O'REILLY®

Compliments of salesforce

FREE CHAPTERS

Kubernetes Cookbook

BUILDING CLOUD NATIVE APPLICATIONS

Sébastien Goasguen & Michael Hausenblas

Deploy, scale, and manage enterprise infrastructure

AND YOUR GOALS AT THE SAME TIME.

Learn more about our open opportunities at **SALESFORCE.COM/TECH**

salesforce

Praise for *Kubernetes Cookbook*

The best infrastructure is the infrastructure you don't see. The *Kubernetes Cookbook* helps move in that direction. This book provides concrete examples of how to get stuff done so you can get back to your real job. Along the way, you'll build a skill set that helps you take your Kubernetes game to the next level.

—Joe Beda
CTO and Founder of Heptio,
Kubernetes Founder

The *Kubernetes Cookbook* is a fantastic hands-on guide to building and running applications on Kubernetes. It is a great reference and a great way to help you learn to build cloud-native containerized applications.

—Clayton Coleman
Red Hat

In this Cookbook, Sebastien and Michael collect together a number of useful recipes to get you going with Kubernetes, fast. They surface dozens of helpful hints and tips that will get readers hands-on with practical aspects of installing Kubernetes and using it to run applications.

—Liz Rice
Chief Technology Evangelist,
Aqua Security

Kubernetes Cookbook

Building Cloud-Native Applications

This excerpt contains Chapters 10 and 12 of *Kubernetes Cookbook*. The final book is available on Safari and through other retailers.

Sébastien Goasguen and Michael Hausenblas

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kubernetes Cookbook
by Sébastien Goasguen and Michael Hausenblas

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*.

Editor: Angela Rufino
Production Editor: Colleen Cole
Copyeditor: Rachel Head
Proofreader: Gillian McGarvey
Indexer: Ellen Troutman
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Rebecca Demarest

March 2018: First Edition

Revision History for the First Edition
2018-02-14: First Release

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

This work is part of a collaboration between O'Reilly and Salesforce. See our *statement of editorial independence*.

978-1-491-97968-6

[LSI]

For my boys, whose smiles, hugs, and spirits make me a better person.
For my wife with whom I am taking this journey through life.

Sébastien

For Saphira, Ranya, Iannis, and Anneliese.

Michael

Table of Contents

CHAPTER 1
Security

Running applications in Kubernetes comes with a shared responsibility between developers and ops folks to ensure that attack vectors are minimized, least-privileges principles are followed, and access to resources is clearly defined. In this chapter, we will present recipes that you can, and should, use to make sure your cluster and apps run securely. The recipes in this chapter cover:

- The role and usage of service accounts
- Role-Based Access Control (RBAC)
- Defining a pod's security context

1.1 Providing a Unique Identity for an Application

Problem

You want to provide an application with a unique identity in order to control access to resources on a fine-grained level.

Solution

Create a service account and use it in a pod specification.

To begin, create a new service account called `myappsa` and have a closer look at it:

```
$ kubectl create serviceaccount myappsa
serviceaccount "myappsa" created

$ kubectl describe sa myappsa
Name:           myappsa
Namespace:      default
```

```
Labels:         <none>
Annotations:    <none>

Image pull secrets:     <none>

Mountable secrets:      myappsa-token-rr6jc

Tokens:                 myappsa-token-rr6jc

$ kubectl describe secret myappsa-token-rr6jc
Name:           myappsa-token-rr6jc
Namespace:      default
Labels:         <none>
Annotations:    kubernetes.io/service-account.name=myappsa
                kubernetes.io/service-account.uid=0baa3df5-c474-11e7-8f08...

Type:   kubernetes.io/service-account-token

Data
====
ca.crt:         1066 bytes
namespace:      7 bytes
token:          eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ...
```

You can use this service account in a pod, like so:

```
kind:                 Pod
apiVersion:           v1
metadata:
  name:               myapp
spec:
  serviceAccountName: myappsa
  containers:
  - name:             main
    image:            centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

You can then verify whether the service account `myappsa` has been properly used by your pod by running:

```
$ kubectl exec myapp -c main \
          cat  /var/run/secrets/kubernetes.io/serviceaccount/token \
          eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9 ...
```

Indeed, the `myappsa` service account token has been mounted in the expected place in the pod and can be used going forward.

While a service account on its own is not super useful, it forms the basis for fine-grained access control; see Recipe 1.2 for more on this.

Discussion

Being able to identify an entity is the prerequisite for authentication and authorization. From the API server's point of view, there are two sorts of entities: human users and applications. While user identity (management) is outside of the scope of Kubernetes, there is a first-class resource representing the identity of an app: the service account.

Technically, the authentication of an app is captured by the token available in a file at the location `/var/run/secrets/kubernetes.io/serviceaccount/token`, which is mounted automatically through a secret. The service accounts are namespaced resources and are represented as follows:

```
system:serviceaccount:$NAMESPACE:$SERVICEACCOUNT
```

Listing the service accounts in a certain namespace gives you something like the following:

```
$ kubectl get sa
NAME         SECRETS   AGE
default      1         90d
myappsa      1         19m
prometheus   1         89d
```

Notice the service account called `default` here. This is created automatically; if you don't set the service account for a pod explicitly, as was done in the Solution, it will be assigned the `default` service account in its namespace.

See Also

- Managing Service Accounts
- Configure Service Accounts for Pods
- Pull an Image from a Private Registry

1.2 Listing and Viewing Access Control Information

Problem

You want to learn what actions you're allowed to do—for example, updating a deployment or listing secrets.

Solution

The following solution assumes you're using Role-Based Access Control as the authorization method.

To check if a certain action on a resource is allowed for a specific user, use `kubectl auth can-i`. For example, you can execute this command to check if the service account `system:serviceaccount:sec:myappsa` is allowed to list pods in the namespace `sec`:

```
$ kubectl auth can-i list pods --as=system:serviceaccount:sec:myappsa -n=sec
yes
```

If you want to try out this recipe in Minikube, you'll need to add `--extra-config=apiserver.Authorization.Mode=RBAC` when executing the binary.

To list the roles available in a namespace, do this:

```
$ kubectl get roles -n=kube-system
NAME                                             AGE
extension-apiserver-authentication-reader        1d
system::leader-locking-kube-controller-manager   1d
system::leader-locking-kube-scheduler            1d
system:controller:bootstrap-signer               1d
system:controller:cloud-provider                 1d
system:controller:token-cleaner                  1d

$ kubectl get clusterroles -n=kube-system
NAME                                             AGE
admin                                            1d
cluster-admin                                    1d
edit                                             1d
system:auth-delegator                            1d
system:basic-user                                1d
system:controller:attachdetach-controller        1d
system:controller:certificate-controller         1d
system:controller:cronjob-controller             1d
system:controller:daemon-set-controller          1d
system:controller:deployment-controller          1d
system:controller:disruption-controller          1d
system:controller:endpoint-controller            1d
system:controller:generic-garbage-collector      1d
system:controller:horizontal-pod-autoscaler      1d
system:controller:job-controller                 1d
system:controller:namespace-controller           1d
system:controller:node-controller                1d
system:controller:persistent-volume-binder       1d
system:controller:pod-garbage-collector          1d
system:controller:replicaset-controller          1d
system:controller:replication-controller         1d
system:controller:resourcequota-controller       1d
system:controller:route-controller               1d
system:controller:service-account-controller     1d
```

```
system:controller:service-controller            1d
system:controller:statefulset-controller        1d
system:controller:ttl-controller                1d
system:discovery                                1d
system:heapster                                 1d
system:kube-aggregator                          1d
system:kube-controller-manager                  1d
system:kube-dns                                 1d
system:kube-scheduler                           1d
system:node                                     1d
system:node-bootstrapper                        1d
system:node-problem-detector                    1d
system:node-proxier                             1d
system:persistent-volume-provisioner            1d
view                                            1d
```

The output shows the predefined roles, which you can use directly for users and service accounts.

To further explore a certain role and understand what actions are allowed, use:

```
$ kubectl describe clusterroles/view -n=kube-system
Name:           view
Labels:         kubernetes.io/bootstrapping=rbac-defaults
Annotations:    rbac.authorization.kubernetes.io/autoupdate=true
PolicyRule:
  Resources                                  Non-Resource URLs     ...  ...
  ---------                                  -----------------     ---  ---
  bindings                                   []                    ...  ...
  configmaps                                 []                    ...  ...
  cronjobs.batch                             []                    ...  ...
  daemonsets.extensions                      []                    ...  ...
  deployments.apps                           []                    ...  ...
  deployments.extensions                     []                    ...  ...
  deployments.apps/scale                     []                    ...  ...
  deployments.extensions/scale               []                    ...  ...
  endpoints                                  []                    ...  ...
  events                                     []                    ...  ...
  horizontalpodautoscalers.autoscaling       []                    ...  ...
  ingresses.extensions                       []                    ...  ...
  jobs.batch                                 []                    ...  ...
  limitranges                                []                    ...  ...
  namespaces                                 []                    ...  ...
  namespaces/status                          []                    ...  ...
  persistentvolumeclaims                     []                    ...  ...
  pods                                       []                    ...  ...
  pods/log                                   []                    ...  ...
  pods/status                                []                    ...  ...
  replicasets.extensions                     []                    ...  ...
  replicasets.extensions/scale               []                    ...  ...
  replicationcontrollers                     []                    ...  ...
  replicationcontrollers/scale               []                    ...  ...
  replicationcontrollers.extensions/scale    []                    ...  ...
```

```
replicationcontrollers/status                []                ...  ...
resourcequotas                               []                ...  ...
resourcequotas/status                        []                ...  ...
scheduledjobs.batch                          []                ...  ...
serviceaccounts                              []                ...  ...
services                                     []                ...  ...
statefulsets.apps                            []                ...  ...
```

In addition to the default roles defined in the `kube-system` namespace, you can define your own; see Recipe 1.3.

When RBAC is enabled, in many environments (including Minikube and GKE) you might see a Forbidden (403) status code and an error message as shown below when you try to access the Kubernetes dashboard:

> User "system:serviceaccount:kube-system:default" cannot list pods in the namespace "sec". (get pods)

To access the dashboard, you'll need to give the `kube-system:default` service account the necessary rights:

```
$ kubectl create clusterrolebinding admin4kubesystem \
  --clusterrole=cluster-admin \
  --serviceaccount=kube-system:default
```

Note that this command gives the service account a lot of rights and might not be advisable in a production environment.

Discussion

As you can see in Figure 1-1, there are a couple of moving parts when dealing with RBAC authorization:

- An entity—that is, a group, user, or service account
- A resource, such as a pod, service, or secret
- A role, which defines rules for actions on a resource
- A role binding, which applies a role to an entity

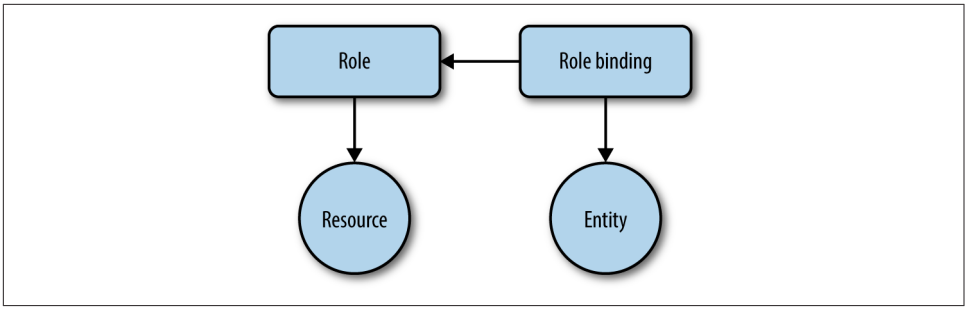

Figure 1-1. The RBAC concept

The actions on a resource that a role uses in its rules are the so-called verbs:

- `get`, `list`, `watch`
- `create`
- `update/patch`
- `delete`

Concerning the roles, we differentiate between two types:

- Cluster-wide: cluster roles and their respective cluster role bindings
- Namespace-wide: roles and role bindings

In Recipe 1.3, we will further discuss how you can create your own rules and apply them to users and resources.

See Also

- Kubernetes Authorization Overview
- Using RBAC Authorization

1.3 Controlling Access to Resources

Problem

For a given user or application, you want to allow or deny a certain action, such as viewing secrets or updating a deployment.

Solution

Let's assume you want to restrict an app to only be able to view pods—that is, list pods and get details about pods.

You'd start off with a pod definition in a YAML manifest, *pod-with-sa.yaml*, using a dedicated service account, `myappsa` (see Recipe 1.1):

