O'REILLY®

Compliments of NGINX

PREVIEW EDITION

Kubernetes Cookbook

BUILDING CLOUD NATIVE APPLICATIONS

Sébastien Goasguen & Michael Hausenblas

Kubernetes Cookbook

Building Distributed Applications

This Preview Edition of *Kubernetes Cookbook*, Chapters 1, 2, 3, 7, 8 and 9, is a work in progress. The final book is currently scheduled for release in January 2018 and will be available at oreilly.com and other retailers once it is published.

Sebastien Goasguen and Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kubernetes Cookbook
by Sébastien Goasguen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com/safari*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Brian Anderson and Virginia Wilson
Production Editor:
Copyeditor:
Proofreader:
Indexer:
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

January 2018: First Edition

Revision History for the First Edition
2017-09-xx: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491979686* for release details.

978-1-491-97968-6

[LSI]

Table of Contents

Foreword

Kubernetes has recently emerged as the favorite amongst container orchestration tools, and some are already declaring it the winner. Kubernetes was originally created by Google and was subsequently donated to the Cloud Native Computing Foundation (CNCF). Kubernetes automatically schedules containers to run evenly among a cluster of servers, abstracting this complex task from developers and operators.

In this excerpt from the handy *Kubernetes Cookbook*, you'll find select recipes to help you get started with Kubernetes. It includes chapters on setting up the Kubernetes command line interface (CLI), as well as setting up a minikube environment on your laptop.

To further help you utilize Kubernetes within your organization, we've created the NGINX Kubernetes Ingress Controller. You can use the same NGINX you know and trust to provide production-grade application delivery to your Kubernetes applications, all managed through the native Kubernetes Ingress framework. This includes request routing, SSL termination, HTTP/2, and other great NGINX features. NGINX Plus customers get additional features such as session persistence and JWT authentication.

To learn more about the Kubernetes Ingress controller, please visit *https://www.nginx.com/products/nginx/kubernetes-ingress-controller*.

We hope that you will find these chapters and the NGINX Kubernetes Ingress controller valuable to you as you explore Kubernetes.

—Faisal Memon
Product Marketer, NGINX Inc.

CHAPTER 1

Getting Started With Kubernetes

The first chapter has recipes around installing Kubernetes, from components such as the command line interface (CLI) that allows you to interact with a cluster to an all-in-one solution you can run on your laptop (Minikube) to setting up a remote cluster running in a public cloud.

1.1 Installing the Kubernetes CLI `kubectl`

Problem

You want to install the Kubernetes command line interface (CLI) to be able to interact with your Kubernetes cluster.

Solution

Install `kubectl` using one of the following ways:

- Download the source tarballs.
- Use a package manager.
- Build from source (see ???).

The documentation highlights a few mechanisms to get `kubectl`. The easiest is to get it from the official releases. For example on a Linux system to get the latest stable version, do:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/ \
  $(curl -s https://storage.googleapis.com/kubernetes-release/ \
  release/stable.txt) \
  /bin/linux/amd64/kubectl
```

```
$ chmod +x ./kubectl

$ sudo mv ./kubectl /usr/local/bin/kubectl
```

Users of macOS can get `kubectl` simply via Homebrew:

```
$ brew install kubectl
```

Google Container Engine users (see Recipe 1.2) will get `kubectl` as part of the `gcloud` command installation. For example, on my local machine:

```
$ which kubectl
/Users/sebgoa/google-cloud-sdk/bin/kubectl
```

Also note that latest versions of Minikube (see Recipe 1.5) packages `kubectl` and will install it in your `$PATH` if it is not found.

Before you move one from this recipe, make sure you have a working `kubectl` by listing its version. It will also try to get the version of a default Kubernetes cluster:

```
$ kubectl version
Client Version: version.Info{Major:"1", \
                             Minor:"6", \
                             GitVersion:"v1.6.0", \
                             GitCommit:"fff5156...", \
                             GitTreeState:"clean", \
                             BuildDate:"2017-03-28T16:36:33Z", \
                             GoVersion:"go1.7.5", \
                             Compiler:"gc", \
                             Platform:"darwin/amd64"}
...
```

See Also

- Documentation on installing `kubectl`

1.2 Creating A Kubernetes Cluster On Google Container Engine (GKE)

Problem

You want to create a Kubernetes cluster on Google Container Engine (GKE).

Solution

Using the `gcloud` command line interface, create a Kubernetes cluster with the `container cluster create` command like so:

```
$ gcloud container clusters create barfoo
```

By default this will create a Kubernetes cluster with three worker nodes. The master node is being managed by the GKE service and can not be accessed.

Discussion

To use GKE you will first need a few things:

- An account on the Google Cloud Platform with billing enabled.
- Create a project and enable the GKE service in it.
- The `gcloud` CLI installed on your local machine.

To speed up the setup of `gcloud`, you can make use of the Google Cloud Shell, a pure on-line, browser-based solution.

Once your cluster is created, you can list it as shown in the following:

```
$ gcloud container clusters list
NAME    ZONE            MASTER_VERSION  MASTER_IP     ...  STATUS
barfoo  europe-west1-b  1.5.7           35.187.80.94  ...  RUNNING
```

The `gcloud` CLI allows you to resize your cluster, update and upgrade it:

```
...
COMMANDS
...
     resize
        Resizes an existing cluster for running contain-
ers.
     update
        Update cluster settings for an existing con-
tainer cluster.
     upgrade
        Upgrade the Kubernetes version of an existing
container cluster.
```

Once you are done using your cluster, do not forget to delete it to avoid being charged:

```
$ gcloud container clusters delete barfoo
```

See Also

- GKE quick start guide
- Google Cloud Shell quick start guide

1.3 Creating A Kubernetes Cluster On Azure Container Service (ACS)

Problem

You want to create a Kubernetes cluster on Azure Container Service (ACS).

Solution

To carry out the following steps you will need to sign up for a (free) Azure account as well as install the Azure CLI `az` in version 2.0.

First, we make sure that we have the correct `az` CLI version installed and log in:

```
$ az --version | grep ^azure-cli
azure-cli (2.0.13)

$ az login
To sign in, use a web browser to open the page https://aka.ms/devicelogin and
enter the code XXXXXXXXX to authenticate.
[
  {
    "cloudName": "AzureCloud",
    "id": "****************************",
    "isDefault": true,
    "name": "Free Trial",
    "state": "Enabled",
    "tenantId": "****************************",
    "user": {
      "name": "******@hotmail.com",
      "type": "user"
    }
  }
]
```

As a preparation, we create an Azure resource group (the equivalent of a project in Google Cloud). This resource group called `k8s` holds all our resources such as VMs or networking components and makes it easy to clean up and tear down, later on:

```
$ az group create --name k8s --location northeurope
{
  "id": "/subscriptions/**********************/resourceGroups/k8s",
  "location": "northeurope",
  "managedBy": null,
  "name": "k8s",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

Tip Selecting Azure Region

If you're unsure what region to use for the `--location` argument, execute `az account list-locations` and pick one near to you.

Now that we have the resource group `k8s` set up, we can create the cluster with one worker node (agent in Azure terminology) like so:

```
$ az acs create --orchestrator-type kubernetes \
                --resource-group k8s \
                --name k8scb \
                --agent-count 1 \
                --generate-ssh-keys
waiting for AAD role to propagate.done
{
...
"provisioningState": "Succeeded",
"template": null,
"templateLink": null,
"timestamp": "2017-08-13T19:02:58.149409+00:00"
},
"resourceGroup": "k8s"
}
```

Note that above `az acs create` command might take up to 10 minutes to complete.

Warning Cluster Size With Free Account

With the Azure Free Trial you don't have enough quota to create a default (three agent) Kubernetes cluster, so you will see something like the following if you try it:

```
Operation results in exceeding quota limits of Core.
Maximum allowed: 4, Current in use: 0, Additional
requested: 8.
```

To work around this, either create a smaller cluster, for example, with `--agent-count 1`, or use a paid subscription instead.

As a result, in the Azure portal you should now see something like depicted in Figure 1-1. Start by finding the `k8s` resource group and then navigate your way through the `Deployments` tab.

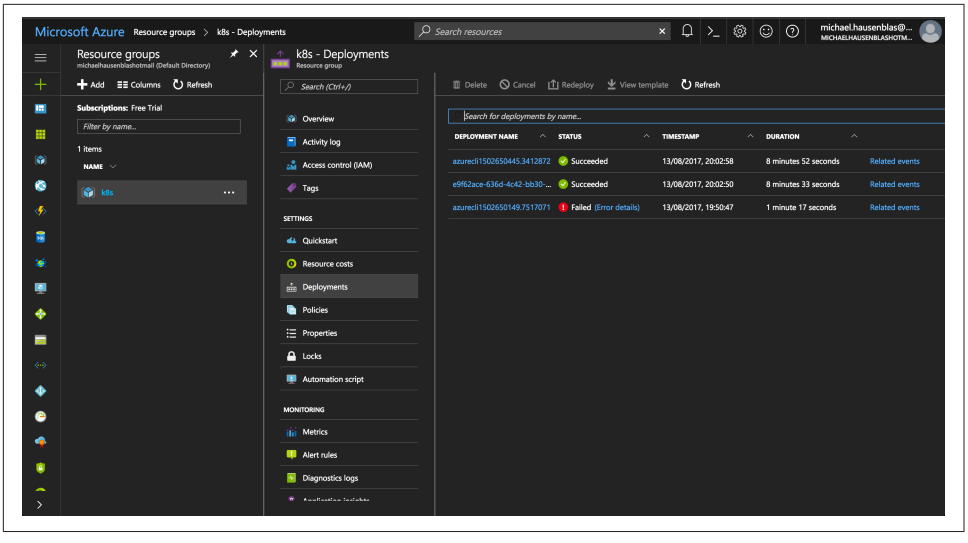


Figure 1-1. Screen Shot Of The Azure Portal, Showing ACS Deployments In The `k8s` Resource Group

We're now in a position to connect to the cluster:

```
$ az acs kubernetes get-credentials --resource-group=k8s --name=k8scb
```

We can now poke around in the environment and verify the setup:

```
$ kubectl cluster-info
Kubernetes master is running at https://k8scb-
k8s-143f1emgmt.northeurope.cloudapp.azure.com
Heapster is running at https://k8scb-
k8s-143f1emgmt.northeurope.cloudapp.azure.com/api/v1/namespaces/kube-system/
services/heapster/proxy
KubeDNS is running at https://k8scb-
k8s-143f1emgmt.northeurope.cloudapp.azure.com/api/v1/namespaces/kube-system/
services/kube-dns/proxy
kubernetes-dashboard is running at https://k8scb-
k8s-143f1emgmt.northeurope.cloudapp.azure.com/api/v1/namespaces/kube-system/
services/kubernetes-dashboard/proxy
tiller-deploy is running at https://k8scb-
k8s-143f1emgmt.northeurope.cloudapp.azure.com/api/v1/namespaces/kube-system/
services/tiller-deploy/proxy

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

$ kubectl get nodes
NAME                    STATUS                     AGE       VERSION
k8s-agent-1a7972f2-0    Ready                      7m        v1.6.6
k8s-master-1a7972f2-0   Ready,SchedulingDisabled   7m        v1.6.6
```

And indeed, as you can see above, there's one agent (worker) node and one master node.

Now is a good time to give it a test drive, for example:

```
$ kubectl run webserver --image=nginx:1.7
deployment "webserver" created

$ kubectl expose deploy/webserver --port=80 --target-port=80
service "webserver" exposed

$ kubectl get deploy,rs,po,svc
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/webserver   1         1         1            1           2m

NAME                      DESIRED   CURRENT   READY     AGE
rs/webserver-660925959    1         1         1         2m

NAME                            READY     STATUS    RESTARTS   AGE
po/webserver-660925959-sp2zk    1/1       Running   0          2m

NAME             CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
svc/kubernetes   10.0.0.1       <none>        443/TCP   12m
svc/webserver    10.0.128.203   <none>        80/TCP    18s
```

OK, that looks great! We've set up a Kubernetes cluster in the Azure Container Service and launched an app in it.

Don't forget to shut down the cluster and remove all the resources when you're done discovering ACS by deleting the resource group `k8s`:

```
$ az group delete --name k8s --yes --no-wait
```

Although above `az group delete` command returns immediately it can take up to 10 minutes until all the resources, such as VMs, virtual networks, or disks are removed and the resource group is actually destroyed. You might want to check in the Azure Portal to make sure everything went according to the plan.

Tip Azure Cloud Shell

If you don't want to or can not install the Azure CLI, you can use the Azure Cloud Shell from within your browser instead to carry out above steps to install the Kubernetes cluster.

See Also

- Deploy Kubernetes cluster for Linux containers in the Azure docs

1.4 Creating A Kubernetes Cluster With `kubicorn`

Problem

You want to create a Kubernetes cluster on AWS.

Solution

Use kubicorn to create and manage Kubernetes clusters on AWS. Since `kubicorn` currently doesn't provide for binary releases, you need to have Go installed for the following to work.

