8080
       env:
        - name: TARGET
          value: "This is my app"
```

Save this to `apps.yaml`, and then you can make changes to some things. For example, you can change your `metadata`, `name`, and `namespace`. You can also change the value of the target (which I set to `This is my app`) so that, rather than `Hello World`, you'll see a new message that says `Hello ${TARGET} !` when you deploy the file.

To deploy a file like this, you will have to use `kubectl apply -f apps.yaml`.

First, deploy your new service using the `apply` command:

```
$ kubectl apply -f apps.yaml
service.serving.knative.dev/helloworld created
```

Next, you can describe your new deployment, which is the name provided in the YAML file:

```
$ kn service describe helloworld
Name:          helloworld
Namespace:  default
Age:           50s
URL:           http://helloworld.default.10.103.12.15.nip.io
```

```
Revisions:
  100%  @latest (helloworld-qfr9s) [1] (50s)
        Image:  gcr.io/knative-samples/helloworld-go (at 5ea96b)

Conditions:
  OK TYPE                  AGE REASON
  ++ Ready                 43s
  ++ ConfigurationsReady   43s
  ++ RoutesReady           43s
```

Run a curl command to confirm it produces the new output you defined in your YAML file:

```
$ curl  http://helloworld.default.10.103.12.15.nip.io
Hello This is my app!
```

Double-check by going to the simple web frontend.

(Jess Cherry, CC BY-SA 4.0)

This proves your application is running! Congratulations!

Final thoughts

Knative is a great way for developers to move quickly on serverless development with networking services that allow users to see changes in apps immediately. It is fun to play with and lets you take a deeper dive into serverless and other exploratory uses of Kubernetes!

Links

[1] https://knative.dev/
[2] https://opensource.com/resources/what-is-kubernetes
[3] https://en.wikipedia.org/wiki/Cloud_native_computing
[4] https://minikube.sigs.k8s.io/docs/
[5] https://github.com/knative-sandbox/net-kourier
[6] https://kubernetes.io/docs/concepts/services-networking/ingress/
[7] https://istio.io/
[8] https://www.envoyproxy.io/
[9] https://golang.org/doc/install

A step-by-step guide to Knative eventing

Knative eventing is a way to create, send, and verify events in your cloud-native environment.

IN A PREVIOUS ARTICLE, I covered how to create a small app with Knative [1], which is an open source project that adds components to Kubernetes [2] for deploying, running, and managing serverless, cloud-native [3] applications. In this article, I'll explain Knative eventing, a way to create, send, and verify events in your cloud-native environment.

Events can be generated from many sources in your environment, and they can be confusing to manage or define. Since Knative follows the CloudEvents [4] specification, it allows you to have one common abstraction point for your environment, where the events are defined to one specification.

This article explains how to install Knative eventing version 0.20.0 and create, trigger, and verify events. Because there are many steps involved, I suggest you look at my GitHub repo [5] to walk through this article with the files.

Set up your configuration

This walkthrough uses Minikube [6] with Kubernetes 1.19.0. It also makes some configuration changes to the Minikube environment.

Minikube pre-configuration commands:

```
$ minikube config set kubernetes-version v1.19.0
$ minikube config set memory 4000
$ minikube config set cpus 4
```

Before starting Minikube, run the following commands to make sure your configuration stays and start Minikube:

```
$ minikube delete
$ minikube start
```

Install Knative eventing

Install the Knative eventing custom resource definitions (CRDs) using kubectl. The following shows the command and a snippet of the output:

```
$ kubectl apply --filename https://github.com/knative/eventing/
  releases/download/v0.20.0/eventing-crds.yaml

customresourcedefinition.apiextensions.k8s.io/apiserversources.
  sources.knative.dev created
customresourcedefinition.apiextensions.k8s.io/brokers.eventing.
  knative.dev created
customresourcedefinition.apiextensions.k8s.io/channels.
  messaging.knative.dev created
customresourcedefinition.apiextensions.k8s.io/triggers.eventing.
  knative.dev created
```

Next, install the core components using kubectl:

```
$ kubectl apply --filename https://github.com/knative/eventing/
  releases/download/v0.20.0/eventing-core.yaml
namespace/knative-eventing created
serviceaccount/eventing-controller created
clusterrolebinding.rbac.authorization.k8s.io/eventing-controller
  created
```

Since you're running a standalone version of the Knative eventing service, you must install the in-memory channel to pass events. Using kubectl, run:

