O'REILLY®

Certified Kubernetes Application Developer (CKAD) Study Guide

In-Depth Guidance and Practice

Free Chapters

compliments of

Cockroach Labs

Benjamin Muschko

O'REILLY®

Certified Kubernetes Application Developer (CKAD) Study Guide

Developers with the ability to operate, troubleshoot, and monitor applications in Kubernetes are in high demand today. To meet this need, the Cloud Native Computing Foundation created a certification exam to establish a developer's credibility and value in the job market to work in a Kubernetes environment.

The Certified Kubernetes Application Developer (CKAD) exam is different from the typical multiple-choice format of other certifications. Instead, the CKAD is a performance-based exam that requires deep knowledge of the tasks under immense time pressure.

This study guide walks you through all the topics you need to fully prepare for the exam. Author Benjamin Muschko also shares his personal experience with preparing for all aspects of the exam.

- Learn when and how to apply Kubernetes concepts to manage an application
- Understand the objectives, abilities, tips, and tricks needed to pass the CKAD exam
- Explore the ins and outs of the kubectl command-line tool
- Demonstrate competency for performing the responsibilities of a Kubernetes application developer
- Solve real-world Kubernetes problems in a hands-on command-line environment
- Navigate and solve questions during the CKAD exam

Benjamin Muschko is a software engineer, consultant, and trainer with over 15 years of experience in the industry. He's passionate about project automation, testing, and continuous delivery. Ben is an author, a frequent speaker at conferences, and an avid open source advocate. He holds the CKAD certification.

"Ben wrangles the wide-gamut of Kubernetes skills and masterfully distills the essentials for that *little quiz*. You've got this."

—**Jonathan Johnson**
Software Architect, Kubernetes & Java Specialist, Dijure, LLC

"This book is all you need to clear the CKAD. And of course practice, lots of practice. Ben does an awesome job preparing you for this essential cloud native certification, providing you with detailed steps, tips, and tricks. It's a no-brainer to buy this book if you plan to do the CKAD."

—**Michael Hausenblas**
Developer Advocate, AWS

SOFTWARE DEVELOPMENT

US $49.99 CAN $65.99
ISBN: 978-1-492-08373-3

9 781492 083733 54999

CockroachDB

K8s + CockroachDB = Effortless App Deployment

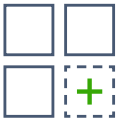
Run your application on the cloud-native database uniquely suited to Kubernetes.

Scale elastically with distributed SQL

Say goodbye to sharding and time-consuming manual scaling.

Survive anything with bullet-proof resilience

Rest easy knowing your application data is always on and always available.



Build fast with PostgreSQL compatibility

CockroachDB works with your current applications and fits how you work today.

Get started for free today

cockroachlabs.com/k8s

Trusted by innovators

COMCAST

SPACEX

BOSE

LUSH

rubrik

wework

Certified Kubernetes Application Developer (CKAD) Study Guide

In-Depth Guidance and Practice

This excerpt contains Chapters 7 and 8. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

Benjamin Muschko

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Certified Kubernetes Application Developer (CKAD) Study Guide
by Benjamin Muschko

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*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins
Development Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Holly Bauer Forsyth
Proofreader: Justin Billing

Indexer: Judy McConville
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

February 2021: First Edition

Revision History for the First Edition
2021-02-02: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492083733* for release details.

978-1-492-08373-3

[LSI]

Table of Contents

Foreword from Cockroach Labs

It's an undeniable fact that Kubernetes (K8s) has taken the app dev and delivery world by storm. The container orchestration project—bolstered by the rise of microservices—has fundamentally changed how we think about structuring our applications, managing our deployments, and how we use infrastructure. The level of freedom, flexibility, and control Kubernetes has brought to us is unparalleled with past innovations. Unless you work at Google, Spotify, or one of the other innovative early adopters of K8s, you (like many of us) are still working to understand and fully unlock the benefits of moving to Kubernetes.

Modern applications are overwhelmingly data-intensive. We have all come to expect highly dynamic, personalized experiences available instantaneously with up-to-the-millisecond data. There is zero tolerance for latency, downtime, or (the cardinal sin) incorrect data. These applications are increasingly global in nature, as well—think Netflix, Uber, or Square—and have complex needs in trying to balance both consistency and locality. How do you balance the desire for a single codebase and data set with the demand for ultra-fast user experiences and data sovereignty rules?

Thankfully, these are two areas that Kubernetes has helped make much, much easier. K8s has dramatically simplified our ability to deliver distributed, global or multi-region applications. And, when used in partnership with a distributed database or as a platform to directly manage stateful services like databases, it creates the ideal platform for this next generation of breakthrough applications.