Let's install `kubicorn` first. We are using a CentOS environment here and make sure that at least Go version 1.8 is available:

```
$ go version
go version go1.8 linux/amd64

$ yum group install "Development Tools" \
  yum install ncurses-devel

$ go get github.com/kris-nova/kubicorn
...
Create, Manage, Image, and Scale Kubernetes infrastructure in the cloud.

Usage:
  kubicorn [flags]
  kubicorn [command]

Available Commands:
  adopt       Adopt a Kubernetes cluster into a Kubicorn state store
  apply       Apply a cluster resource to a cloud
  completion  Generate completion code for bash and zsh shells.
  create      Create a Kubicorn API model from a profile
  delete      Delete a Kubernetes cluster
  getconfig   Manage Kubernetes configuration
  help        Help about any command
  image       Take an image of a Kubernetes cluster
  list        List available states
  version     Verify Kubicorn version

Flags:
  -C, --color         Toggle colorized logs (default true)
  -f, --fab           Toggle colorized logs
  -h, --help          help for kubicorn
  -v, --verbose int   Log level (default 3)

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

Now that we've got the `kubicorn` command installed, we create the cluster resources by selecting a so called profile and verify if the resources are properly defined:

```
$ kubicorn create --name k8scb --profile aws
2017-08-14T05:18:24Z [ ]  Selected [fs] state store
2017-08-14T05:18:24Z [ ]  The state [./_state/k8scb/cluster.yaml] has been cre-
ated. You can edit the file, then run `kubicorn apply -n k8scb`

$ cat _state/k8scb/cluster.yaml
SSH:
  Identifier: ""
  metadata:
    creationTimestamp: null
  publicKeyPath: ~/.ssh/id_rsa.pub
  user: ubuntu
cloud: amazon
kubernetesAPI:
  metadata:
    creationTimestamp: null
  port: "443"
location: us-west-2
...
```

The default resource profile we're using assumes you've got the key pair in `~/.ssh` named `id_rsa` (private key) and `id_rsa.pub` (public key). If this it not the case, you might want to change this. Also, note that the default region used is Oregon, `us-west-2`.

To continue, you need to have an IAM user with the following permissions available: `AmazonEC2FullAccess`, `AutoScalingFullAccess`, and `AmazonVPCFullAccess`. If you don't have such an IAM user, now is a good time to create one.

One last thing we need to prepare for `kubicorn` to work. Set the credentials of the IAM user you're using (see previous step) as environment variables as follows:

```
$ export AWS_ACCESS_KEY_ID=**************************
$ export AWS_SECRET_ACCESS_KEY=****************************************
```

Now we're in a position to create the cluster, based on the resource definitions above as well as the AWS access we provided:

```
$ kubicorn apply --name k8scb
2017-08-14T05:45:04Z [ ]  Selected [fs] state store
2017-08-14T05:45:04Z [ ]  Loaded cluster: k8scb
2017-08-14T05:45:04Z [ ]  Init Cluster
2017-08-14T05:45:04Z [ ]  Query existing resources
2017-08-14T05:45:04Z [ ]  Resolving expected resources
2017-08-14T05:45:04Z [ ]  Reconciling
2017-08-14T05:45:07Z [ ]  Created KeyPair [k8scb]
2017-08-14T05:45:08Z [ ]  Created VPC [vpc-7116a317]
```

```
2017-08-14T05:45:09Z [ ]  Created Internet Gateway [igw-e88c148f]
2017-08-14T05:45:09Z [ ]  Attaching Internet Gateway [igw-e88c148f] to VPC
[vpc-7116a317]
2017-08-14T05:45:10Z [ ]  Created Security Group [sg-11dba36b]
2017-08-14T05:45:11Z [ ]  Created Subnet [subnet-50c0d919]
2017-08-14T05:45:11Z [ ]  Created Route Table [rtb-8fd9dae9]
2017-08-14T05:45:11Z [ ]  Mapping route table [rtb-8fd9dae9] to internet gate-
way [igw-e88c148f]
2017-08-14T05:45:12Z [ ]  Associated route table [rtb-8fd9dae9] to subnet
[subnet-50c0d919]
2017-08-14T05:45:15Z [ ]  Created Launch Configuration [k8scb.master]
2017-08-14T05:45:16Z [ ]  Created Asg [k8scb.master]
2017-08-14T05:45:16Z [ ]  Created Security Group [sg-e8dca492]
2017-08-14T05:45:17Z [ ]  Created Subnet [subnet-cccfd685]
2017-08-14T05:45:17Z [ ]  Created Route Table [rtb-76dcdf10]
2017-08-14T05:45:18Z [ ]  Mapping route table [rtb-76dcdf10] to internet gate-
way [igw-e88c148f]
2017-08-14T05:45:19Z [ ]  Associated route table [rtb-76dcdf10] to subnet
[subnet-cccfd685]
2017-08-14T05:45:54Z [ ]  Found public IP for master: [34.213.102.27]
2017-08-14T05:45:58Z [ ]  Created Launch Configuration [k8scb.node]
2017-08-14T05:45:58Z [ ]  Created Asg [k8scb.node]
2017-08-14T05:45:59Z [ ]  Updating state store for cluster [k8scb]
2017-08-14T05:47:13Z [ ]  Wrote kubeconfig to [/root/.kube/config]
2017-08-14T05:47:14Z [ ]  The [k8scb] cluster has applied successfully!
2017-08-14T05:47:14Z [ ]  You can now `kubectl get nodes`
2017-08-14T05:47:14Z [ ]  You can SSH into your cluster ssh -i ~/.ssh/id_rsa
ubuntu@34.213.102.27
```

Although you don't see the beautiful coloring here, the last four lines of output are green and tell you that everything has been successfully set up. You can also verify this by visiting the Amazon EC2 console in a browser as shown in Figure 1-2.

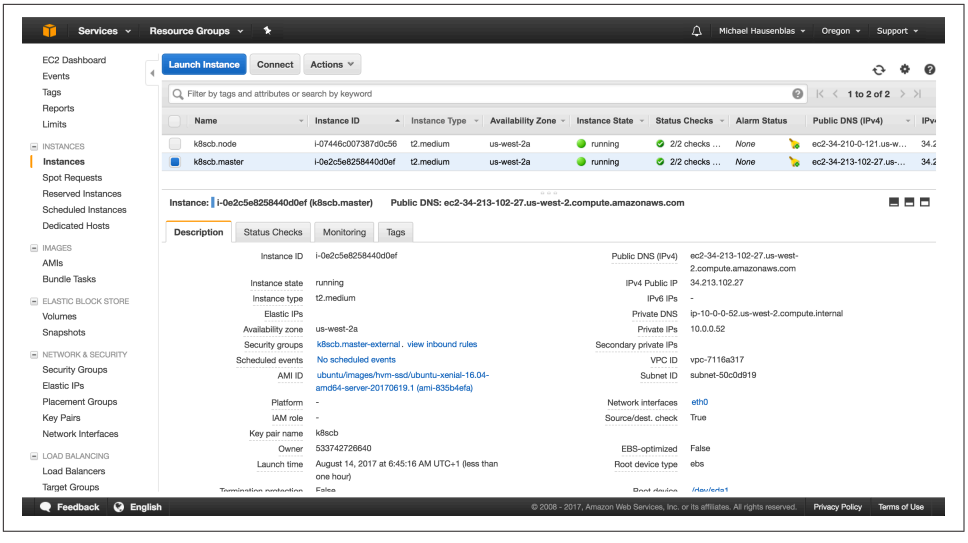


Figure 1-2. Screen Shot Of Amazon EC2 Console, Showing Two Nodes Created By `kubicorn`

Now let's do as instructed in the last output line of the `kubicorn apply` command and ssh into the cluster:

```
$ ssh -i ~/.ssh/id_rsa ubuntu@34.213.102.27
The authenticity of host '34.213.102.27 (34.213.102.27)' can't be established.
ECDSA key fingerprint is ed:89:6b:86:d9:f0:2e:3e:50:2a:d4:09:62:f6:70:bc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '34.213.102.27' (ECDSA) to the list of known hosts.
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-1020-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  Get cloud support with Ubuntu Advantage Cloud Guest:
    http://www.ubuntu.com/business/services/cloud

75 packages can be updated.
32 updates are security updates.


To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-10-0-0-52:~$ kubectl get all -n kube-system
NAME                                       READY     STATUS
RESTARTS   AGE
po/calico-etcd-qr3f1                       1/1       Running
0          4m
```

```
po/calico-node-9t472                              2/2       Running
4          4m
po/calico-node-qlpp6                              2/2       Running
1          4m
po/calico-policy-controller-1727037546-f152z      1/1       Running
0          4m
po/etcd-ip-10-0-0-52                              1/1       Running
0          3m
po/kube-apiserver-ip-10-0-0-52                    1/1       Running
0          3m
po/kube-controller-manager-ip-10-0-0-52           1/1       Running
0          4m
po/kube-dns-2425271678-zcfdd                      0/3       ContainerCreating
0          4m
po/kube-proxy-3s2c0                               1/1       Running
0          4m
po/kube-proxy-t10ck                               1/1       Running
0          4m
po/kube-scheduler-ip-10-0-0-52                    1/1       Running
0          3m

NAME              CLUSTER-IP      EXTERNAL-IP   PORT(S)         AGE
svc/calico-etcd   10.96.232.136   <none>        6666/TCP        4m
svc/kube-dns      10.96.0.10      <none>        53/UDP,53/TCP   4m

NAME                              DESIRED   CURRENT   UP-TO-DATE   AVAILABLE
AGE
deploy/calico-policy-controller   1         1         1            1
4m
deploy/kube-dns                   1         1         1            0
4m

NAME                                        DESIRED   CURRENT   READY     AGE
rs/calico-policy-controller-1727037546      1         1         1         4m
rs/kube-dns-2425271678                      1         1         0         4m
```

When you're done, tear down the Kubernetes cluster like so (note: takes a couple of minutes):

```
$ kubicorn delete --name k8scb
2017-08-14T05:53:38Z [ ]  Selected [fs] state store
Destroying resources for cluster [k8scb]:
2017-08-14T05:53:41Z [ ]  Deleted ASG [k8scb.node]
...
2017-08-14T05:55:42Z [ ]  Deleted VPC [vpc-7116a317]
```

Discussion

While `kubicorn` is a rather young project it is fully functional and you can also create clusters on Azure and Digital Ocean with it.

It does require you to have Go installed as it doesn't ship binaries (yet) but it's very flexible in terms of configuration and also rather intuitive to handle, especially if you have an admin background.

See Also

- Setting up Kubernetes in AWS in the `kubicorn` docs Setting up Kubernetes in Azure
- Setting up a cluster with `kubicorn` on Digital Ocean video walkthrough

1.5 Installing Minikube To Run A Local Kubernetes Instance

Problem

You want to use Kubernetes for testing or development or for training purposes on your local machine.

Solution

Use Minikube. Minikube is a tool that lets you use Kubernetes on your local machine without any installation except installing `minikube` itself. It takes advantage of your local hypervisor (e.g VirtualBox, KVM) and launches a virtual machine that runs Kubernetes in a single node.

To install the `minikube` CLI locally, you can get the latest release or build from source. To get the `v0.18.0`release and install `minikube` on a Linux based machine do:

```
$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/v0.18.0 \
       /minikube-linux-amd64

$ chmod +x minikube

$ sudo mv minikube /usr/local/bin/
```

This will put the `minikube` binary in your path and make it accessible from everywhere.

Discussion

Once `minikube` is installed, you can verify the version that it is running with the following command:

```
$ minikube version
minikube version: v0.16.0
```

You can start it with:

```
$ minikube start
```

Once the startup phase has finished, your Kubernetes client `kubectl` will have a `minikube` context and will automatically start using this context. Checking what nodes you have in your cluster will return the `minikube` hostname:

```
$ kubectl get nodes
NAME       STATUS    AGE
minikube   Ready     5d
```

See Also

- Official Minikube docs
- `minikube` source on GitHub

1.6 Using Minikube Locally for Development

Problem

You want to use `minikube` locally for testing and development of your Kubernetes application. You have installed and started `minikube` (see Recipe 1.5) and want to know a few extras commands to simplify your development experience.

Solution

The `minikube` CLI offers a few commands that make your life easier. The CLI has built-in help that you can use discover the subcommands on your own, see below for a snippet of these subcommands:

```
$ minikube
...
Available Commands:
  addons           Modify minikube's kubernetes addons
...
  start            Starts a local kubernetes cluster.
  status           Gets the status of a local kubernetes cluster.
  stop             Stops a running local kubernetes cluster.
  version          Print the version of minikube.
```

Aside from `start`, `stop` and `delete`, you should become familiar with `ip`, `ssh`, `dashboard`, `docker-env`.

Minikube runs a Docker engine to be able to start containers. In order to access this Docker engine from your local machine using your local Docker client you can setup the correct Docker environment with `minikube docker-env`.

Discussion

The `minikube start` command starts the Virtual Machine (VM) that will run Kubernetes locally. By default it will allocate 2GB of RAM, so when you are done do not forget to stop it with `minikube stop`. Also, you can give the VM more memory and CPUs as well as pick a certain Kubernetes version to run, for example:

```
$ minikube start --cpus=4 --memory=4000 --kubernetes-version=v1.7.2
```

For debugging the Docker daemon that is used inside Minikube you might find `minikube ssh` handy, it will log you inside the virtual machine. To get the IP address of the Minikube VM, use `minikube ip`. Finally, to launch the Kubernetes dashboard in your default browser, use `minikube dashboard`.

If for any reason your Minikube gets unstable, or you want to start afresh, you can `minikube stop` and `minikube delete`. Then a `minikube start` will give you a fresh installation.

1.7 Starting your First Application on Minikube

Problem

You started Minikube (see Recipe 1.5) and now you want to launch your first application on Kubernetes.

Solution

We are going to start the Ghost micro-blogging platform on Minikube, using two `kubectl` commands.

```
$ kubectl run ghost --image=ghost:0.9
$ kubectl expose deployments ghost --port=2368 --type=NodePort
```

Monitor the pod manually to see when it starts running and then use `minikube service` command to open your browser automatically and access Ghost:

```
$ kubectl get pods
$ minikube service ghost
```

Discussion

The `kubectl run` command is called a generator, it is a convenience command to create a Deployment object (see Recipe 3.4). The `kubectl expose` command is also a generator, a convenience command to create a service object (see ???) that routes network traffic to the containers started by your deployment.

1.8 Accessing the Dashboard in Minikube

Problem

You are using Minikube and want to access the Kubernetes dashboard to start your first application from a graphical user interface.

Solution

You can open the Kubernetes dashboard from Minikube with:

```
minikube dashboard
```

By clicking on the plus icon at the top right of the UI that opened in your browser you will get to the page depicted in Figure 1-3.

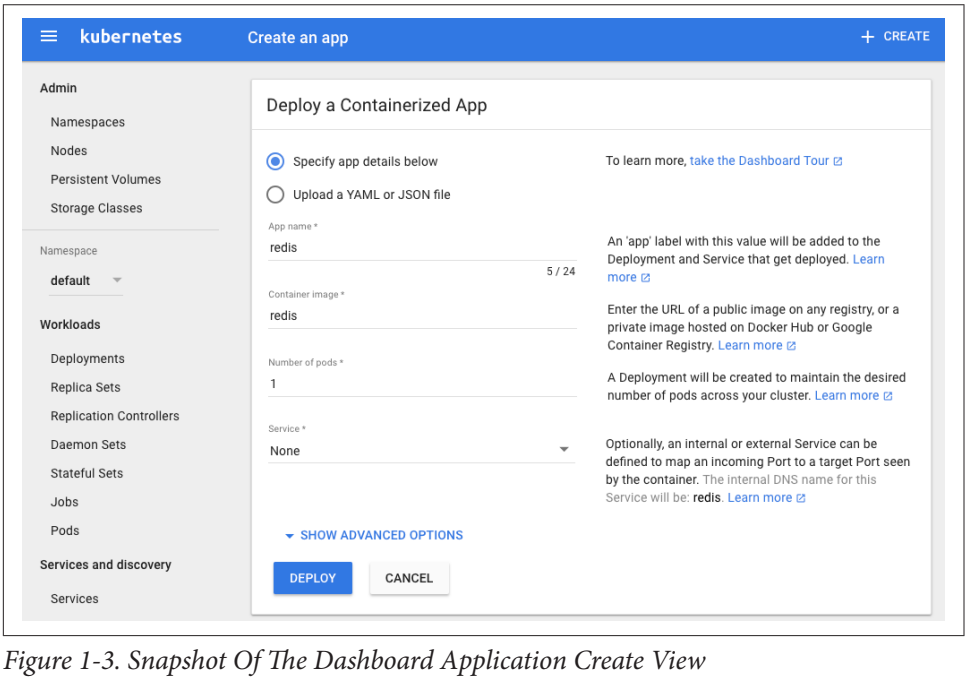


Figure 1-3. Snapshot Of The Dashboard Application Create View

Discussion

To create an application, give it a name and specify the Docker image that you want to use. Then click on the create button. You will be presented with a new view that will show deployments, replica sets and after a bit of time you will see a pod. These are some key API primitives we will deal with in greater detail in the rest of the book.

The snapshot below presents a typical dashboard view after having created a single application using the Redis container:

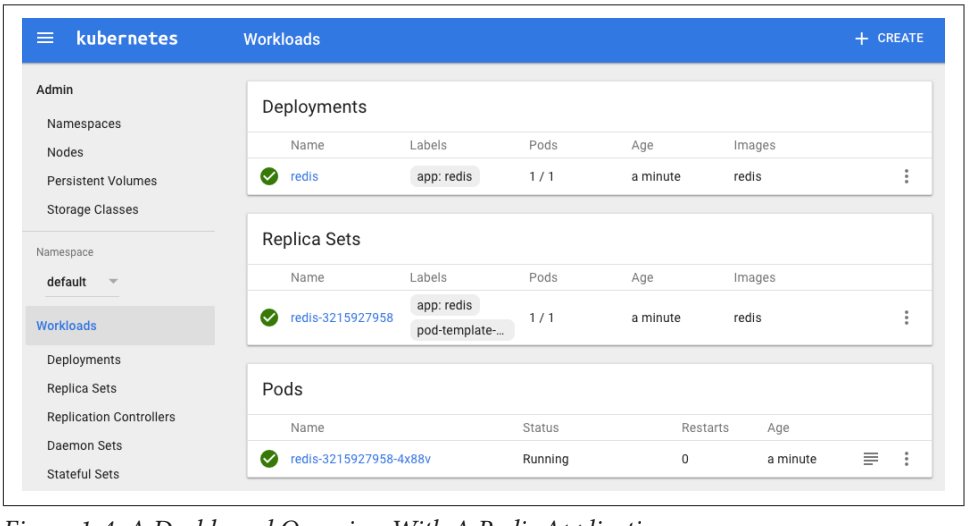


Figure 1-4. A Dashboard Overview With A Redis Application

If you get back to a terminal session and use the command line client you will see the same thing:

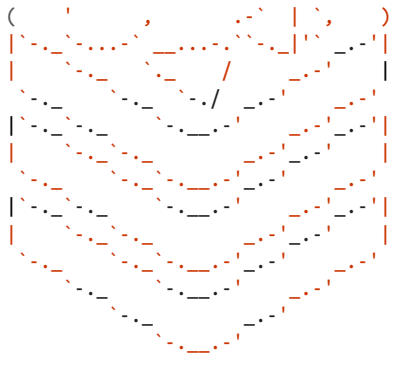
```
$ kubectl get pods,rs,deployments
NAME                        READY     STATUS    RESTARTS   AGE
po/redis-3215927958-4x88v   1/1       Running   0          24m

NAME                   DESIRED   CURRENT   READY     AGE
rs/redis-3215927958    1         1         1         24m

NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
deploy/redis   1         1         1            1           24m
```

Your Redis pod will be running the Redis server, as the logs below show:

```
$ kubectl logs redis-3215927958-4x88v
...<snip>
                _.-``__ ''-._
           _.-``    `.  `_.  ''-._           Redis 3.2.9 (00000000/0) 64 bit
       .-`` .-```.  ```\/    _.,_ ''-._
```




```
...<snip>
1:M 14 Jun 07:28:56.637 # Server started, Redis version 3.2.9
1:M 14 Jun 07:28:56.643 * The server is now ready to accept connections on port
6379
```

1.9 Installing kubeadm To Create A Kubernetes Cluster

Problem

You want to use `kubeadm` to bootstrap a Kubernetes cluster from scratch.

Solution

Download the `kubeadm` CLI tool from the Kubernetes package repository.

You will need `kubeadm` installed on all the servers that will be part of your Kubernetes cluster, not only the head node but also all the worker nodes.

For example, if you are using Ubuntu based hosts, on each host do the following to setup the Kubernetes package repository:

```
$ apt-get update && apt-get install -y apt-transport-https

$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | apt-key add -

$ cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
  deb http://apt.kubernetes.io/ kubernetes-xenial main
  EOF

$ apt-get update
```

Now you can install the Docker engine and the various Kubernetes tools:

- `kubelet` binary.
- `kubeadm` CLI.

- `kubectl` client.
- `kubernetes-cni` the Container Networking Interface (CNI) plugin.

```
$ apt-get install -y docker-engine

$ apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

Discussion

Once all the binaries and tools are installed, you are ready to start bootstrapping your Kubernetes cluster. On your head node, initialize the cluster with:

```
$ kubeadm init
```

At the end of the initialization, you will be given a command to execute on all your worker nodes (see Recipe 1.10). This command uses a token auto-generated by the initialization process.

See Also

- `kubeadm` official documentation.

1.10 Bootstrapping A Kubernetes Cluster Using `kubeadm`

Problem

You have initialized your Kubernetes head node (see Recipe 1.9) and now need to add worker nodes to your cluster.

Solution

With the Kubernetes package repository configured and `kubeadm` installed as shown in Recipe 1.9, run the `join` command using the token given to you when running the `init` step on the head node.

```
$ kubeadm join
```

Head back to your head node terminal session and you will see your worker nodes join:

```
$ kubectl get nodes
```

Discussion

The final step is to create a network that satisfies the Kubernetes networking requirements, especially the single IP per Pod. You can use any of the network add-ons.

Weave-net for example can be installed on Kubernetes clusters v1.6.0 and above with a single `kubectl` command like so:

```
$ kubectl apply -f https://git.io/weave-kube-1.6
```

This command will create Daemon sets running on all nodes in the cluster. These Daemon sets use the host network of the host and a CNI plugin to configure the local node network. Once the network is in place, your cluster nodes will enter the `READY` state.

See Also

- Creating a cluster with kubeadm

1.11 Downloading A Kubernetes Release From GitHub

Problem

You want to download an official Kubernetes release instead of compiling from source.

Solution

You can follow a manual process and go to the GitHub releases page. Choose the release, or potentially pre-release that you want to download. You then choose between the source bundle which you will need to compile or download the `kubernetes.tar.gz` file.

Alternatively you can check the latest release tags with using the GitHub API as shown below:

```
$ curl -s https://api.github.com/repos/kubernetes/kubernetes/releases | \
        jq -r .[].assets[].browser_download_url
https://github.com/kubernetes/kubernetes/releases/download/\
        v1.7.0-alpha.3/kubernetes.tar.gz
https://github.com/kubernetes/kubernetes/releases/download/\
        v1.5.7/kubernetes.tar.gz
...<snip>
```

Then download the `kubernetes.tar.gz` release package of your choice. For example to get `v1.5.7` do:

```
$ wget https://github.com/kubernetes/kubernetes/releases/download/\
       v1.5.7/kubernetes.tar.gz
```

If you want to compile Kubernetes from source see ???.

Do not forget to verify the secure hash of the `kubernetes.tar.gz` archive. The `sha256` is listed on the GitHub release page. After downloading the archive locally, generate the hash and compare it. Even though the release is not signed with GPG, verifying the hash will check the integrity of the archive.

1.12 Downloading Client And Server Binaries

Problem

You have downloaded a release archive (see Recipe 1.11) but it does not contain the actual binaries.

Solution

To keep the size of the release archive small, it does not contain the release binaries. Instead you need to download them separately. To do so run the `get-kube-binaries.sh` script as shown below:

```
$ tar -xvf kubernetes.tar.gz
$ cd kubernetes/cluster
$ ./get-kube-binaries.sh
```

Once complete, you will have the client binaries in `client/bin`:

```
$ tree ./client/bin
./client/bin
├── kubectl
└── kubefed
```

And the server binaries in `server/kubernetes/server/bin`:

```
$ tree server/kubernetes/server/bin
server/kubernetes/server/bin
├── cloud-controller-manager
├── kube-apiserver
...<snip>
```

If you want to skip downloading the release and you want to quickly download the client and/or server binaries. You can directly get them from `https://dl.k8s.io`. For example to get the `v1.6.2` binaries for linux do:

```
$ wget https://dl.k8s.io/v1.6.2/kubernetes-client-linux-
amd64.tar.gz
$ wget https://dl.k8s.io/v1.6.2/kubernetes-server-linux-
amd64.tar.gz
```

1.13 Using Hyperkube Image To Run A Kubernetes Head Node With Docker

Problem

You want to create a Kubernetes head node using a few Docker containers. Specifically, you want to run the API server, scheduler, controller and etcd key value store within containers.

Solution

You can use the `hyperkube` binary, available as a Docker image plus a `etcd` container. `hyperkube` is an all-in-one binary available as a Docker image. You can use it to start all the Kubernetes processes.

To created a Kubernetes cluster, you need a storage solution to keep the cluster state. In Kubernetes, this solution is a distributed key/value store called `etcd`, therefore first you need start an `etcd` instance. You can run it like this:

```
$ docker run -d \
         --name=k8s \
         -p 8080:8080 \
         gcr.io/google_containers/etcd:2.2.1 \
         etcd --data-dir /var/lib/data
```

Then you will start the API server using the so-called `hyperkube` image which contains the API server binary. This image is available from the Google container registry (GCR) at `gcr.io/google_containers/hyperkube:v1.6.2`. We use a few settings to serve the API insecurely on a local port. Replace `v1.6.2` with the latest version or the one you want to run.

```
$ docker run -d \
         --net=container:k8s \
         gcr.io/google_containers/hyperkube:v1.6.2 \
         /apiserver --etcd-servers=http://127.0.0.1:4001 \
        --service-cluster-ip-range=10.0.0.1/24 \
        --insecure-bind-address=0.0.0.0 \
        --insecure-port=8080 \
        --storage-backend=etcd2 \
        --admission-control=AlwaysAdmit
```

Finally, you can start the admission controller, which points to the API server:

```
$ docker run -d \
         --net=container:k8s \
         gcr.io/google_containers/hyperkube:v1.6.2 \
         /controller-manager --master=127.0.0.1:8080
```

Notice that since `etcd`, the API server and the controller manager share the same network namespace, they can reach each other on `127.0.0.1` even though they are running in different containers.

To test that you have a working setup, use the `etcdctl` command line utility in the `etcd` container and list the what is the `/registry` directory:

```
$ docker exec -ti k8s etcdctl ls /registry
/registry/ranges
/registry/namespaces
/registry/services
/registry/events
/registry/serviceaccounts
```

You can now also reach your Kubernetes API server and start exploring the API:

```
$ curl http://127.0.0.1:8080/
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/apps/v1beta1",
...<snip>
```

So far, you did not start the scheduler, nor did you setup nodes with the `kubelet` and the `kube-proxy`. This just shows you how by starting three local containers you can run the Kubernetes API server.

It is sometimes helpful to use the `hyperkube` Docker image to verify some of the configuration options of one of the Kubernetes binaries. For example to check the help of the main `/apiserver` command try:

```
$ docker run --rm -ti gcr.io/google_containers/hyper-
kube:v1.6.2 /apiserver --help
```

Discussion

While this is a very useful way to start exploring the various Kubernetes components locally, it is not recommended for production setup.

See Also

- Hyperkube Docker images

1.14 Writing A Systemd Unit File To Run Kubernetes Components

Problem

You have used Minikube (see Recipe 1.5) for learning and know how to bootstrap a Kubernetes cluster using `kubeadm` (see Recipe 1.10), but you would like to install a cluster from scratch. To do so you need to run the Kubernetes components using systemd unit files.

Solution

systemd is a system and services manager, sometimes referred to as an init system. It is now the default services manager on Ubuntu 16.04 and CentOS 7.

Checking how `kubeadm` does it is a very go way to figure out how to do it on your own. If you look closely at the `kubeadm` configuration, you will see that the `kubelet` running on every node in your cluster, including the head node, is managed by `systemd`.

Here is an example, which you can reproduce by logging into any nodes in a cluster built with `kubeadm` (see Recipe 1.10).

```
# systemctl status kubelet
  kubelet.service - kubelet: The Kubernetes Node Agent
    Loaded: loaded (/lib/systemd/system/kubelet.service; enabled; vendor preset:
enabled)
  Drop-In: /etc/systemd/system/kubelet.service.d
           └─10-kubeadm.conf
    Active: active (running) since Tue 2017-06-13 08:29:33 UTC; 2 days ago
      Docs: http://kubernetes.io/docs/
 Main PID: 4205 (kubelet)
    Tasks: 17
   Memory: 47.9M
      CPU: 2h 2min 47.666s
   CGroup: /system.slice/kubelet.service
           ├─4205 /usr/bin/kubelet --kubeconfig=/etc/kubernetes/kubelet.conf \
           |                       --require-kubeconfig=true \
           |                       --pod-manifest-path=/etc/kubernetes/mani-
fests \
           |                       --allow-privileged=true \
           |                       --network-plugin=cni \
           |                       --cni-conf
           └─4247 journalctl -k -f
```

Which gives you a link to the systemd unit file in `/lib/systemd/system/kubelet.service` and its configuration in `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`

The unit file is straightforward, it points to the `kubelet` binary installed in `/usr/bin`:

```
[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=http://kubernetes.io/docs/

[Service]
ExecStart=/usr/bin/kubelet
Restart=always
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target
```

The configuration file tells us how the `kubelet` binary is started:

```
[Service]
Environment="KUBELET_KUBECONFIG_ARGS=--kubeconfig=/etc/kubernetes/kubelet.conf
--require-kubeconfig=true"
Environment="KUBELET_SYSTEM_PODS_ARGS=--pod-manifest-path=/etc/kubernetes/mani
fests --allow-privileged=true"
Environment="KUBELET_NETWORK_ARGS=--network-plugin=cni --cni-conf-dir=/etc/cni/
net.d --cni-bin-dir=/opt/cni/bin"
Environment="KUBELET_DNS_ARGS=--cluster-dns=10.96.0.10 --cluster-
domain=cluster.local"
Environment="KUBELET_AUTHZ_ARGS=--authorization-mode=Webhook --client-ca-
file=/etc/kubernetes/pki/ca.crt"
ExecStart=
ExecStart=/usr/bin/kubelet $KUBELET_KUBECONFIG_ARGS $KUBELET_SYSTEM_PODS_ARGS
$KUBELET_NETWORK_ARGS $KUBELET_DNS_ARGS $KUBELET_AUTHZ_ARGS $KUBELET_EXTRA_ARGS
```

All the options specified, such as `--kubeconfig` defined by the environment `$KUBELET_CONFIG_ARGS` are startup options of the `kubelet` binary.