```
$ kubectl apply --filename https://github.com/knative/eventing/
  releases/download/v0.20.0/in-memory-channel.yaml
```

Install the broker, which utilizes the channels and runs the event routing:

```
$ kubectl apply --filename https://github.com/knative/eventing/
  releases/download/v0.20.0/mt-channel-broker.yaml
clusterrole.rbac.authorization.k8s.io/knative-eventing-mt-
  channel-broker-controller created
clusterrole.rbac.authorization.k8s.io/knative-eventing-mt-
  broker-filter created
```

Next, create a namespace and add a small broker to it; this broker routes events to triggers. Create your namespace using kubectl:

```
$ kubectl create namespace eventing-test
namespace/eventing-test created
```

Now create a small broker named default in your namespace. The following is the YAML from my **broker.yaml** file (which can be found in my GitHub repository):

```
apiVersion: eventing.knative.dev/v1
kind: broker
metadata:
  name: default
  namespace: eventing-test
```

Then apply your broker file using kubectl:

```
$ kubectl create -f broker.yaml
   broker.eventing.knative.dev/default created
```

Verify that everything is up and running (you should see the confirmation output) after you run the command:

```
$ kubectl -n eventing-test get broker default
NAME     URL                                          AGE  READY  REASON
default  http://broker-ingress.knative-eventing.
         svc.cluster.local/eventing-test/default      3m6s True
```

You'll need this URL from the broker output later for sending events, so save it.

Create event consumers

Now that everything is installed, you can start configuring the components to work with events.

First, you need to create event consumers. You'll create two consumers in this walkthrough: **hello-display** and **goodbye-display**. Having two consumers allows you to see how to target a consumer per event message.

The hello-display YAML code:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-display
spec:
  replicas: 1
  selector:
    matchLabels: &labels
      app: hello-display
  template:
    metadata:
      labels: *labels
    spec:
      containers:
        - name: event-display
          image: gcr.io/knative-releases/knative.dev/eventing-
                 contrib/cmd/event_display

---

kind: Service
apiVersion: v1
metadata:
  name: hello-display
spec:
  selector:
    app: hello-display
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

The goodbye-display YAML code:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: goodbye-display
spec:
  replicas: 1
  selector:
    matchLabels: &labels
      app: goodbye-display
  template:
    metadata:
      labels: *labels
    spec:
      containers:
        - name: event-display
          # Source code: https://github.com/knative/eventing-
                         contrib/tree/master/cmd/event_display
          image: gcr.io/knative-releases/knative.dev/eventing-
contrib/cmd/event_display

---

kind: Service
apiVersion: v1
metadata:
  name: goodbye-display
spec:
  selector:
    app: goodbye-display
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

The differences in the YAML between the two consumers are in the `app` and `metadata name` sections. While both consumers are on the same ports, you can target one when generating an event. Create the consumers using kubectl:

```
$ kubectl -n eventing-test apply -f hello-display.yaml
deployment.apps/hello-display created
service/hello-display created

$ kubectl -n eventing-test apply -f goodbye-display.yaml
deployment.apps/goodbye-display created
service/goodbye-display created
```

Check to make sure the deployments are running after you've applied the YAML files:

```
$ kubectl -n eventing-test get deployments hello-display
goodbye-display
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
hello-display     1/1     1            1           2m4s
goodbye-display   1/1     1            1           34s
```

Create triggers

Now, you need to create the triggers, which define the events the consumer receives. You can define triggers to use any filter from your cloud events. The broker receives events from the trigger and sends the events to the correct consumer. This set of examples creates two triggers with different definitions. For example, you can send events with the attribute type `greeting` to the `hello-display` consumer.

The greeting-trigger.yaml code:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: hello-display
spec:
  broker: default
  filter:
    attributes:
      type: greeting
  subscriber:
    ref:
     apiVersion: v1
     kind: Service
     name: hello-display
```

To create the first trigger, apply your YAML file:

```
$ kubectl -n eventing-test apply -f greeting-trigger.yaml
trigger.eventing.knative.dev/hello-display created
```

Next, make the second trigger using **sendoff-trigger.yaml**. This sends anything with the attribute `source sendoff` to your `goodbye-display` consumer.