```
kind:                Pod
apiVersion:          v1
metadata:
  name:              myapp
  namespace:         sec
spec:
  serviceAccountName: myappsa
  containers:
  - name:            main
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
```

Next, you'd define a role—let's call it `podreader` in the manifest *pod-reader.yaml*—that defines the allowed actions on resources:

```
kind:        Role
apiVersion:  rbac.authorization.k8s.io/v1beta1
metadata:
  name:      podreader
  namespace: sec
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs:     ["get", "list"]
```

Last but not least you need to apply the role `podreader` to the service account `myappsa`, using a role binding in *pod-reader-binding.yaml*:

```
kind:        RoleBinding
apiVersion:  rbac.authorization.k8s.io/v1beta1
metadata:
  name:      podreaderbinding
  namespace: sec
roleRef:
  apiGroup:  rbac.authorization.k8s.io
  kind:      Role
  name:      podreader
subjects:
- kind:      ServiceAccount
  name:      myappsa
  namespace: sec
```

When creating the respective resources, you can use the YAML manifests directly (assuming the service account has already been created):

```
$ kubectl create -f pod-reader.yaml
$ kubectl create -f pod-reader-binding.yaml
$ kubectl create -f pod-with-sa.yaml
```

Rather than creating manifests for the role and the role binding, you can use the following commands:

```
$ kubectl create role podreader \
          --verb=get --verb=list \
          --resource=pods -n=sec

$ kubectl create rolebinding podreaderbinding \
          --role=sec:podreader \
          --serviceaccount=sec:myappsa \
          --namespace=sec -n=sec
```

Note that this is a case of namespaced access control setup, since you're using roles and role bindings. For cluster-wide access control, you'd use the corresponding `create clusterrole` and `create clusterrolebinding` commands.

Sometimes it's not obvious if you should use a role or a cluster role and/or role binding, so here are a few rules of thumb you might find useful:

- If you want to restrict access to a namespaced resource (like a service or pod) in a certain namespace, use a role and a role binding (as we did in this recipe).
- If you want to reuse a role in a couple of namespaces, use a cluster role with a role binding.
- If you want to restrict access to cluster-wide resources such as nodes or to namespaced resources across all namespaces, use a cluster role with a cluster role binding.

See Also

- Configure RBAC in Your Kubernetes Cluster
- Antoine Cotten's blog post "Kubernetes v1.7 Security in Practice"

1.4 Securing Pods

Problem

You want to define the security context for an app on the pod level. For example, you want to run the app as a nonprivileged process or restrict the types of volumes the app can access.

Solution

To enforce policies on the pod level in Kubernetes, use the `securityContext` field in a pod specification.

Let's assume you want an app running as a nonroot user. For this, you'd use the security context on the container level as shown in the following manifest, *securedpod.yaml*:

```
kind:                Pod
apiVersion:          v1
metadata:
  name:              secpod
spec:
  containers:
  - name:            shell
    image:           centos:7
    command:
      - "bin/bash"
      - "-c"
      - "sleep 10000"
    securityContext:
      runAsUser:     5000
```

Now create the pod and check the user under which the container runs:

```
$ kubectl create -f securedpod.yaml
pod "secpod" created

$ kubectl exec secpod ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
5000         1  0.0  0.0   4328   672 ?        Ss   12:39   0:00 sleep 10000
5000         8  0.0  0.1  47460  3108 ?        Rs   12:40   0:00 ps aux
```

As expected, it's running as the user with ID `5000`. Note that you can also use the `securityContext` field on the pod level rather than on specific containers.

A more powerful method to enforce policies on the pod level is to use pod security policies (PSP). These are cluster-wide resources that allows you to define a range of policies, including some similar to what you've seen here but also restrictions around storage and networking. For a walk-through on how to use PSPs, see "Secure a Kubernetes Cluster with Pod Security Policies" in the Bitnami docs for Kubernetes.

See Also

- Pod Security Policies documentation
- Configure a Security Context for a Pod or Container

CHAPTER 2

Maintenance and Troubleshooting

In this chapter, you will find recipes that deal with both app-level and cluster-level maintenance. We cover various aspects of troubleshooting, from debugging pods and containers, to testing service connectivity, interpreting a resource's status, and node maintenance. Last but not least, we look at how to deal with `etcd`, the Kubernetes control plane storage component. This chapter is relevant for both cluster admins and app developers.

2.1 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`.

For Linux and the *bash* shell, you can enable `kubectl` autocompletion in your current shell using the following command:

```
$ source <(kubectl completion bash)
```

For other operating systems and shells, please check the documentation.

See Also

- Overview of kubectl
- kubectl Cheat Sheet

2.2 Removing a Pod from a Service

Problem

You have a well-defined service (see ???) backed by several pods. But one of the pods is misbehaving, and you would like to take it out of the list of endpoints to examine it at a later time.

Solution

Relabel the pod using the `--overwrite` option—this will allow you to change the value of the `run` label on the pod. By overwriting this label, you can ensure that it will not be selected by the service selector (???) and will be removed from the list of endpoints. At the same time, the replica set watching over your pods will see that a pod has disappeared and will start a new replica.

To see this in action, start with a straightforward deployment generated with `kubectl run` (see ???):

```
$ kubectl run nginx --image nginx --replicas 4
```

When you list the pods and show the label with key `run`, you'll see four pods with the value `nginx` (`run=nginx` is the label that is automatically generated by the `kubectl run` command):

```
$ kubectl get pods -Lrun
NAME                     READY     STATUS            RESTARTS   AGE       RUN
nginx-d5dc44cf7-5g45r    1/1       Running           0          1h        nginx
nginx-d5dc44cf7-l429b    1/1       Running           0          1h        nginx
nginx-d5dc44cf7-pvrfh    1/1       Running           0          1h        nginx
nginx-d5dc44cf7-vm764    1/1       Running           0          1h        nginx
```

You can then expose this deployment with a service and check the endpoints, which correspond to the IP addresses of each pod:

```
$ kubectl expose deployments nginx --port 80

$ kubectl get endpoints
NAME         ENDPOINTS                                                  AGE
nginx        172.17.0.11:80,172.17.0.14:80,172.17.0.3:80 + 1 more...    1h
```

Moving the first pod out of the service traffic via relabeling is done with a single command:

```
$ kubectl label pods nginx-d5dc44cf7-5g45r run=notworking --overwrite
```

To find the IP address of a pod, you can list the pod's manifest in JSON and run a JQuery query:

```
$ kubectl get pods nginx-d5dc44cf7-5g45r -o json | \
  jq -r .status.podIP172.17.0.3
```

You will see a brand new pod appear with the label `run=nginx`, and you will see that your nonworking pod still exists but no longer appears in the list of service endpoints:

```
$ kubectl get pods -Lrun
NAME                     READY     STATUS        RESTARTS   AGE       RUN
nginx-d5dc44cf7-5g45r    1/1       Running       0          21h       notworking
nginx-d5dc44cf7-hztlw    1/1       Running       0          21s       nginx
nginx-d5dc44cf7-l429b    1/1       Running       0          5m        nginx
nginx-d5dc44cf7-pvrfh    1/1       Running       0          5m        nginx
nginx-d5dc44cf7-vm764    1/1       Running       0          5m        nginx

$ kubectl describe endpoints nginx
Name:         nginx
Namespace:    default
Labels:       run=nginx
Annotations:  <none>
Subsets:
  Addresses:          172.17.0.11,172.17.0.14,172.17.0.19,172.17.0.7
...
```

2.3 Accessing a ClusterIP Service Outside the Cluster

Problem

You have an internal service that is causing you trouble and you want to test that it is working well locally without exposing the service externally.

Solution

Use a local proxy to the Kubernetes API server with `kubectl proxy`.

Let's assume that you have created a deployment and a service as described in Recipe 2.2. You should see an `nginx` service when you list the services:

```
$ kubectl get svc
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
nginx        ClusterIP   10.109.24.56   <none>        80/TCP    22h
```

This service is not reachable outside the Kubernetes cluster. However, you can run a proxy in a separate terminal and then reach it on `localhost`.

Start by running the proxy in a separate terminal:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

You can specify the port that you want the proxy to run on with the `--port` option.

In your original terminal, you can then use your browser or `curl` to access the application exposed by your service. Note the specific path to the service; it contains a `/proxy` part. Without this, you get the JSON object representing the service:

```
$ curl http://localhost:8001/api/v1/proxy/namespaces/default/services/nginx/
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

Note that you can now also access the entire Kubernetes API over `localhost` using `curl`.

2.4 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 two methods described here.

If you have the JSON query utility `jq` installed, you can use it to parse the resource status. Let's assume you have a pod called `jump` and want to know what Quality of Service (QoS) class[1] 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 subfield.

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 this technique to any resource. For example, you can 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",
```

1 Medium, "What are Quality of Service (QoS) Classes in Kubernetes".

```
          "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 you'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.

The Go programming language defines templates in a package called `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 this:

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

See Also

- jq Manual
- jq playground to try out queries without installing `jq`
- Package Template in the Go docs

2.5 Debugging Pods

Problem

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

Solution

To systematically discover and fix the cause of the problem, enter an OODA loop:

1. *Observe*. What do you see in the container logs? What events have occurred? How is the network connectivity?
2. *Orient*. Formulate a set of plausible hypotheses—stay as open-minded as possible and don't jump to conclusions.
3. *Decide*. Pick one of the hypotheses.
4. *Act*. Test the hypothesis. If it's confirmed, you're done; otherwise, go back to step 1 and continue.

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 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","apiVersion":
"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   ...   Reason                  Message
  ---------   ...   ------                  -------
  25s         ...   Scheduled               Successfully assigned
                                            unhappy-3626010456-4j251 to minikube
  25s         ...   SuccessfulMountVolume   MountVolume.SetUp succeeded for
                                            volume "default-token-rlm2s"
  24s         ...   Pulling                 pulling image "busybox"
  22s         ...   Pulled                  Successfully pulled image "busybox"
  22s         ...   Created                 Created container
  22s         ...   Started                 Started container
  19s         ...   BackOff                 Back-off restarting failed container
  19s         ...   FailedSync              Error syncing pod
```

As you can see, 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 to view the deployment (Figure 2-1), as well as the supervised replica set and the pod (Figure 2-2).

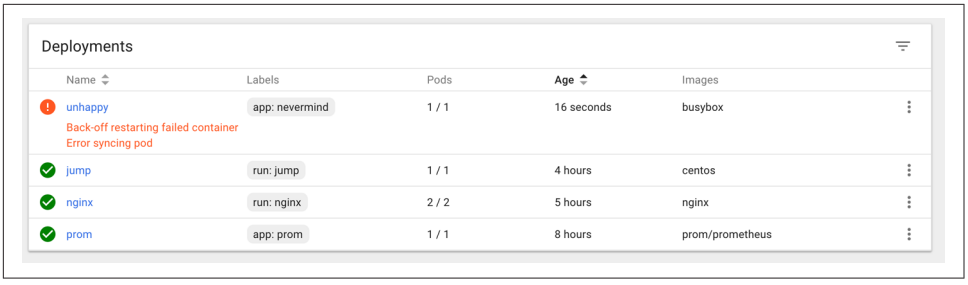

Figure 2-1. Screenshot of deployment in error state

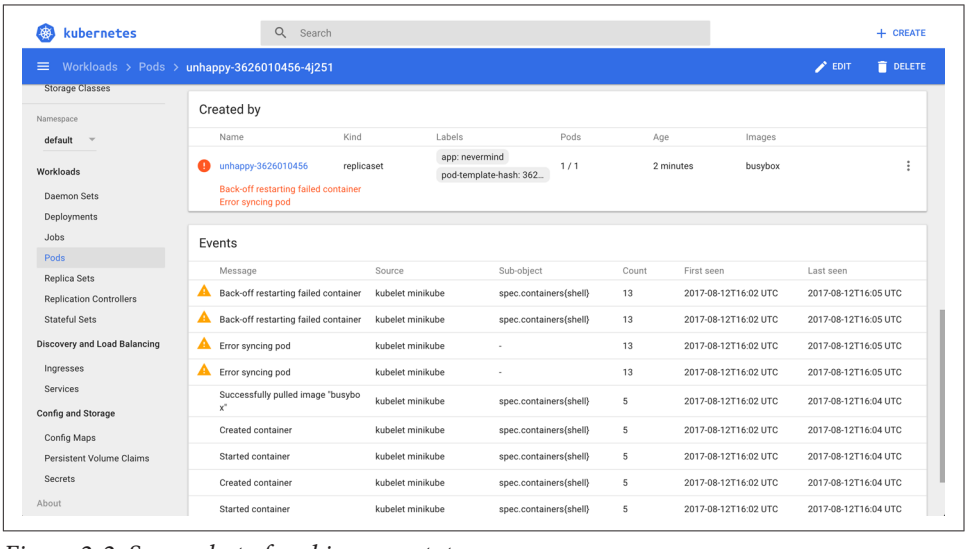

Figure 2-2. Screenshot of pod in error state

Discussion

An issue, be it a pod failing or a node behaving strangely, can have many different causes. Here are some things you'll 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 the nodes?

- Did you restrict the container's resource usage?

See Also

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

2.6 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` command. For example, to create a dump of the cluster state in a subdirectory *cluster-state-2017-08-13*, do this:

```
$ 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
```

2.7 Adding Kubernetes Worker Nodes

Problem

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

Solution

Provision a new machine in whatever way your environment requires (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:

`kubelet`
: This is 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 cannot 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 the following).

`Container runtime`
: This is responsible for downloading container images and running the containers. Initially, this was hardwired to the Docker engine, but nowadays it is a pluggable system based on the Container Runtime Interface (CRI), so you can, for example, use CRI-O rather than Docker.

`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 the installation method used (cloud, `kubeadm`, etc.). For a list of available options, see the `kubelet` reference and `kube-proxy` reference.

Discussion

Worker nodes, unlike other Kubernetes resources such as a deployments or services, 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

- "The Kubernetes Node" in the Kubernetes Architecture design document
- Master-Node communication
- Static Pods

2.8 Draining Kubernetes Nodes for Maintenance

Problem

You need to carry out maintenance on a node—for example, to 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, reproduced in Figure 2-3.

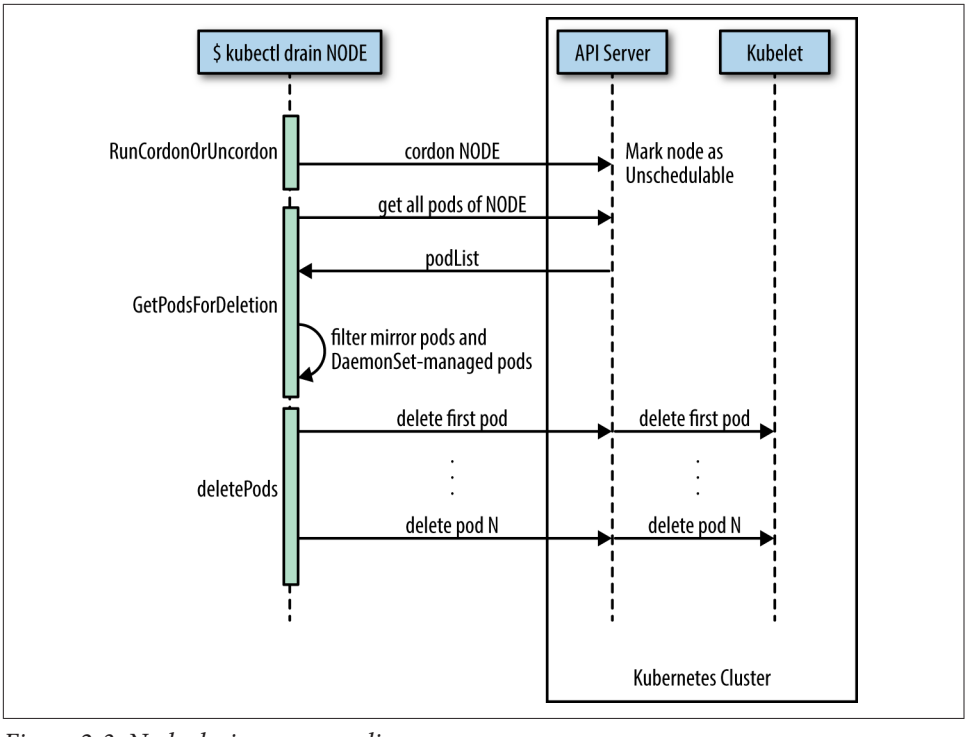

Figure 2-3. Node drain sequence diagram

The `kubectl drain` command 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—those pods would be immediately replaced by the `DaemonSet` controller, which ignores unschedulable markings.

`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 an RC, RS, job, `DaemonSet`, or `StatefulSet`.

See Also

- Safely Drain a Node while Respecting Application SLOs
- The `kubectl` reference docs

2.9 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 (with `jq` installed):