Discussion

The unit file above only deals with the `kubelet`. You can write your own unit files for all the other components of a Kubernetes cluster (i.e API server, controller-manager, scheduler, proxy). Kubernetes the hard way has examples of unit files for each component.

However, you only need to run the `kubelet`. Indeed the configuration option `--pod-manifest-path` allows you to pass a directory where the kubelet will look for manifests that it will automatically start. With `kubeadm` this directory is used to pass the manifests of the API server, scheduler, etcd and controller-manager. Hence, Kubernetes manages itself and the only thing managed by systemd is the `kubelet` process.

To illustrate this you can list the content of the `/etc/kubernetes/manifests` directory in your `kubeadm` based cluster:

```
# ls -l /etc/kubernetes/manifests
total 16
-rw------- 1 root root 1071 Jun 13 08:29 etcd.yaml
-rw------- 1 root root 2086 Jun 13 08:29 kube-apiserver.yaml
-rw------- 1 root root 1437 Jun 13 08:29 kube-controller-manager.yaml
-rw------- 1 root root  857 Jun 13 08:29 kube-scheduler.yaml
```

Looking at the details of the `etcd.yaml` manifest. You see that it is a Pod with a single container that runs `etcd`:

```
root@c7abdf:/home/ubuntu# cat /etc/kubernetes/manifests/etcd.yaml
apiVersion: v1
kind: Pod
metadata:
  creationTimestamp: null
  labels:
    component: etcd
    tier: control-plane
  name: etcd
  namespace: kube-system
spec:
  containers:
  - command:
    - etcd
    - --listen-client-urls=http://127.0.0.1:2379
    - --advertise-client-urls=http://127.0.0.1:2379
    - --data-dir=/var/lib/etcd
    image: gcr.io/google_containers/etcd-amd64:3.0.17
...<snip>
```

See Also

- `kubelet` configuration options.

1.15 Using Kubernetes Without Installation

Problem

You want to try out Kubernetes without installing it.

Solution

To use Kubernetes without installing it, use the Kubernetes playground on Katacoda.

Once you're signed in with GitHub or one of the social media authentication methods you will get to the page depicted in Figure 1-5.

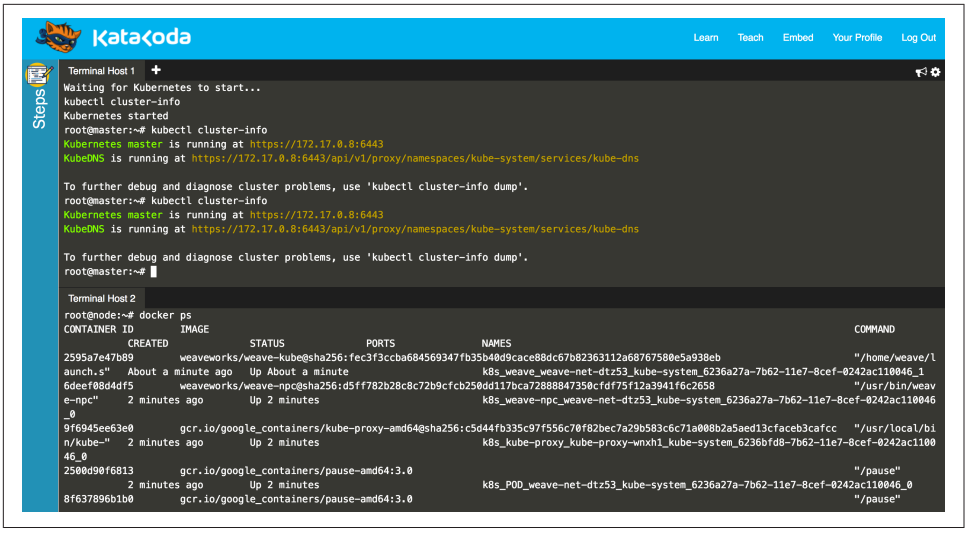


Figure 1-5. Screen Shot of The Katacoda Kubernetes Playground

Note that an environment you launch in the playground is only available for a limited time, currently one hour, but it's free of charge and all you need is a browser.

CHAPTER 2

Basics of Managing Kubernetes Objects

In this chapter we present recipes that address the basic interaction with Kubernetes objects as well as the API. Every object in Kubernetes, no matter if namespaced such as a deployment or cluster-wide ones such as nodes, has certain fields available, for example metadata, specification (spec for short) and status. The *spec* describes the desired state for an object, and the *status* captures the actual state of the object, managed by the Kubernetes API server.

2.1 Discovering API Endpoints Of The Kubernetes API Server

Problem

You want to discover the various API endpoints available on the Kubernetes API server.

Solution

If you have access to the API server via an unauthenticated private port you can directly issue HTTP request to the API server and explore the various endpoints. For example, with Minikube, you can ssh inside the Virtual Machine (`minikube ssh`) and reach the API server on port `8080`, like shown in the following:

```
$ curl localhost:8080/api/v1
...<snip>
{
      "name": "pods",
      "namespaced": true,
      "kind": "Pod",
      "verbs": [
```

```
            "create",
            "delete",
            "deletecollection",
            "get",
            "list",
            "patch",
            "proxy",
            "update",
            "watch"
          ],
          "shortNames": [
            "po"
          ]
        },
...<snip>
```

In above listing you see an example of an object of kind *Pod* as well as the allowed operations, such as `get` or `delete` on this object.

Alternatively, if you don't have direct access to the machine the Kubernetes API server is running, you can use `kubectl` to proxy the API locally. This will allow you to reach the API server locally but using an authenticated session:

```
$ kubectl proxy --port=8001 --api-prefix=/
```

And then, in another window you do:

```
$ curl localhost:8001/foobar
```

The use of the `/foobar` API paths allows you to list all the API endpoints. Note that both `--port` and `--api-prefix` are optional.

Discussion

Discovering the API endpoints as shown above, you will see different ones like:

- `/api/v1`
- `/apis/apps`
- `/apis/authentication.k8s.io`
- `/apis/authorization.k8s.io`
- `/apis/autoscaling`
- `/apis/batch`

Each of these endpoints correspond to an API group. Within a group, API objects are versioned (e.g `v1beta1`, `v1beta2`) to indicate the maturity of the objects.

Pods, services, configmaps, secrets for examples are all part of the `/api/v1` API group, while deployments are part of the `/apis/extensions/v1beta1` API group.

The group an object is part of is what is referred to as `apiVersion` in the object specification, available via the API reference.

See Also

- API groups and versioning
- API conventions

2.2 Understanding The Structure Of A Kubernetes Manifest

Problem

While Kubernetes has a few convenience generators with `kubectl run` and `kubectl create`, you need to learn how to write Kubernetes manifests, expressing Kubernetes object specifications. To do this you need to understand the general structure of manifests.

Solution

In Recipe 2.1 you explored the various API groups and were able to look for the group where a particular object is in.

All API resources are either Objects or Lists. All resources have a **kind** and an **apiVersion**.

In addition, every object kind must have **metadata**. The metadata contains the name of the object, the namespace it is in (see Recipe 2.3) and potentially some labels (see Recipe 2.6) and annotations (see ???).

A Pod, for example, will be of kind `Pod` in apiVersion `v1` and the beginning of a simple manifest written in YAML will look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
...
```

To complete a manifest, most objects will have a `spec` and once created will also return a `status`:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
...<snip>
status:
...<snip>
```

See Also

- Understanding Kubernetes objects Documentation.

2.3 Creating Namespaces To Avoid Name Collisions

Problem

You want to create two objects with the same name, but want to avoid naming collisions.

Solution

Create namespaces and place your objects in different ones.

Without specifying anything, objects will get created in the *default* namespace. So, we create a second namespace *foobar*, as shown below, and list the existing namespaces. You will see the *default* namespace and the *kube-system* namespace that were created on startup and the *foobar* namespace you just created.

```
$ kubectl create namespace foobar
namespace "foobar" created
$ kubectl get ns
NAME          STATUS    AGE
default       Active    30s
foobar        Active    1s
kube-public   Active    29s
kube-system   Active    30s
```

Alternatively, you can write a manifest to create your namespace. If you save the following manifest as *foobar.yaml* you can then create the namespace with the `kubectl create -f foobar.yaml` command:

```
apiVersion: v1
kind: Namespace
metadata:
  name: foobar
```

Discussion

You can verify that starting objects with the same name leads to collision and an error returned by the Kubernetes API server. However, when creating the objects in different namespaces, the API server does create the objects. This is due to the fact that many API objects in Kubernetes are namespaced. The namespace they belong to is defined as part of the object's metadata.

```
$ kubectl run foobar --image=ghost:0.9
deployment "foobar" created

$ kubectl run foobar --image=nginx:1.13
Error from server (AlreadyExists): deployments.extensions "foobar" already
exists

$ kubectl run foobar --image=nginx:1.13 --namespace foobar
deployment "foobar" created
```

2.4 Setting Quotas Within A Namespace

Problem

You want to limit the resources available in a namespace, for example, the overall number of pods that can run in a namespace.

Solution

Use a `ResourceQuota` object to specify the limitations on a namespace-basis:

```
$ cat resource-quota-pods.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: podquota
spec:
  hard:
    pods: "10"

$ kubectl create namespace limitedns
$ kubectl create -f resource-quota-pods.yaml --namespace=limitedns
$ kubectl describe resourcequota podquota --namespace=limitedns
Name:           podquota
Namespace:      limitedns
Resource        Used    Hard
--------        ----    ----
pods            0       10
```

Discussion

You can set a number of quotas on a per-namespace basis, including but not limited to pods, secrets, and configmaps.

See Also

- Configure Quotas for API Objects

2.5 Labeling An Object

Problem

You want to label an object in order to find it later on. The label can be used for further end-user queries (see Recipe 2.6) or in the context of system automation.

Solution

Use the `kubectl label` command, for example to label a pod named `foobar` with the key-value pair `tier=frontend`, do:

```
$ kubectl label pods foobar tier=frontend
```

Check the complete help of the command `kubectl label --help`. You can use it to remove labels, overwrite existing ones and even label all resources in a namespace.

Discussion

In Kubernetes you use labels to organize objects in a flexible, non-hierarchical manner. A label is a key-value pair without any pre-defined meaning for Kubernetes. In other words, the content of the key-value pair is not interpreted by the system. You can use labels to express membership (e.g., object X belongs to department ABC), environments (like this service runs in production), or really anything you need to organize your objects. Note that labels do have restrictions concerning their length and allowed values.

2.6 Using Labels For Queries

Problem

You want to query objects efficiently.

Solution

Use the `kubectl get --selector` command. For example, given the following pods:

```
$ kubectl get pods --show-labels
NAME                     READY     STATUS    RESTARTS   AGE       LABELS
cockroachdb-0            1/1       Running   0          17h
app=cockroachdb,controller-revision-hash=cockroachdb-165722682
cockroachdb-1            1/1       Running   0          17h
app=cockroachdb,controller-revision-hash=cockroachdb-165722682
cockroachdb-2            1/1       Running   0          17h
app=cockroachdb,controller-revision-hash=cockroachdb-165722682
jump-1247516000-sz87w    1/1       Running   9          18h       pod-template-
hash=1247516000,run=jump
nginx-4217019353-462mb   1/1       Running   0          19h       pod-template-
hash=4217019353,run=nginx
nginx-4217019353-z3g8d   1/1       Running   0          20h       pod-template-
hash=4217019353,run=nginx
prom-2436944326-pr60g    1/1       Running   0          22h       app=prom,pod-
template-hash=2436944326
```

You want to select the pods that belong to the CockroachDB app (`app=cockroachdb`):

```
$ kubectl get pods --selector app=cockroachdb
NAME            READY     STATUS    RESTARTS   AGE
cockroachdb-0   1/1       Running   0          17h
cockroachdb-1   1/1       Running   0          17h
cockroachdb-2   1/1       Running   0          17h
```

Discussion

Labels are part of an object metadata. Any object in Kubernetes can be labeled. Labels are also used by Kubernetes itself for pod selection by deployments (for example, Recipe 3.1) and services (see ???).

Labels can be added manually (see Recipe 2.5) and are part of the object metadata. You can define labels in an object manifest like so:

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
  labels:
    tier: frontend
...<snip>
```

Once labels are present you can list them with `kubectl get`, noting the following:

- `-l` is the short form of `--selector` and will query object with a specified `key=value` pair.
- `--show-labels` will show all the labels of each object returned.

- `-L` will add a column to the results returned with the value of the specified label.
- Many object kinds support set-based querying, meaning you can state a query in a form, like must be labelled with X and/or Y. For example, a `kubectl get pods -l 'env in (production, development)'` would give you pods that are either in production or development.

For example, with two pods running one with label `run=barfoo` and the other one with label `run=foobar` you would get outputs similar to the following:

```
$ kubectl get pods --show-labels
NAME                      READY   ...    LABELS
barfoo-76081199-h3gwx     1/1     ...    pod-template-hash=76081199,run=barfoo
foobar-1123019601-6x9w1   1/1     ...    pod-template-hash=1123019601,run=foobar

$ kubectl get pods -Lrun
NAME                      READY   ...    RUN
barfoo-76081199-h3gwx     1/1     ...    barfoo
foobar-1123019601-6x9w1   1/1     ...    foobar

$ kubectl get pods -l run=foobar
NAME                      READY   ...
foobar-1123019601-6x9w1   1/1     ...
```

See Also

- Concept documentation labels.

2.7 Annotating a Resource With One Command

Problem

You want to annotate a resource with a generic, non-identifying key-value pair, possibly using non-human-readable data.

Solution

Use the `kubectl annotate` command.