The sendoff-trigger.yaml code:

```
apiVersion: eventing.knative.dev/v1
kind: Trigger
metadata:
  name: goodbye-display
spec:
  broker: default
  filter:
    attributes:
      source: sendoff
  subscriber:
    ref:
      apiVersion: v1
      kind: Service
      name: goodbye-display
```

Next, apply your second trigger definition to the cluster:

```
$ kubectl -n eventing-test apply -f sendoff-trigger.yaml
trigger.eventing.knative.dev/goodbye-display created
```

Confirm everything is correctly in place by getting your triggers from the cluster using kubectl:

```
$ kubectl -n eventing-test get triggers
NAME            BROKER  SUBSCRIBER_URI                          AGE   READY
goodbye-display default http://goodbye-display.eventing-test.
                        svc.cluster.local/                      24s   True
hello-display   default http://hello-display.eventing-test.
                        svc.cluster.local/                      46s   True
```

Create an event producer

Create a pod you can use to send events. This is a simple pod deployment with curl and SSH access for you to send events using curl [7]. Because the broker can be accessed only from inside the cluster where Knative eventing is installed, the pod needs to be in the cluster; this is the only way to send events into the cluster. Use the **event-producer.yaml** file with this code:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: curl
  name: curl
spec:
  containers:
    - image: radial/busyboxplus:curl
```

```
    imagePullPolicy: IfNotPresent
    name: curl
    resources: {}
    stdin: true
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    tty: true
```

Next, deploy the pod by using kubectl:

```
$ kubectl -n eventing-test apply -f event-producer.yaml
pod/curl created
```

To verify, get the deployment and make sure the pod is up and running:

```
$ kubectl get pods -n eventing-test
NAME                          READY   STATUS    RESTARTS   AGE
curl                          1/1     Running   0          8m13s
```

Send some events

Since this article has been so configuration-heavy, I imagine you'll be happy to finally be able to send some events and test out your services. Events have to be passed internally in the cluster. Usually, events would be defined around applications internal to the cluster and come from those applications. But this example will manually send events from your pod named **curl**.

Begin by logging into the pod:

```
$ kubectl -n eventing-test attach curl -it
```

Once logged in, you'll see output similar to:

```
Defaulting container name to curl.
Use 'kubectl describe pod/curl -n eventing-test' to see all of
  the containers in this pod.
If you don't see a command prompt, try pressing enter.
[ root@curl:/ ]$
```

Now, generate an event using curl. This needs some extra definitions and requires the broker URL generated during the installation. This example sends a greeting to the broker:

```
curl -v "http://broker-ingress.knative-eventing.svc.cluster.
local/eventing-test/default" \
  -X POST \
  -H "Ce-Id: say-hello" \
  -H "Ce-Specversion: 1.0" \
  -H "Ce-Type: greeting" \
  -H "Ce-Source: not-sendoff" \
  -H "Content-Type: application/json" \
  -d '{"msg":"Hello Knative!"}'
```

`Ce` is short for CloudEvent, which is the standardized CloudEvents specification [8] that Knative follows. You also need to know the event ID (this is useful to verify it was delivered), the type, the source (which must specify that it is not a `sendoff` so that it doesn't go to the source defined in the sendoff trigger), and a message.

When you run the command, this should be the output (and you should receive a 202 Accepted [9] response):

```
> POST /eventing-test/default HTTP/1.1
> User-Agent: curl/7.35.0
> Host: broker-ingress.knative-eventing.svc.cluster.local
> Accept: */*
> Ce-Id: say-hello
> Ce-Specversion: 1.0
> Ce-Type: greeting
> Ce-Source: not-sendoff
> Content-Type: application/json
> Content-Length: 24
>
< HTTP/1.1 202 Accepted
< Date: Sun, 24 Jan 2021 22:25:25 GMT
< Content-Length: 0
```

The 202 means the trigger sent it to the **hello-display** consumer (because of the definition.)

Next, send a second definition to the **goodbye-display** consumer with this new curl command:

```
curl -v "http://broker-ingress.knative-eventing.svc.cluster.
         local/eventing-test/default" \
  -X POST \
  -H "Ce-Id: say-goodbye" \
  -H "Ce-Specversion: 1.0" \
  -H "Ce-Type: not-greeting" \
  -H "Ce-Source: sendoff" \
  -H "Content-Type: application/json" \
  -d '{"msg":"Goodbye Knative!"}'
```

This time, it is a `sendoff` and not a greeting based on the previous setup section's trigger definition. It is directed to the **goodbye-display** consumer.

Your output should look like this, with another 202 returned:

```
> POST /eventing-test/default HTTP/1.1
> User-Agent: curl/7.35.0
> Host: broker-ingress.knative-eventing.svc.cluster.local
> Accept: */*
> Ce-Id: say-goodbye
> Ce-Specversion: 1.0
> Ce-Type: not-greeting
> Ce-Source: sendoff
> Content-Type: application/json
```

```
> Content-Length: 26
>
< HTTP/1.1 202 Accepted
< Date: Sun, 24 Jan 2021 22:33:00 GMT
< Content-Length: 0
```

Congratulations, you sent two events!

Before moving on to the next section, exit the pod by typing **exit**.

Verify the events

Now that the events have been sent, how do you know that the correct consumers received them? By going to each consumer and verifying it in the logs.

Start with the **hello-display** consumer::

```
$ kubectl -n eventing-test logs -l app=hello-display --tail=100
```

There isn't much running in this example cluster, so you should see only one event:

```
cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: greeting
  source: not-sendoff
  id: say-hello
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T22:25:25.760867793Z
Data,
  {
    "msg": "Hello Knative!"
  }
```

You've confirmed the **hello-display** consumer received the event! Now check the **goodbye-display** consumer and make sure the other message made it.

Start by running the same command but with **goodbye-display**:

```
$ kubectl -n eventing-test logs -l app=goodbye-display
  --tail=100
cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: not-greeting
  source: sendoff
  id: say-goodbye
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T22:33:00.515716701Z
Data,
  {
    "msg": "Goodbye Knative!"
  }
```

It looks like both events made it to their proper locations. Congratulations—you have officially worked with Knative eventing!

Bonus round: Send an event to multiple consumers

So you sent events to each consumer using curl, but what if you want to send an event to both consumers? This uses a similar curl command but with some interesting changes. In the previous triggers, each one was defined with a different attribute. The greeting trigger had attribute `type`, and sendoff trigger had attribute source. This means you can make a curl call and send it to both consumers.

Here is a curl example of a definition for sending an event to both consumers:

```
curl -v "http://broker-ingress.knative-eventing.svc.cluster.
local/eventing-test/default" \
  -X POST \
  -H "Ce-Id: say-hello-goodbye" \
  -H "Ce-Specversion: 1.0" \
  -H "Ce-Type: greeting" \
  -H "Ce-Source: sendoff" \
  -H "Content-Type: application/json" \
  -d '{"msg":"Hello Knative! Goodbye Knative!"}'
```

As you can see, the definition of this curl command changed to set the `source` for **goodbye-display** and the `type` for **hello-display**.

Here is sample output of what the events look like after they are sent.

Output of the event being sent:

```
> POST /eventing-test/default HTTP/1.1
> User-Agent: curl/7.35.0
> Host: broker-ingress.knative-eventing.svc.cluster.local
> Accept: */*
> Ce-Id: say-hello-goodbye
> Ce-Specversion: 1.0
> Ce-Type: greeting
> Ce-Source: sendoff
> Content-Type: application/json
> Content-Length: 41
>
< HTTP/1.1 202 Accepted
< Date: Sun, 24 Jan 2021 23:04:15 GMT
< Content-Length: 0
```

Output of hello-display (showing two events):

```
$ kubectl -n eventing-test logs -l app=hello-display --tail=100
☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: greeting
  source: not-sendoff
  id: say-hello
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T22:25:25.760867793Z
Data,
  {
    "msg": "Hello Knative!"
  }
☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: greeting
  source: sendoff
  id: say-hello-goodbye
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T23:04:15.036352685Z
Data,
  {
    "msg": "Hello Knative! Goodbye Knative!"
  }
```

Output of goodbye-display (also with two events):