```
$ 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
      },
...
```

This technique can be used in environments where `etcd` is used with the v2 API.

Discussion

In Kubernetes, `etcd` is a component of the control plane. The API server (see ???) is stateless and 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; in `etcd2` the keys formed a hierarchy, but 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 we 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 with v2 being deprecated soon. Though 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 is used in which API mode.

In general, it's the responsibility of the cluster admin to manage `etcd`—that is, to upgrade it and make sure the data is backed up. In certain environments where the control plane is managed for you, such as in Google Kubernetes Engine, you cannot access `etcd` directly. This is by design, and there's no workaround for it.

See Also

- etcd v2 Cluster Administration guide
- etcd v3 Disaster Recovery guide
- Operating etcd clusters for Kubernetes
- "Accessing Localkube Resources from Inside a Pod: Example etcd" in the Minikube docs
- Stefan Schimanski and Michael Hausenblas's blog post "Kubernetes Deep Dive: API Server – Part 2"
- Michael Hausenblas's blog post "Notes on Moving from etcd2 to etcd3"

About the Authors

Sébastien Goasguen built his first compute cluster in the late '90s and takes pride in having completed his PhD thanks to Fortran 77 and partial differential equations. His struggles with parallel computers led him to work on making computing a utility and to 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 applications 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*, an avid blogger, and an online instructor of Kubernetes concepts for Safari subscribers.

Michael Hausenblas is a developer advocate for Go, Kubernetes, and OpenShift at Red Hat, where he helps AppOps to build and operate distributed services. His background is in large-scale data processing and container orchestration and he's experienced in advocacy and standardization at W3C and IETF. Before Red Hat, Michael worked at Mesosphere, MapR, and two research institutions in Ireland and Austria. He contributes to open source software (mainly using Go), blogs, and hangs out on Twitter too much.

Colophon

The animal on the cover of *Kubernetes Cookbook* is a Bengal eagle owl *(Bubo bengalensis)*. These large horned owls are usually seen in pairs and can be found in hilly and rocky scrub forests throughout South Asia.

The Bengal eagle owl measures 19–22 inches tall and weighs between 39–70 ounces. Its feathers are brownish-gray or beige and its ears have brown tufts. In contrast to the neutral color of its body, its eye color is strikingly orange. Owls with orange eyes hunt during the day. It prefers a meaty diet and mostly feasts on rodents such as mice or rats but will also resort to eating other birds during the winter. This owl produces a deep, resonant, booming, two-note "whooo" call that can be heard at dusk and dawn.

Females build nests in shallow recesses in the ground, rock ledges, and river banks, and lay 2–5 cream colored eggs. The eggs hatch after 33 days. By the time the chicks are around 10 weeks of age, they are adult-sized, though not mature yet, and they depend on their parents for nearly six months. To distract predators from their offspring, the parents will pretend to have a wing injury or fly in a zigzag manner.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Meyers Kleines Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.