```
$ kubectl annotate pods foobar description='something that you can use for auto
mation'
```

Annotation tend to be used for added automation of Kubernetes (see ???). For example in Kubernetes as of `v1.6.0`, annotating a deployment with the `kubernetes.io/change-cause` key will show the cause for change of the object in the `kubectl rollout history` output.

Discussion

For example when you create a deployment with the `kubectl run` command and your forget to use the `--record` option, you will notice that the *change-cause* in your rollout history is empty. To start recording the commands that cause changes of the deployment, you can annotate it. For example given a deployment foobar, you would annotate like with:

```
$ kubectl annotate deployment foobar kubernetes.io/change-cause="Reason for cre
ating a new revision"
```

Subsequent changes to the deployment will be recorded.

2.8 Listing Resources

Problem

You want list Kubernetes resources of a certain kind.

Solution

Use the `get` verb of `kubectl` along with the resource type:

```
# To list all pods:
$ kubectl get pods

# To list all services and deployments:
$ kubectl get services,deployments

# To list a specific deployment:
kubectl get deployment myfirstk8sapp

# To list all resources:
kubectl get all
```

Note that `kubectl get` is a very basic but extremely useful command to get a quick overview what is going on in the cluster, it's essentially the equivalent to `ps` on Unix.

Many resources have short names you can use with `kubectl`, saving you time and sanity. Here are some examples:

- configmaps (aka *cm*)
- daemonsets (aka *ds*)
- deployments (aka *deploy*)
- endpoints (aka *ep*)
- events (aka *ev*)
- horizontalpodautoscalers (aka *hpa*)
- ingresses (aka *ing*)
- namespaces (aka *ns*)
- nodes (aka *no*)
- persistentvolumeclaims (aka *pvc*)
- persistentvolumes (aka *pv*)
- pods (aka *po*)
- replicasets (aka *rs*)
- replicationcontrollers (aka *rc*)
- resourcequotas (aka *quota*)
- serviceaccounts (aka *sa*)
- services (aka *svc*)

2.9 Deleting Resources

Problem

You no longer need a certain resource and want to get rid of it.

Solution

Use the `delete` verb of `kubectl` along with the type and name of the resource you wish to delete:

```
$ kubectl get ns
NAME          STATUS    AGE
default       Active    2d
kube-public   Active    2d
kube-system   Active    2d
limitedns     Active    20m
```

```
$ kubectl delete namespace/limitedns
namespace "limitedns" deleted
```

Discussion

- Don't delete supervised objects such as pods by deployment directly, kill its supervisor
- cascading vs. simple (CRDs are cascading, RC, RS are)

2.10 Watching Resource Changes With `kubectl`

Problem

You want to watch the changes to Kubernetes objects in an interactive manner on the terminal.

Solution

The `kubectl` command has a `--watch` option which gives you this behavior. For example, to watch pods:

```
$ kubectl get pods --watch
```

Note that above is a blocking and auto-updating command, akin to `top`.

Discussion

The watch option is useful but sometimes not very reliable in terms of refreshing the screen correctly. Alternatively, you can use the watch command like so `watch kubectl get ...`.

2.11 Editing Resources With `kubectl`

Problem

You want to update a property of a Kubernetes resource.

Solution

Use the `edit` verb of `kubectl` along with the resource type:

```
$ kubectl run nginx --image=nginx
$ kubectl edit deploy/nginx

# Now edit the nginx deployment in your editor,
```

```
# for example, change replicas to '2'. Once you save
# you'll see something like:

deployment "nginx" edited
```

Discussion

Might have editor issues, use `EDITOR=vi`. Not all changes trigger a deployment. Some triggers have shortcuts, for example, if you want to change the image version a deployment uses, simply use `kubectl set image` which updates existing container images of resources (valid for deployments, replica sets/replication controllers, daemon sets, jobs, and simple pods).

2.12 Letting `kubectl` Explain Resources And Fields

Problem

You want to gain a deeper understanding of a certain resource, for example `service` and/or understand what exactly a certain field in a Kubernetes manifest such as `svc.spec.externalIPs` means, including default values and if it's required or optional.

Solution

Use the `explain` verb of `kubectl`

```
$ kubectl explain svc
DESCRIPTION:
Service is a named abstraction of software service (for example, mysql) consist-
ing of local port (for example 3306) that the proxy listens on, and the selec-
tor that determines which pods will answer requests sent through the proxy.

FIELDS:
   status       <Object>
     Most recently observed status of the service. Populated by the system.
     Read-only. More info: https://git.k8s.io/community/contributors/devel/
     api-conventions.md#spec-and-status/

   apiVersion   <string>
     APIVersion defines the versioned schema of this representation of an
     object. Servers should convert recognized schemas to the latest internal
     value, and may reject unrecognized values. More info:
     https://git.k8s.io/community/contributors/devel/api-
conventions.md#resources

   kind <string>
     Kind is a string value representing the REST resource this object
     represents. Servers may infer this from the endpoint the client submits
```

```
     requests to. Cannot be updated. In CamelCase. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#types-
kinds

   metadata     <Object>
     Standard object's metadata. More info:
     https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata

   spec <Object>
     Spec defines the behavior of a service. https://git.k8s.io/community/
     contributors/devel/api-conventions.md#spec-and-status/

$ kubectl explain svc.spec.externalIPs
FIELD: externalIPs <[]string>

DESCRIPTION:
     externalIPs is a list of IP addresses for which nodes in the cluster will
     also accept traffic for this service.  These IPs are not managed by
     Kubernetes.  The user is responsible for ensuring that traffic arrives at a
     node with this IP.  A common example is external load-balancers that are
not
     part of the Kubernetes system.
```

Discussion

The kubectl explain command pulls the description for resources and fields from the Swagger/OpenAPI definitions, exposed by the API server.

See Also

- Blog post: kubectl explain — #HeptioProTip

2.13 Building System Automation Using Resource Annotations

Problem

Solution

CHAPTER 3

Basics of Managing Workloads

In this chapter we present recipes that show you how to manage Kubernetes workloads including pods and deployments.

3.1 Creating A Deployment Using `kubectl run`

Problem

You want to quickly launch a long-running application such as a web server.

Solution

Use the `kubectl run` command, a generator that creates a deployment manifest on the fly. For example, to create a deployment that runs the Ghost micro-blogging platform do the following:

```
$ kubectl run ghost --image=ghost:0.9
$ kubectl get deploy/ghost
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
ghost     1         1         1            0           16s
```

Discussion

The `kubectl run` command can take a number of arguments to configure additional parameters of the deployments, for example:

- Set environment variables with `--env`.
- Define container ports `--port`.
- Define a command to run `--command`.

- Automatically create an associated service with `--expose`.
- Define the number of pods using `--replicas`.

Typical usages are as follows.

To launch Ghost serving on port `2368` and create a service along with it, do:

```
$ kubectl run ghost --image=ghost:0.9 --port=2368 --expose
```

To launch MySQL with the root password set, do:

```
$ kubectl run mysql --image=mysql:5.5 --env=MYSQL_ROOT_PASSWORD=root
```

To launch a busybox container and execute the command `sleep 3600` on start, do:

```
$ kubectl run myshell --image=busybox --command -- sh -c "sleep 3600"
```

See also `kubectl run --help` for more details about the available arguments.

3.2 Creating Objects From File Manifests

Problem

Rather than creating an object via a generator such as `kubectl run` you want to explicitly state its properties and then create it.

Solution

Use `kubectl create` like so:

```
$ kubectl create -f <manifest>
```

Discussion

You can point `kubectl create` to a URL instead or a filename in your local filesystem. For example, to create the frontend for the canonical Guestbook application, get the URL of the **raw** YAML that defines the application in a single manifest and do:

```
kubectl create -f https://raw.githubusercontent.com/kubernetes/kubernetes/
master/examples/guestbook/frontend-deployment.yaml
```

3.3 Writing A Pod Manifest From Scratch

Problem

You want to write a pod manifest from scratch and not use a generator such as `kubectl run`.

Solution

A Pod is a `/api/v1` object and like any other Kubernetes object it has *metadata* and a *specification*. Every Kubernetes object has the following parts:

- The API version, via the `apiVersion` field.
- The type, via the `kind` field.
- Some metadata, via the `metadata` field.
- The object specification, via the `spec` field.

The pod manifest contains an array of containers as well as an array of volumes. In its simplest form, a single container and no volume, it looks as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: foobar
spec:
  containers:
  - name: foobar
    image: nginx
```

Save this YAML manifest in a file called `foobar.yaml` and then use `kubectl` to create it:

```
$ kubectl create -f foobar.yaml
```

Discussion

Unless for very specific reasons, never create a pod on its own. Use deployments to supervise a pod.

See Also

- The Pod schema.

3.4 Launching Deployment Using A Manifest

Problem

You want to have full control over how a (long-running) app is launched and supervised.

Solution

Write a manifest using a deployment object in it. For the basics see also Recipe 3.3.

Let's say we have manifest file called `fancyapp.yaml` with the following content:

```
apiVersion:       extensions/v1beta1
kind:             Deployment
metadata:
  name:           fancyapp
spec:
  replicas:       5
  template:
    metadata:
      labels:
        app:      fancy
        env:      development
    spec:
      containers:
      - name:     sise
        image:    mhausenblas/simpleservice:0.5.0
        ports:
        - containerPort: 9876
        env:
        - name:   SIMPLE_SERVICE_VERSION
          value:  "0.9"
```

As you can see above, there are a couple of things you might want to specify explicitly when launching the app:

- Set the number of pods, that is the identical copies (`replicas`) that should be launched and supervised.
- Label it, such as with `env=development` (see also Recipe 2.5 and Recipe 2.6).
- Set environment variables, such as `SIMPLE_SERVICE_VERSION`.

Now we have a look at what the deployment entails:

```
$ kubectl create -f fancyapp.yaml
deployment "fancyapp" created

$ kubectl get deploy
NAME       DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
fancyapp   5         5         5            0           8s

$ kubectl get rs
NAME                  DESIRED   CURRENT   READY     AGE
fancyapp-1223770997   5         5         0         13s

$ kubectl get po
NAME                        READY     STATUS              RESTARTS   AGE
fancyapp-1223770997-18msl   0/1       ContainerCreating   0          15s
```

```
fancyapp-1223770997-1zdg4   0/1       ContainerCreating   0          15s
fancyapp-1223770997-6rqn2   0/1       ContainerCreating   0          15s
fancyapp-1223770997-7bnbh   0/1       ContainerCreating   0          15s
fancyapp-1223770997-qxg4v   0/1       ContainerCreating   0          15s

# and a few seconds later again:

$ kubectl get po
NAME                        READY     STATUS    RESTARTS   AGE
fancyapp-1223770997-18msl   1/1       Running   0          1m
fancyapp-1223770997-1zdg4   1/1       Running   0          1m
fancyapp-1223770997-6rqn2   1/1       Running   0          1m
fancyapp-1223770997-7bnbh   1/1       Running   0          1m
fancyapp-1223770997-qxg4v   1/1       Running   0          1m
```

Warning Deployment Deletion

When you want to get rid of the deployment and with it the replica set and the pods it supervises, execute a command like `kubectl delete deploy/fancyapp`. Do **not** try to delete individual pods as they will be recreated by the deployment. This is something that confuses beginners a lot.

Deployments allow you to scale the app (number of replicas) as well as roll out a new version as well as roll back to a previous version. They are in general good for stateless apps that require pods with identical characteristics.

Discussion

A deployment is a supervisor for pods and replica sets (RS), giving you fine-grained control over how and when a new pod version is rolled out as well as rolled back to a previous state. The RS and pods a deployment supervises are in general of no interest to you, unless, for example, when you need to debug a pod Recipe 6.3. In Figure 3-1> you see how you can move back and forth between deployment revisions.

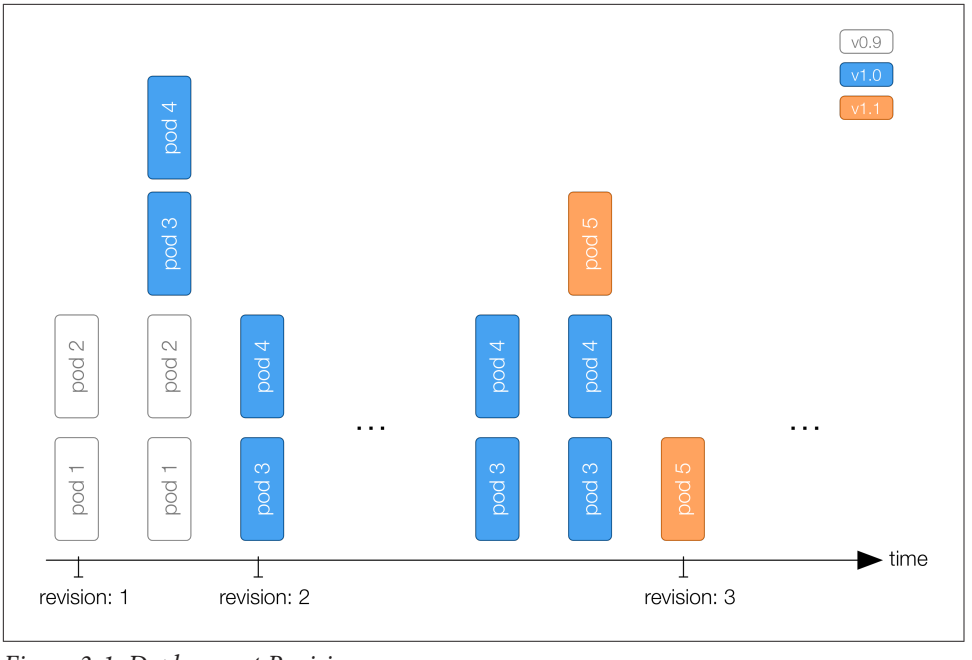


Figure 3-1. Deployment Revisions

Note that RS will, going forward, replace the original replication controller (RC) so it's a good thing to start thinking in terms of RSs rather than RCs. For now, the only difference is that RS support set-based labels/querying, but we can expect that there will be more features added to RS and RC will eventually be deprecated.

See Also

- Deployments in concepts.

CHAPTER 7

Monitoring and Logging

In this chapter we focus on recipes around monitoring and logging, both on the infrastructure as well as on the application level. In the context of Kubernetes, different roles typically have different scopes:

- *Administrator roles* such as cluster admins, networking operations folks, or namespace-level admins focus on infrastructure aspects. Exemplary questions might be: Are nodes healthy? Shall we add a worker node? What is the cluster-wide utilization? Are users close to their usage quotas?
- *Developer roles* mainly think and act in the context of their applications, which may well be—in the age of microservices—a handful to a dozen. For example, a person in a developer role might ask: Do I have enough resources allocated to run my app? How many replicas should I scale my app to? Do I have access to the right volumes and how full are they? Is one of my apps failing and if so, why?

We will first discuss recipes around cluster-internal monitoring leveraging Kubernetes Liveness and Readiness probes, then focus on monitoring with Heapster and Prometheus, and finally cover logging-related recipes.

7.1 Adding Liveness and Readiness Probes

Problem

You want to be able to automatically check if your app is healthy and let Kubernetes take actions if this it not the case.

Solution

To signal to Kubernetes how your app is doing, add a Liveness and a Readiness probe like described in the following.