```
$ kubectl -n eventing-test logs -l app=goodbye-display --tail=100
☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: not-greeting
  source: sendoff
  id: say-goodbye
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T22:33:00.515716701Z
Data,
  {
    "msg": "Goodbye Knative!"
  }
  ☁ cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 1.0
  type: greeting
  source: sendoff
  id: say-hello-goodbye
  datacontenttype: application/json
Extensions,
  knativearrivaltime: 2021-01-24T23:04:15.036352685Z
Data,
  {
    "msg": "Hello Knative! Goodbye Knative!"
  }
```

As you can see, the event went to both consumers based on your curl definition. If an event needs to be sent to more than one place, you can write definitions to send it to more than one consumer.

Give it a try!

Internal eventing in cloud events is pretty easy to track if it's going to a predefined location of your choice. Enjoy seeing how far you can go with eventing in your cluster!

Links

[1] https://opensource.com/article/20/11/knative
[2] https://opensource.com/resources/what-is-kubernetes
[3] https://en.wikipedia.org/wiki/Cloud_native_computing
[4] https://cloudevents.io/
[5] https://github.com/Alynder/knative_eventing
[6] https://minikube.sigs.k8s.io/docs/
[7] https://www.redhat.com/sysadmin/use-curl-api
[8] https://github.com/cloudevents/spec
[9] https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/202

5 interview questions every Kubernetes job candidate should know

If you're interviewing people for Kubernetes-related roles, here's what to ask and why it matters.

JOB INTERVIEWS are hard for people on both sides of the table, but I've discovered that interviewing candidates for Kubernetes-related jobs has seemed especially hard lately. Why, you ask? For one thing, it's hard to find someone who can answer some of my questions. Also, it has been hard to confirm whether they have the right experience, regardless of their answers to my questions.

I'll skip over my musings on that topic and get to some questions that you should ask of any job candidate who would be working with Kubernetes [1].

What is Kubernetes?

I've always found this question to be one of the best ones to ask in interviews. I always hear, "I work with Kubernetes," but when I ask, "what is it?" I never get a confident answer.

My favorite answer is from Chris Short [2]: "Just an API with some YAML files."

While he is not wrong, I'll give you a more detailed version. Kubernetes is a portable container orchestration tool that is used to automate the tasks of managing, monitoring, scaling, and deploying containerized applications.

I've found that "an orchestration tool for deploying containerized applications" is probably as good as you're going to get as an answer, which in my opinion is good enough. While many believe Kubernetes adds a great deal more, overall, it offers many APIs to add to this core feature: container orchestration.

In my opinion, this is one of the best questions you can ask in an interview, as it at least proves whether the candidate knows what Kubernetes is.

What is the difference between a Kubernetes node and a pod?

This question reveals a great first look at the complexity of Kubernetes. It shifts the conversation to an architectural overview and can lead to many interesting follow-up details. It has also been explained incorrectly to me an innumerable amount of times.

A node [3] is the worker machine. This machine can be a virtual machine (VM) or a physical machine, depending on whether you are running on a hypervisor or on bare metal. The node contains services to run containers, including the kubelet, kube-proxy, and container runtime.

A pod [4] includes (1) one or more containers (2) with shared network (3) and storage (4) and the specification on how to run the containers deployed together. All four of these details are important. For bonus points, an applicant could mention that, technically, a pod is the smallest deployable unit Kubernetes can create and manage—not a container.

The best short answer I've received for this question is: "The node is the worker, and the pod is the thing the containers are in." The distinction matters. Most of a Kubernetes administrator's job depends on knowing when to deploy what, and nodes can be very, very expensive, depending on where they are run. I wouldn't want someone deploying nodes over and over when what they needed to do was deploy a bunch of pods.

What is kubectl? (And how do you pronounce it?)

This question is one of my higher priority questions, but it may not be relevant for you and your team. In my organization, we don't use a graphical interface to manage our Kubernetes environments, which means command-line actions are all we do.

So what is kubectl [5]? It is the command-line interface to Kubernetes. You can get and set anything from there, from gathering logs and events to editing deployments and secrets. It's always helpful to pop in a random question about how to use this tool to test the interviewee's familiarity with kubectl.

How do you pronounce it? Well, that's up to you (there's a big disagreement on the matter), but I will gladly point you to this great video presentation by my friend Waldo [6].

Watch the video

What is a namespace?

I haven't received an answer to this question on multiple interviews. I am not sure that namespaces are used as often in other environments as they are in the organization I work in. I'll give a short answer here: a namespace is a virtual cluster in a pod. This abstraction is what enables you to keep several virtual clusters in various environments for isolation purposes.

What is a container?

It always helps to know what is being deployed in your pod, because what's a deployment without knowing what you're deploying in it? A container is a standard unit of software that packages up code and all its dependencies. Two optional secondary answers I have received and am OK with include: a) a slimmed-down image of an OS and b) an application running in a limited OS environment. Bonus points if you can name orchestration software that uses containers other than Docker [7], like your favorite public cloud's container service.

Other questions

If you're wondering why I didn't add more to this list of questions, I have an easy answer for you: these are the minimum set of things *you* should know when you are asking candidates interview questions. The next set of questions should come from a large list of questions based on your specific team, environment, and organization. As you think through these, try to find interesting questions about how technology interacts with each other to see how people think through infrastructure challenges. Think about recent challenges your team had (outages), ask to walk through deployments step-by-step, or about strategies to improve something your team actively wants to improve (like a reduction to deployment time). The less abstract the questions, the more your asking about skills that will actually matter after the interview.

No two environments will be the same, and this also applies when you are interviewing people. I mix up questions in every interview. I also have a small environment I can use to test interviewees. I always find that answering the questions is the easiest part, and doing the work is the real test you need to give.

My last bit of advice for anyone giving interviews: If you meet someone who has potential but none of the experience, give them a chance to prove themselves. I wouldn't have the knowledge and experience I have today if someone hadn't seen the potential of what I could do and given me an opportunity.

Links

[1] https://kubernetes.io/
[2] https://twitter.com/ChrisShort
[3] https://kubernetes.io/docs/concepts/architecture/nodes/
[4] https://kubernetes.io/docs/concepts/workloads/pods/pod/
[5] https://kubernetes.io/docs/reference/kubectl/kubectl/
[6] https://opensource.com/article/18/12/kubectl-definitive-pronunciation-guide
[7] https://opensource.com/resources/what-docker

Demystifying namespaces and containers in Linux

BY SETH KENLON

Peek behind the curtains to understand the backend of Linux container technology.

CONTAINERS have taken the world by storm. Whether you think of Kubernetes, Docker, CoreOS, Silverblue, or Flatpak when you hear the term, it's clear that modern applications are running in containers for convenience, security, and scalability.

Containers can be confusing to understand, though. What does it mean to run in a container? How can processes in a container interact with the rest of the computer they're running on? Open source dislikes mystery, so this article explains the backend of container technology, just as my article on Flatpak [1] explained a common frontend.

Namespaces

Namespaces are common in the programming world. If you dwell in the highly technical places of the computer world, then you have probably seen code like this:

```
using namespace std;
```

Or you may have seen this in XML:

```
<book xmlns="http://docbook.org/ns/docbook" xml:lang="en">
```

These kinds of phrases provide context for commands used later in a source code file. The only reason C++ knows, for instance, what programmers mean when they type cout is because C++ knows the cout namespace is a meaningful word.

If that's too technical for you to picture, you may be surprised to learn that we all use namespaces every day in real life, too. We don't call them namespaces, but we use the concept all the time. For instance, the phrase "I'm a fan of the Enterprise" has one meaning in an IT company that serves large businesses (which are commonly called "enterprises"), but it may have a different meaning at a science fiction convention. The question "what engine is it running?" has one meaning in a garage and a different meaning in web development. We don't always declare a namespace in casual conversation because we're human, and our brains can adapt quickly to determine context, but for computers, the namespace must be declared explicitly.

For containers, a namespace is what defines the boundaries of a process' "awareness" of what else is running around it.

lsns

You may not realize it, but your Linux machine quietly maintains different namespaces specific to given processes. By using a recent version of the **util-linux** package, you can list existing namespaces on your machine:

```
$ lsns
        NS TYPE   NPROCS   PID USER    COMMAND
4026531835 cgroup     85  1571 seth /usr/lib/systemd/systemd --user
4026531836 pid        85  1571 seth /usr/lib/systemd/systemd --user
4026531837 user       80  1571 seth /usr/lib/systemd/systemd --user
4026532601 user        1  6266 seth /usr/lib64/firefox/firefox [...]
4026532928 net         1  7164 seth /usr/lib64/firefox/firefox [...]
[...]
```

If your version of **util-linux** doesn't provide the **lsns** command, you can see namespace entries in **/proc**:

```
$ ls /proc/*/ns
1571
6266
7164
[...]
$ ls /proc/6266/ns
ipc net pid user uts [...]
```

Each process running on your Linux machine is enumerated with a process ID (PID). Each PID is assigned a namespace. PIDs in the same namespace can have access to one another because they are programmed to operate within a given

namespace. PIDs in different namespaces are unable to interact with one another by default because they are running in a different context, or *namespace*. This is why a process running in a "container" under one namespace cannot access information outside its container or information running inside a different container.

Creating a new namespace

A usual feature of software dealing with containers is automatic namespace management. A human administrator starting up a new containerized application or environment doesn't have to use **lsns** to check which namespaces exist and then create a new one manually; the software using PID namespaces does that automatically with the help of the Linux kernel. However, you can mimic the process manually to gain a better understanding of what's happening behind the scenes.

First, you need to identify a process that is not running on your computer. For this example, I'll use the Z shell (Zsh) [2] because I'm running the Bash shell on my machine. If you're running Zsh on your computer, then use **Bash** or **tcsh** or some other shell that you're not currently running. The goal is to find something that you can prove is not running. You can prove something is not running with the **pidof** command, which queries your system to discover the PID of any application you name:

```
$ pidof zsh
$ sudo pidof zsh
```

As long as no PID is returned, the application you have queried is not running.

Unshare

The **unshare** command runs a program in a namespace *unshared* from its parent process. There are many kinds of namespaces available, so read the **unshare** man page for all options available.

To create a new namespace for your test command:

```
$ sudo unshare --fork --pid --mount-proc zsh
%
```

Because Zsh is an interactive shell, it conveniently brings you into its namespace upon launch. Not all processes do that, because some processes run in the background, leaving you at a prompt in its native namespace. As long as you remain in the Zsh session, you can see that you have left the usual namespace by looking at the PID of your new forked process:

```
% pidof zsh
pid 1
```

If you know anything about Linux process IDs, then you know that PID 1 is always reserved, mostly by nature of the boot process, for the initialization application (systemd on most distributions outside of Slackware, Devuan, and maybe some customized installations of Arch). It's next to impossible for Zsh, or any application that isn't a boot initialization application, to be PID 1 (because without an init system, a computer wouldn't know how to boot up). Yet, as far as your shell knows in this demonstration, Zsh occupies the PID 1 slot.

Despite what your shell is now telling you, PID 1 on your system has not been replaced. Open a second terminal or terminal tab on your computer and look at PID 1:

```
$ ps 1
init
```

And then find the PID of Zsh:

```
$ pidof zsh
7723
```

As you can see, your "host" system sees the big picture and understands that Zsh is actually running as some high-numbered PID (it probably won't be 7723 on your computer, except by coincidence). Zsh sees itself as PID 1 only because its scope is confined to (or *contained* within) its namespace. Once you have forked a process into its own namespace, its children processes are numbered starting from 1, but only within that namespace.

Namespaces, along with other technologies like **cgroups** and more, form the foundation of containerization. Understanding that namespaces exist within the context of the wider namespace of a host environment (in this demonstration, that's your computer, but in the real world the host is typically a server or a hybrid cloud) can help you understand how and why containerized applications act the way they do. For instance, a container running a Wordpress blog doesn't "know" it's not running in a container; it knows that it has access to a kernel and some RAM and whatever configuration files you've provided it, but it probably can't access your home directory or any directory you haven't specifically given it permission to access. Furthermore, a runaway process within that blog software can't affect any other process on your system, because as far as it knows, the PID "tree" only goes back to 1, and 1 is the container it's running in.

Containers are a powerful Linux feature, and they're getting more popular every day. Now that you understand how they work, try exploring container technology such as Kubernetes, Silverblue, or Flatpak, and see what you can do with containerized apps. Containers are Linux, so start them up, inspect them carefully, and learn as you go.

Links

[1] https://opensource.com/article/19/10/how-build-flatpak-packaging
[2] https://opensource.com/article/19/9/getting-started-zsh