Starting point is a deployment manifest `webserver.yaml`:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webserver
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:stable
        ports:
        - containerPort: 80
        livenessProbe:
          initialDelaySeconds: 2
          periodSeconds: 10
          httpGet:
            path: /
            port: 80
```

And now you can launch it and check the probes:

```
$ kubectl create -f webserver.yaml

$ kubectl get pods
NAME                         READY     STATUS    RESTARTS   AGE
webserver-4288715076-dk9c7   1/1       Running   0          2m

$ kubectl describe pod/webserver-4288715076-dk9c7
Name:           webserver-4288715076-dk9c7
Namespace:      default
Node:           node/172.17.0.128
...
Status:         Running
IP:             10.32.0.2
Controllers:    ReplicaSet/webserver-4288715076
Containers:
  nginx:
    ...
    Ready:          True
    Restart Count:  0
    Liveness:       http-get http://:80/ delay=2s timeout=1s period=10s #suc
```

```
    cess=1 #failure=0
    ...
```

Note that the output of the `kubectl describe` command has been edited down to the important bits; there's much more information available but it's not pertinent to our problem here.

Discussion

In order to verify if a container in a pod is healthy and ready to serve traffic, Kubernetes provides for a range of health checking mechanisms. Health checks or *probes* as they are called in Kubernetes, are defined on the container level, not on the pod level, and are carried out by two different components:

- The `kubelet` on each worker node uses the `livenessProbe` directive in the spec to determine when to restart a container. These liveness probes can help overcome ramp-up issues or deadlocks.
- A service, load-balancing a set of pods uses the `readinessProbe` directive to determine if a pod is ready and hence should receive traffic. If this it not the case it is excluded from the pool of endpoints of this service. Note that a pod is considered ready when all of its containers are ready.

When to use which probe? That depends on the behavior of the container, really. Use a liveness probe and a `restartPolicy` of either `Always` or `OnFailure` if your container can and should be killed and restarted if the probe fails. If you want to send traffic to a pod when use a respective readiness probe. Note that in this latter case, the readiness probe can be the same as the liveness probe.

See Also

- The pod lifecycle.
- Init containers (stable in 1.6 and above).

7.2 Enabling Heapster on Minikube To Monitor Resources

Problem

You want to use the `kubectl top` command in Minikube to monitor resources but Heapster is not running as shown below:

```
$ kubectl top
Display Resource (CPU/Memory/Storage) usage.

The top command allows you to see the resource consumption for nodes or pods.
```

```
...<snip>
$ kubectl top pods
Error from server (NotFound): the server could not find the requested resource \
                              (get services http:heapster:)
```

Solution

The latest versions of the `minikube` command includes an add-on manager, which lets you enable Heapster—as well as a few other add-ons such as an ingress controller—with a single command:

```
$ minikube addons enable heapster
```

Enabling the Heapster add-on triggers the creation of two pods in the `kube-system` namespace: one pod running Heapster and another pod running the Influxdb time series database along with a Grafana dashboard.

After some minutes, once the first metrics have been collected, the `kubectl top` command will return resource metrics as expected:

```
$ kubectl top node
NAME       CPU(cores)   CPU%      MEMORY(bytes)   MEMORY%
minikube   187m         9%        1154Mi          60%

$ kubectl top pods --all-namespaces
NAMESPACE     NAME                              CPU(cores)   MEMORY(bytes)
default       ghost-2663835528-fb044            0m           140Mi
kube-system   kube-dns-v20-4bkhn                3m           12Mi
kube-system   heapster-6j5m8                    0m           21Mi
kube-system   influxdb-grafana-vw9x1            23m          37Mi
kube-system   kube-addon-manager-minikube       47m          3Mi
kube-system   kubernetes-dashboard-scsnx        0m           14Mi
kube-system   default-http-backend-75m71        0m           1Mi
kube-system   nginx-ingress-controller-p8fmd    4m           51Mi
```

Now, you will also be able to access the Grafana dashboard and customize it to your liking:

```
$ minikube service monitoring-grafana -n kube-system
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Waiting, endpoint for service is not ready yet...
Opening kubernetes service kube-system/monitoring-grafana in default browser...
```

As a result of above command your default browser should automagically open and you should see something like depicted in Figure 7-1.

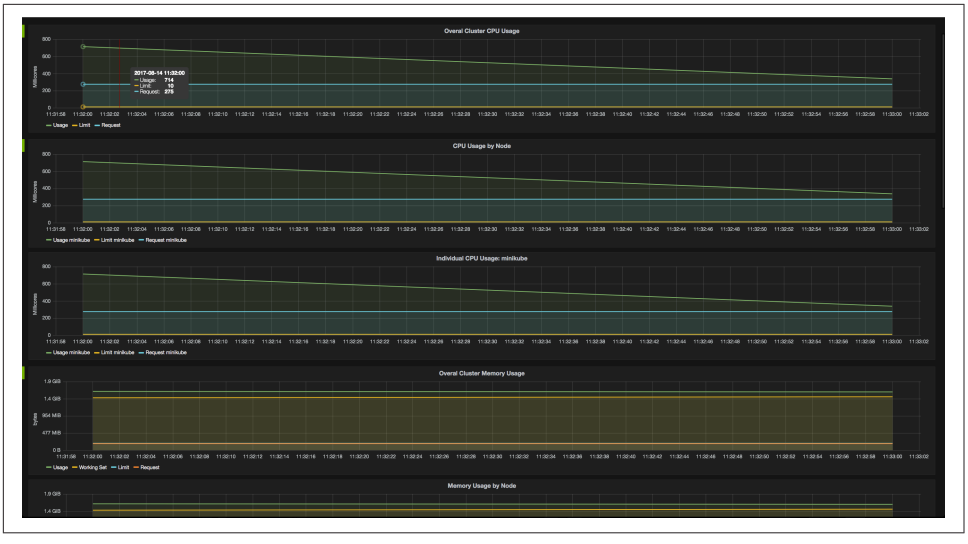


Figure 7-1. Screen Shot Of Grafana Dashboard, Showing Minikube Metrics

Note that you can drill into the metrics in Grafana at this point.

7.3 Using Prometheus On Minikube

Problem

You want to view and query the system and application metrics of your cluster in a central place.

Solution

Use Prometheus as follows:

1. Create a config map holding the Prometheus configuration.
2. Set up a service account for Prometheus and assign permissions via RBAC to the service account allowing access to all metrics.
3. Create an app consisting of 1. a deployment, 2. a service, and 3. an ingress resource for Prometheus so that you can access it via browser from outside of the cluster.

First, we set up the Prometheus configuration via a config map. We will be using this later in the Prometheus app. So, create a file `prometheus.yml`, holding the Prometheus configuration, with the following content:

```
global:
  scrape_interval:     5s
  evaluation_interval: 5s
scrape_configs:
- job_name:            'kubernetes-nodes'
  scheme:              https
  tls_config:
    ca_file:           /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
    server_name:       'gke-k8scb-default-pool-be16f9ee-522p'
    insecure_skip_verify: true
  bearer_token_file:   /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role:              node
  relabel_configs:
  - action:            labelmap
    regex:             __meta_kubernetes_node_label_(.+)
- job_name:            'kubernetes-cadvisor'
  scheme:              https
  tls_config:
    ca_file:           /var/run/secrets/kubernetes.io/serviceaccount/ca.crt
  bearer_token_file:   /var/run/secrets/kubernetes.io/serviceaccount/token
  kubernetes_sd_configs:
  - role:              node
  relabel_configs:
  - action:            labelmap
    regex:             __meta_kubernetes_node_label_(.+)
  - target_label:      __address__
    replacement:       kubernetes.default.svc:443
  - source_labels:     [__meta_kubernetes_node_name]
    regex:             (.+)
    target_label:      __metrics_path__
    replacement:       /api/v1/nodes/${1}:4194/proxy/metrics
```

Now that we have the Prometheus configuration file `prometheus.yml` we can use it to create a config map like so:

```
$ kubectl create configmap prom-config-cm --from-file=prometheus.yml
```

Next, we set up the Prometheus service account and role binding (permissions) in a manifest file `prometheus-rbac.yaml` as follows:

```
apiVersion:  v1
kind:        ServiceAccount
metadata:
  name:      prometheus
  namespace: default
---
apiVersion:  rbac.authorization.k8s.io/v1beta1
kind:        ClusterRoleBinding
metadata:
  name:      prometheus
roleRef:
  apiGroup:  rbac.authorization.k8s.io
```

```
  kind:      ClusterRole
  name:      cluster-admin
subjects:
- kind:      ServiceAccount
  name:      prometheus
  namespace: default
```

Using the above manifest `prometheus-rbac.yaml` we can now create the service account and role binding:

```
$ kubectl create -f prometheus-rbac.yaml
```

Now that we have all the prerequisites sorted (configuration and access permissions) we can move on to the Prometheus app itself. Remember, the app entails a deployment, a service, and the ingress resource as well as uses the config map and service account we created in the previous step.

Let's define the Prometheus app manifest in `prometheus-app.yaml`:

```
kind:                                   Deployment
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 prom
  namespace:                            default
  labels:
    app:                                prom
spec:
  replicas:                             1
  selector:
    matchLabels:
      app:                              prom
  template:
    metadata:
      name:                             prom
      labels:
        app:                            prom
    spec:
      serviceAccount:                   prometheus
      containers:
      - name:                           prom
        image:                          prom/prometheus
        imagePullPolicy:                Always
        volumeMounts:
        - name:                         prometheus-volume-1
          mountPath:                    "/prometheus"
        - name:                         prom-config-volume
          mountPath:                    "/etc/prometheus/"
      volumes:
      - name:                           prometheus-volume-1
        emptyDir:                       {}
      - name:                           prom-config-volume
        configMap:
          name:                         prom-config-cm
```

```
          defaultMode:                    420
---
kind:                                     Service
apiVersion:                               v1
metadata:
  name:                                   prom-svc
  labels:
    app:                                  prom
spec:
  ports:
  - port:                                 80
    targetPort:                           9090
  selector:
    app:                                  prom
  type:                                   LoadBalancer
  externalTrafficPolicy:                  Cluster
---
kind:                                     Ingress
apiVersion:                               extensions/v1beta1
metadata:
  name:                                   prom-public
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host:
    http:
      paths:
      - path:                             /
        backend:
          serviceName:                    prom-svc
          servicePort:                    80
```

And now create the app from the above manifest:

```
$ kubectl create -f prometheus-app.yaml
```

Great, you did it! You created a fullfledged app and wow you can access Prometheus via `$MINISHIFT_IP/graph`, for example, `https://192.168.99.100/graph` and should see something like in Figure 7-2.

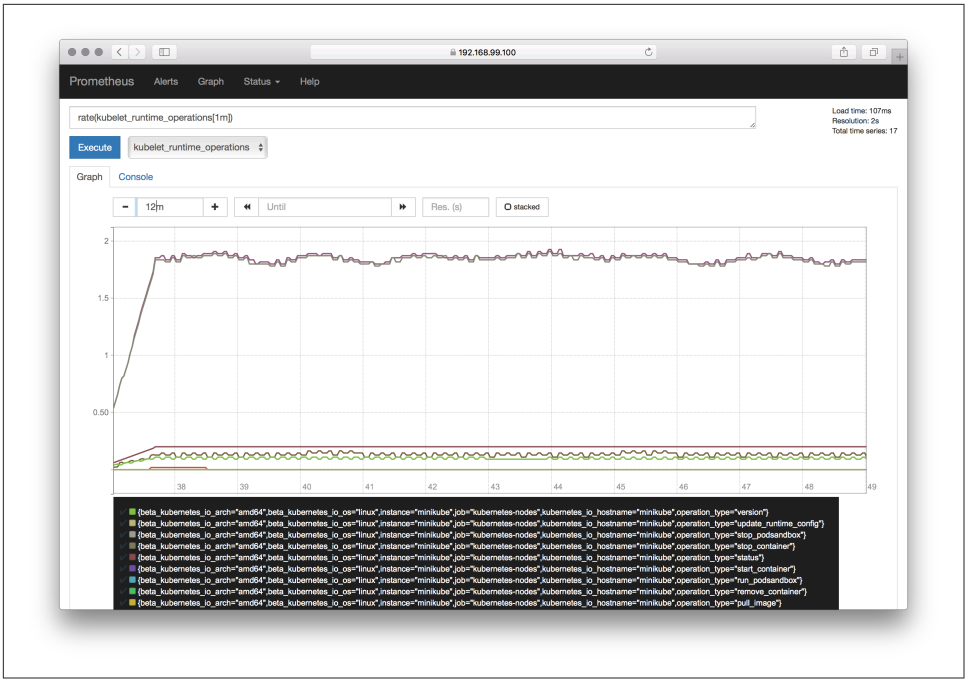


Figure 7-2. Prometheus Screen Shot

Discussion

Prometheus is a powerful and flexible monitoring and alerting system. You can use it, or better one of the many instrumentation libraries, to make your own app report higher-level metrics, for example, the number of transactions performed just like the way, say, a kubelet reports the CPU usage.

While Prometheus is fast and scalable, you probable want to use something else to visualize the metrics. The canonical way to do this is to connect it with Grafana.

Warning Kubernetes 1.7.0 to 1.7.3

There is a known issue with Kubernetes version 1.7.0 through 1.7.2, as the kubelet's behavior changed concerning exposing container metrics.

Note that the Prometheus config shown in the solution is valid for 1.7.0 to 1.7.2 and if you're using a later version such as 1.7.3 you want to check out the example Prometheus configuration file for more details what you need to change.

Note that above solution is not limited to Minikube. In fact, as long as you can create the service account (that is, you have sufficient rights to give Prometheus the necessary permissions) you can apply the same solution to environments such as GKE, ACS or OpenShift.

See Also

- Instrumentation in Prometheus docs
- Grafana with Prometheus in Prometheus docs

7.4 Using Elasticsearch-Fluentd-Kibana (EFK) On Minikube

Problem

You want to view and query the logs of all of the apps in your cluster in a central place.

Solution

Use Elasticsearch, Fluentd, and Kibana as described in the following.

As a preparation, make sure Minikube has enough resources assigned, for example, use `--cpus=4 --memory=4000` and make sure the ingress add-on is enabled, like so:

```
$ minikube start
Starting local Kubernetes v1.7.0 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.

$ minikube addons list # is ingress add-on enabled?

$ minikube addons enable ingress # if not enabled, enable it
```

Next, create a manifest file `efk-logging.yaml` with the following content:

```
kind:                                       Ingress
apiVersion:                                 extensions/v1beta1
metadata:
  name:                                     kibana-public
  annotations:
    ingress.kubernetes.io/rewrite-target: /
spec:
```

```
  rules:
  - host:
    http:
      paths:
      - path:                           /
        backend:
          serviceName:                  kibana
          servicePort:                  5601
---
kind:                                   Service
apiVersion:                             v1
metadata:
  labels:
    app:                                efk
  name:                                 kibana
spec:
  ports:
  - port:                               5601
  selector:
    app:                                efk
---
kind:                                   Deployment
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 kibana
spec:
  replicas:                             1
  template:
    metadata:
      labels:
        app:                            efk
    spec:
      containers:
      - env:
        - name:                         ELASTICSEARCH_URL
          value:                        http://elasticsearch:9200
        name:                           kibana
        image:                          docker.elastic.co/kibana/kibana:5.5.1
        ports:
          - containerPort:              5601
---
kind:                                   Service
apiVersion:                             v1
metadata:
  labels:
    app:                                efk
  name:                                 elasticsearch
spec:
  ports:
  - port:                               9200
  selector:
    app:                                efk
```

```
---
kind:                                   Deployment
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 es
spec:
  replicas:                             1
  template:
    metadata:
      labels:
        app:                            efk
    spec:
      containers:
      - name:                           es
        image:                          docker.elastic.co/elasticsearch/
elasticsearch:5.5.1
        ports:
        - containerPort:                9200
        env:
        - name:                         ES_JAVA_OPTS
          value:                        "-Xms256m -Xmx256m"
---
kind:                                   DaemonSet
apiVersion:                             extensions/v1beta1
metadata:
  name:                                 fluentd
spec:
  template:
    metadata:
      labels:
        app:                            efk
      name:                             fluentd
    spec:
      containers:
      - name:                           fluentd
        image:                          gcr.io/google_containers/fluentd-
elasticsearch:1.3
        env:
         - name:                        FLUENTD_ARGS
           value:                       -qq
        volumeMounts:
         - name:                        varlog
           mountPath:                   /varlog
         - name:                        containers
           mountPath:                   /var/lib/docker/containers
      volumes:
         - hostPath:
             path:                      /var/log
           name:                        varlog
         - hostPath:
             path:                      /var/lib/docker/containers
           name:                        containers
```

Now you can launch the EFK stack:

```
$ kubectl create -f efk-logging.yaml
```

Once everything has started up, log in to Kibana with the following credentials:

- Username: `kibana`
- Password: `changeme`

Visit the `Discover` tab at `https://$IP/app/kibana#/discover?_g=()` and start exploring the logs from here.

If you want to clean up and or restart the EFK stack, use the following:

```
$ kubectl delete deploy/es && \
  kubectl delete deploy/kibana && \
  kubectl delete svc/elasticsearch && \
  kubectl delete svc/kibana && \
  kubectl delete ingress/kibana-public && \
  kubectl delete daemonset/fluentd
```

Discussion

The log shipping can also be done using Logstash rather than Fluentd. Since Fluentd is a CNCF project and gains a lot of traction, we've used it in the solution.

Note that Kibana can take some time to come up and you might need to reload the Web app a couple of times until you get to the configuration bits.

See Also

- *https://medium.com/@manoj.bhagwat60/to-centralize-your-docker-logs-with-fluentd-and-elasticsearch-on-kubernetes-42d2ac0e8b6c*
- *https://github.com/Skillshare/kubernetes-efk*
- *https://github.com/kayrus/elk-kubernetes*

CHAPTER 8
Ecosystem

In this chapter we have a look at the wider Kubernetes ecosystem, that is, software in the Kubernetes incubator and related projects, such as Helm or kompose.

8.1 Installing Helm, The Kubernetes Package Manager

Problem

You do not want to write all the Kubernetes manifests by hand. Instead you would like to use a package, search for it in a repository, download and install it with a command line interface.

Solution

Use Helm. Helm is the Kubernetes package manager, it defines a Kubernetes package as a set of manifests and some metadata. The manifests are actually templates. The values in the templates are filled when the package is instantiated by Helm. A Helm package is called a Chart.

Helm has a client side CLI, called `helm` and a server called `tiller`. You interact with charts using the `helm`, `tiller` runs within your Kubernetes cluster as a regular Kubernetes deployment.

You can build Helm from source or download it from the Github release page, extract the archive and move the `helm` binary in your PATH. For example, on macOS, for the `v2.4.1` release of Helm, do:

```
$ wget https://storage.googleapis.com/kubernetes-helm/helm-v2.4.1-darwin-
amd64.tar.gz

$ tar -xvf helm-v2.4.1-darwin-amd64.tar.gz
```

```
$ sudo mv darwin-amd64/64 /usr/local/bin

$ helm version
```

Now that the `helm` command is in your `$PATH`, you can use it to start the server side `tiller` on your Kubernetes cluster. Below we use Minikube as an example:

```
$ kubectl get nodes
NAME       STATUS    AGE       VERSION
minikube   Ready     4m        v1.6.0

$ helm init
$HELM_HOME has been configured at /Users/sebgoa/.helm.

Tiller (the helm server side component) has been installed into your Kubernetes
Cluster.
Happy Helming!

$ kubectl get pods --all-namespaces | grep tiller
kube-system   tiller-deploy-1491950541-4kqxx   0/1  ContainerCreating  0  1s
```

You're all set now and can install one of the over 100 packages available.

8.2 Using Helm to Install Applications

Problem

You installed the `helm` command (see Recipe 8.1) and now you would like to search for charts and deploy them.

Solution

By default, Helm comes with some Chart repositories configured. These repositories are maintained by the community. The source of the Charts is available on GitHub. There are over 100 charts available.

For example, let's assume you would like to deploy Redis. You will search for `redis` in the Helm repositories and then be able to install it. Helm will take the chart and create an instance of it called a release.

Let's first verify that `tiller` is running and that you have the default repositories configured:

```
$ kubectl get pods --all-namespaces | grep tiller
kube-system   tiller-deploy-1491950541-4kqxx   1/1   Running   0   3m

$ helm repo list
NAME    URL
stable  http://storage.googleapis.com/kubernetes-charts
```

```
local   http://localhost:8879/charts
testing http://storage.googleapis.com/kubernetes-charts-testing
```

You can now search for a Redis package:

```
$ helm search redis
WARNING: Deprecated index file format. Try 'helm repo update'
NAME                    VERSION        DESCRIPTION
stable/redis            0.5.1   Open source, advanced key-value store. It is
of...
testing/redis-cluster   0.0.5   Highly available Redis cluster with multiple
se...
testing/redis-standalone        0.0.1   Standalone Redis Master
stable/sensu            0.1.2   Sensu monitoring framework backed by the
Redis ...
testing/example-todo    0.0.6   Example Todo application backed by Redis
```

We will pick the stable repository. We can use `helm install` to create a release like so:

```
$ helm install stable/redis
```

Helm will create all the Kubernetes objects defined in the chart. For example, a secret, a PVC, a service, and/or a deployment. The association of all these objects will make a Helm release that you can manage as a single unit.

The end result however—in this case—is that we will have a `redis` pod running. Below we show the release and the associated pod:

```
$ helm ls
NAME          REVISION  UPDATED                   STATUS   CHART       NAMESPACE
broken-badger   1               Fri May 12 11:50:43 2017        DEPLOYED
redis-0.5.1     default

$ kubectl get pods
NAME                                  READY     STATUS    RESTARTS   AGE
broken-badger-redis-4040604371-tcn14   1/1       Running   0          3m
```

To learn more about Helm charts and how to create your own charts, see Recipe 8.3.

See Also

- *https://kubeapps.com*

8.3 Creating Your Own Chart To Package Your Application with Helm

Problem

You have written an application with multiple Kubernetes manifests and would like to package it as a Helm chart.

Solution

TO BE DONE

8.4 Converting Your Docker Compose Files To Kubernetes Manifests

Problem

You started using containers with Docker and wrote some Docker compose files to define your multi-container application. Now you would like to start using Kubernetes and wonder if and how you can re-use your Docker compose files.

Solution

Use `kompose`, a CLI tool that converts your Docker compose files into Kubernetes manifests.

To start, download `kompose` from the GitHub release page and move it to your `$PATH`, for convenience.

For example, on macOS, do the following:

```
$ wget https://github.com/kubernetes-incubator/kompose/releases/download/v0.6.0/
kompose-darwin-amd64

$ sudo mv kompose-darwin-amd64 /usr/local/bin/kompose

$ sudo chmod +x /usr/local/bin/kompose

$ kompose version
0.6.0 (ae4ef9e)
```

Given the following Docker compose file that starts a `redis` container:

```
version: '2'

services:
  redis:
```

```
    image: redis
    ports:
    - "6379:6379"
```

You can automatically convert this into Kubernetes manifests with the following command:

```
$ kompose convert --stdout
```

The manifests will be printed to `stdout` and you will see a Kubernetes service and a deployment as a result. To create these objects automatically, you can use the Docker compose compliant command `up` like so:

```
$ kompose up
```

Some Docker compose directives are not converted to Kubernetes. In this case `kompose` prints out a warning informing you that the conversion did not happen.

While in general it doesn't cause problems, it is possible that the conversion may not result in a working manifest in Kubernetes. This is expected as this type of transformation can not be perfect. However, it gets you close to a working Kubernetes manifest. Most notably the way to handle volumes and network isolation will typically require manual, custom work from your side.

Discussion

The `kompose` main commands are `convert`, `up` and `down`. Each command has a detailed help with the CLI using the `--help` option.

By default, `kompose` converts your Docker services into a Kubernetes deployment and associated service. You can also specify the use of a `DaemonSet` (see ???) or you can use OpenShift-specific objects such as a deployment config.

CHAPTER 9

Maintenance And Troubleshooting

In this chapter you find recipes that deal with cluster and pod-level maintenance as well as troubleshooting: from debugging pods and containers to node maintenance to checking routes and services, as well as how to manage etcd.

9.1 Enabling Autocomplete For `kubectl`

Problem

It is cumbersome to type full commands and arguments for the `kubectl` command, so you want an autocomplete function for it.

Solution

Enable autocompletion for `kubectl`.

See Also

- Overview of kubectl
- kubectl Cheat Sheet

9.2 Understanding And Parsing Resource Statuses

Problem

You want to react based on the status of a resource, say, a pod in a script or in another automated environment like a CI/CD pipeline.

Solution

Use `kubectl get $KIND/$NAME -o json` and parse the JSON output using one of the following two methods

Using `jq` To Parse Resource Status

If you have the JSON query util `jq` installed you can use it as shown in the following to parse the resource status. Let's assume we have a pod called `jump` and want to known what Quality of Service (QoS) class the pod is in:

```
$ kubectl get po/jump -o json | jq --raw-output .status.qosClass
BestEffort
```

Note that the `--raw-output` argument for `jq` will show the raw value and that `.status.qosClass` is the expression that matches the respective sub-field.

Another status query could be around the events or state transitions:

```
$ kubectl get po/jump -o json | jq .status.conditions
[
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-28T08:06:19Z",
    "status": "True",
    "type": "Initialized"
  },
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-31T08:21:29Z",
    "status": "True",
    "type": "Ready"
  },
  {
    "lastProbeTime": null,
    "lastTransitionTime": "2017-08-28T08:06:19Z",
    "status": "True",
    "type": "PodScheduled"
  }
]
```

Of course, these queries are not limited to pods, you can apply it to any resource, for example, to query the revisions of a deployment:

```
$ kubectl get deploy/prom -o json | jq .metadata.annotations
{
  "deployment.kubernetes.io/revision": "1"
}
```

Or, you can list all the endpoints that make up a service:

```
$ kubectl get ep/prom-svc -o json | jq '.subsets'
[
```

```
  {
    "addresses": [
      {
        "ip": "172.17.0.4",
        "nodeName": "minikube",
        "targetRef": {
          "kind": "Pod",
          "name": "prom-2436944326-pr60g",
          "namespace": "default",
          "resourceVersion": "686093",
          "uid": "eee59623-7f2f-11e7-b58a-080027390640"
        }
      }
    ],
    "ports": [
      {
        "port": 9090,
        "protocol": "TCP"
      }
    ]
  }
]
```

Now that we've seen `jq` in action let's move on to a method that doesn't require external tooling, that is, the built-in feature of using Go templates.

Using A Go Template To Parse Resource Status

The Go programming language defines templates in package `text/template` that can be used for any kind of text or data transformation and `kubectl` has built-in support for it. For example, to list all the container images used in the current namespace, do:

```
$ kubectl get pods -o go-template --template="{{range .items}}{{range .spec.con
tainers}}{{.image}} {{end}}{{end}}"
busybox prom/prometheus
```

See Also

- jq manual
- jq playground to try out queries without installing `jq`
- Go templates docs

9.3 Debugging Pods

Problem

You've a situation where a pod is either not starting up as expected or fails after some time.

Solution

To systematically approach the cause of the problem, enter an OODA loop until you've discovered and fixed it:

1. *Observe*: What do you see in the container logs? What events do you see? How is the network connectivity or setup (firewall)?
2. *Orient*: Formulate a set of plausible hypotheses, stay as open-minded as possible, don't jump to conclusions.
3. *Decide*: Pick one of the hypotheses.
4. *Act*: Test the hypothesis and if confirmed you're done, otherwise continue with step 1.

Let's have a look at a concrete example where a pod fails. Create a manifest called `unhappy-pod.yaml` with this content:

```
apiVersion:       extensions/v1beta1
kind:             Deployment
metadata:
  name:           unhappy
spec:
  replicas:       1
  template:
    metadata:
      labels:
        app:      nevermind
    spec:
      containers:
      - name:     shell
        image:    busybox
        command:
        - "sh"
        - "-c"
        - "echo I will just print something here and then exit"
```

Now, when you launch that above deployment and look at the pod it creates, you'll see it's unhappy:

```
$ kubectl create -f unhappy-pod.yaml
deployment "unhappy" created
```

```
$ kubectl  get po
NAME                          READY     STATUS             RESTARTS   AGE
unhappy-3626010456-4j251   0/1       CrashLoopBackOff   1          7s

$ kubectl describe po/unhappy-3626010456-4j251
Name:           unhappy-3626010456-4j251
Namespace:      default
Node:           minikube/192.168.99.100
Start Time:     Sat, 12 Aug 2017 17:02:37 +0100
Labels:         app=nevermind
                pod-template-hash=3626010456
Annotations:    kubernetes.io/created-by={"kind":"SerializedReference","apiVer
sion":"v1","reference":
{"kind":"ReplicaSet","namespace":"default","name":"unhappy-3626010456","uid":
"a9368a97-7f77-11e7-b58a-080027390640"...
Status:         Running
IP:             172.17.0.13
Created By:     ReplicaSet/unhappy-3626010456
Controlled By:  ReplicaSet/unhappy-3626010456
...
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-rlm2s:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-rlm2s
    Optional:   false
QoS Class:      BestEffort
Node-Selectors: <none>
Tolerations:    <none>
Events:
  FirstSeen     LastSeen        Count   From                    SubObject-
Path           Type            Reason                  Message
  ---------     --------        -----   ----
-------------           --------        ------                  -------
  25s           25s             1       default-
scheduler                               Normal          Scheduled
Successfully assigned unhappy-3626010456-4j251 to
minikube
  25s           25s             1       kubelet, mini-
kube                               Normal          SuccessfulMountVolume
MountVolume.SetUp succeeded for volume "default-to
ken-rlm2s"
  24s           5s              3       kubelet, minikube       spec.contain-
ers{shell}  Normal          Pulling                 pulling image "busybox"
  22s           3s              3       kubelet, minikube       spec.contain-
ers{shell}  Normal          Pulled                  Successfully pulled image
"busybox"
  22s           3s              3       kubelet, minikube       spec.contain-
```

```
ers{shell}  Normal          Created                 Created container
  22s           3s              3       kubelet, minikube       spec.contain-
ers{shell}  Normal          Started                 Started container
  19s           2s              3       kubelet, minikube       spec.contain-
ers{shell}  Warning         BackOff                 Back-off restarting failed
container
  19s           2s              3       kubelet, mini-
kube                            Warning         FailedSync
Error syncing pod
```

As you can see above, Kubernetes considers this pod as not ready to serve traffic as it encountered an *error syncing pod*.

Another way to observe this is using the Kubernetes dashboard:

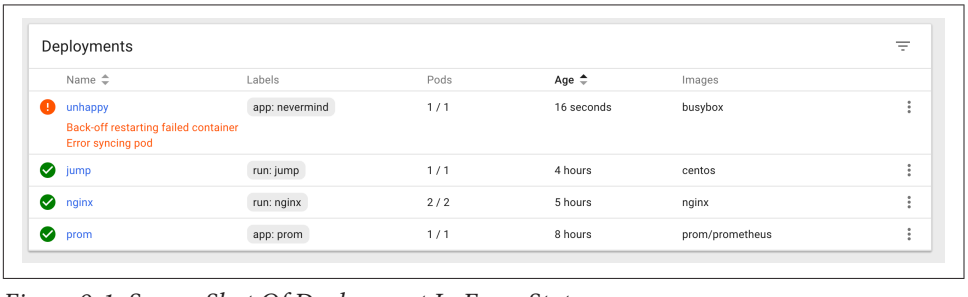


Figure 9-1. Screen Shot Of Deployment In Error State

As well as the supervised `ReplicaSet` and the pod:

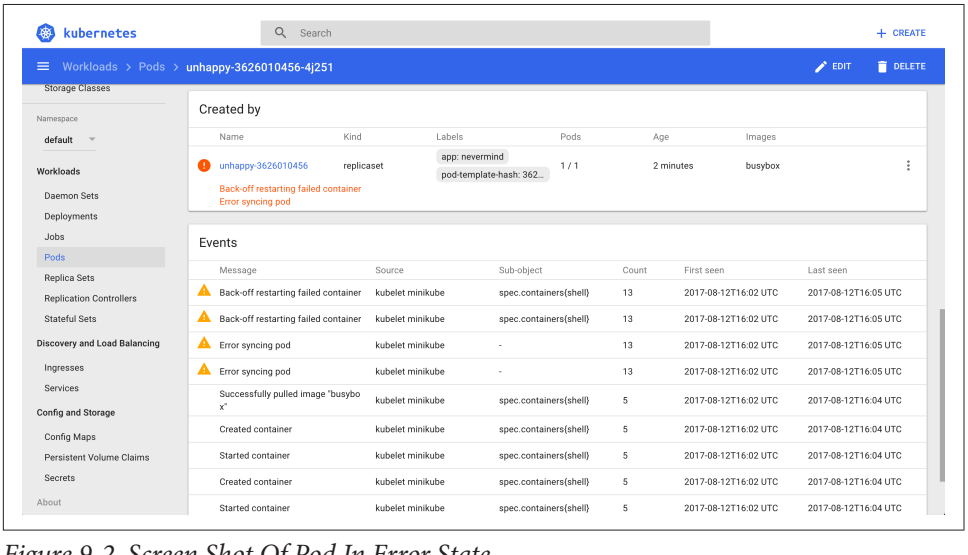


Figure 9-2. Screen Shot Of Pod In Error State

Discussion

An issue, be it a pod failing or a node behaving strangely, can have many different causes, here are some things you want to check before suspecting software bugs:

- Is the manifest correct? Check with the Kubernetes JSON schema
- Does the container run standalone, locally, that is, outside of Kubernetes?
- Can Kubernetes reach the container registry and actually pull the container image?
- Can the nodes talk to each other?
- Can the nodes reach the master?
- Is DNS available in the cluster?
- Are there sufficient resources available on nodes?
- Did you restrict the container resource usage?

See Also

- Troubleshoot Applications
- Application Introspection and Debugging
- Debug Pods and Replication Controllers
- Debug Services
- Troubleshoot Clusters

9.4 Getting A Detailed Snapshot Of The Cluster State

Problem

You want to get a detailed snapshot of the overall cluster state for orientation, auditing or troubleshooting purposes.

Solution

Use the `kubectl cluster-info dump` for it. For example, to create a dump of the cluster state in a sub-directory `cluster-state-2017-08-13`, do:

```
$ kubectl cluster-info dump --all-namespaces --output-directory=$PWD/cluster-
state-2017-08-13

$ tree ./cluster-state-2017-08-13
.
```

```
├── default
│   ├── cockroachdb-0
│   │   └── logs.txt
│   ├── cockroachdb-1
│   │   └── logs.txt
│   ├── cockroachdb-2
│   │   └── logs.txt
│   ├── daemonsets.json
│   ├── deployments.json
│   ├── events.json
│   ├── jump-1247516000-sz87w
│   │   └── logs.txt
│   ├── nginx-4217019353-462mb
│   │   └── logs.txt
│   ├── nginx-4217019353-z3g8d
│   │   └── logs.txt
│   ├── pods.json
│   ├── prom-2436944326-pr60g
│   │   └── logs.txt
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
├── kube-public
│   ├── daemonsets.json
│   ├── deployments.json
│   ├── events.json
│   ├── pods.json
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
├── kube-system
│   ├── daemonsets.json
│   ├── default-http-backend-wdfwc
│   │   └── logs.txt
│   ├── deployments.json
│   ├── events.json
│   ├── kube-addon-manager-minikube
│   │   └── logs.txt
│   ├── kube-dns-910330662-dvr9f
│   │   └── logs.txt
│   ├── kubernetes-dashboard-5pqmk
│   │   └── logs.txt
│   ├── nginx-ingress-controller-d2f2z
│   │   └── logs.txt
│   ├── pods.json
│   ├── replicasets.json
│   ├── replication-controllers.json
│   └── services.json
└── nodes.json
```

9.5 Adding Kubernetes Worker Nodes

Problem

You need to add a worker node to your Kubernetes cluster.

Solution

Provision a new machine depending on your environment, for example in a bare metal environment you might need to physically install a new server in a rack, in a public cloud setting you need to create a new VM, etc. and then install the three components that make up a Kubernetes worker node:

1. *kubelet*: The node manager and supervisor for all pods, no matter if they're controlled by the API server or running locally, such as static pods. Note that the `kubelet` is the final arbiter of what pods can or can not run on a given node and takes care of:
   - Reporting node and pod statuses to the API server.
   - Periodically executing liveness probes.
   - Mounting the pod volumes and downloading secrets.
   - Controlling the container runtime (see below).
2. *Container Runtime*: Is responsible for downloading container images and running the containers. Initially, this was hardwired to the Docker engine but nowadays it is a plug-able system based on the Container Runtime Interface (CRI), so you can for example use CRI-O, rather than Docker.
3. *kube-proxy*: This process dynamically configures IPtables rules on the node to enable the Kubernetes service abstraction (redirecting the VIP to the endpoints, one or more pods representing the service).

The actual installation of the components depends heavily on your environment and install method used (cloud, kubeadm, etc.). In general, the components have the following parameters:

- `kubelet`, see kubelet reference
- `kube-proxy` see kube-proxy reference

Discussion

Worker nodes, unlike other Kubernetes resources such as a deployment or a service are not directly created by the Kubernetes control plane but only managed by it. That

means, when Kubernetes creates a node, it actually only creates an object that **represents** the worker node: it validates the node by health checks based on the node's `metadata.name` field and if the node is valid, that is, all necessary components are running, it is considered part of the cluster, otherwise it will be ignored for any cluster activity until it becomes valid.

See Also

- In the Kubernetes Design and Architecture document, the Kubernetes Node section
- Master-Node communication
- Container Runtime Interface
- CRI-O
- Static Pods

9.6 Draining Kubernetes Nodes For Maintenance

Problem

You need to carry out maintenance on a node, for example apply a security patch or upgrade the operating system.

Solution

Use the `kubectl drain` command, for example, to do maintenance on node `123-worker`:

```
$ kubectl drain 123-worker
```

When you are ready to put the node back into service, use `kubectl uncordon 123-worker`, which will make the node schedulable again.

Discussion

What the `kubectl drain` command does is to first mark the specified node unschedulable to prevent new pods from arriving (essentially a `kubectl cordon`). Then, it evicts the pods if the API server supports eviction. Otherwise, it will use normal `kubectl delete` to delete the pods. The Kubernetes docs have a concise sequence diagram of the steps, depicted in Figure 9-3.

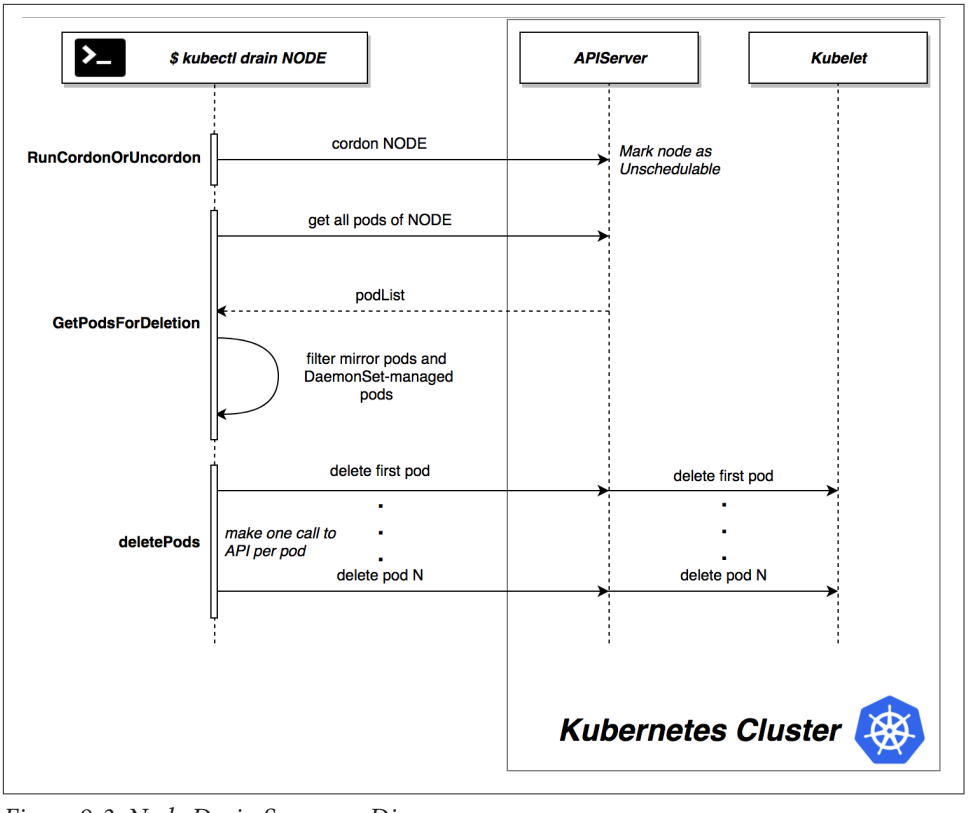


Figure 9-3. Node Drain Sequence Diagram

The *kubectl drain* commands evicts or deletes all pods except mirror pods (which cannot be deleted through the API server). For pods supervised by a `DaemonSet`, drain will not proceed without using `--ignore-daemonsets`, and regardless it will not delete any DaemonSet-managed pods, because those pods would be immediately replaced by the DaemonSet controller, which ignores unschedulable markings.

Tip Termination

Drain waits for graceful termination, so you should **not** operate on this node until the *kubectl drain* command has completed. Note that `kubectl drain $NODE --force` will also evict pods not managed by a RC, RS, job, `DaemonSet` or `StatefulSet`.

See Also

- Safely Drain a Node while Respecting Application SLOs.

- kubectl reference docs

9.7 Managing `etcd`

Problem

You need to access etcd to back it up or verify the cluster state directly.

Solution

Get access to etcd and query it, either using `curl` or etcdctl. For example, in the context of Minikube (and having `jq` installed in Minikube):

```
$ minikube ssh

$ curl 127.0.0.1:2379/v2/keys/registry | jq .
{
  "action": "get",
  "node": {
    "key": "/registry",
    "dir": true,
    "nodes": [
      {
        "key": "/registry/persistentvolumeclaims",
        "dir": true,
        "modifiedIndex": 241330,
        "createdIndex": 241330
      },
      {
        "key": "/registry/apiextensions.k8s.io",
        "dir": true,
        "modifiedIndex": 641,
        "createdIndex": 641
      },
...
```

Note that in certain environments where the control plane is managed for you, notable in GKE, you **can not** access etcd. This is by design and there's no workaround.

Discussion

In Kubernetes, etcd is a component of the control plane. The API Server Recipe 2.1 is stateless and is the only Kubernetes component that directly communicates with etcd, the distributed storage component that manages the cluster state. Essentially, etcd is a key-value store, where in etcd2 the keys formed a hierarchy and with the introduction of etcd3 this was replaced with a flat model while maintaining backwards compatibility concerning hierarchical keys.

Up until Kubernetes 1.5.2 we used etcd2 and from then on switched to etcd3. In Kubernetes 1.5.x etcd3 is still used in v2 API mode and going forward this is changing to the etcd v3 API. While from a developer's point of view this doesn't have any implications, because the API Server takes care of abstracting the interactions away, as an admin you want to pay attention to which etcd version in which API mode is used.

See Also

- Operating etcd clusters for Kubernetes
- Kubernetes Deep Dive: API Server – Part 2
- Accessing Localkube Resources From Inside A Pod: Example etcd

About the Authors

Sébastien Goasguen built his first compute cluster in the late 90s and takes pride in having completed his Ph-D thanks to Fortran 77 and partial differential equations. His struggles with parallel computers, led him to work on making computing a utility and focus on Grids and later clouds. Fifteen years later, he secretly hopes that containers and Kubernetes will let him get back to writing applications.

He is currently the senior director of cloud technologies at Bitnami, where he leads the Kubernetes efforts. He founded Skippbox, a Kubernetes startup in late 2015. While at Skippbox he created several open source software and tools to enhance the user experience of Kubernetes. He is a member of the Apache Software Foundation and a former vice president of Apache CloudStack. Sébastien focuses on the cloud ecosystem and has contributed to dozens of open source projects. He is the author of the Docker cookbook, and avid blogger and an on-line instructor of Kubernetes concepts for Safari subscribers.

Michael Hausenblas is a Developer Advocate at Red Hat. He covers Go, Kubernetes, and OpenShift.