It is for this reason that Cockroach Labs is proud to sponsor and make available two chapters from *Certified Kubernetes Application Developer (CKAD) Study Guide*: "Services & Networking" and "State Persistence". Cockroach Labs is the maker of CockroachDB, the leading cloud native, distributed SQL database for modern applications and the only database built for Kubernetes.

Our customers are at the forefront of innovation in distributed, data-intensive applications, and we're excited to help bring that perspective and insight to every developer who could benefit. We hope you enjoy this book excerpt, and that you consider CockroachDB for your next development project.

— Cockroach Labs, May 2021

CHAPTER 7

Services & Networking

In Chapter 2, we learned that you can communicate with a Pod by targeting its IP address. It's important to recognize that Pods' IP addresses are virtual and will therefore change to random values over time. A restart of a Pod will automatically assign a new virtual cluster IP address. Therefore, other parts of your system cannot rely on the Pod's IP address if they need to talk to one another.

The Kubernetes primitive Service implements an abstraction layer on top of Pods, assigning a fixed virtual IP fronting all the Pods with matching labels, and that virtual IP is called Cluster IP. This chapter will focus on the ins and outs of Services, and most importantly the exposure of Pods inside or outside of the cluster depending on their declared type.

By default, Kubernetes does not restrict inter-Pod communication in any shape or form. You can define a network policy to mitigate potential security risks. Network policies describe the access rules for incoming and outgoing network traffic to and from Pods. By the end of this chapter, you will have a basic understanding of its functionality based on common use cases.

At a high level, this chapter covers the following concepts:

- Service
- Deployment
- Network Policy

Understanding Services

Services are one of the central concepts in Kubernetes. Without a Service, you won't be able to expose your application to consumers in a stable and predictable fashion.

In a nutshell, Services provide discoverable names and load balancing to Pod replicas. The services and Pods remain agnostic from IPs with the help of the Kubernetes DNS control plane component. Similar to a Deployment, the Service determines the Pods it works on with the help of label selection.

Figure 7-1 illustrates the functionality. `Pod 1` and `Pod 2` receive traffic, as their assigned labels match with the label selection defined in the Service. `Pod 3` does not receive traffic, as it defines non-matching labels. Note that it is possible to create a Service without a label selector for less-common scenarios. Refer to the relevant Kubernetes documentation for more information.

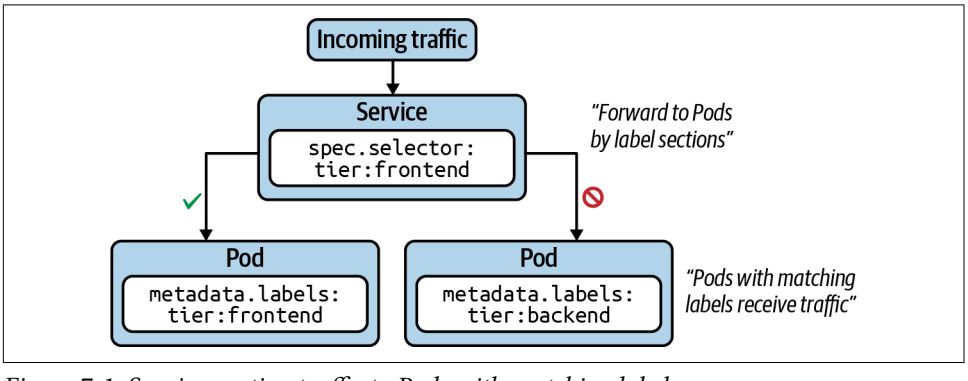


Figure 7-1. Service routing traffic to Pods with matching labels

Service Types

Every Service needs to define a type. The type determines how the Service exposes the matching Pods, as listed in Table 7-1.

Table 7-1. Service types

Type	Description
`ClusterIP`	Exposes the Service on a cluster-internal IP. Only reachable from within the cluster.
`NodePort`	Exposes the Service on each node's IP address at a static port. Accessible from outside of the cluster.
`LoadBalancer`	Exposes the Service externally using a cloud provider's load balancer.
`ExternalName`	Maps a Service to a DNS name.

The most important types you will need to understand for the CKAD exam are `ClusterIP` and `NodePort`. Those types make Pods reachable from within the cluster and from outside of the cluster. Later in this chapter, we'll explore both types by example.

Creating Services

As usual, we'll look at creating a Service from both the imperative and declarative approach angles. In fact, there are two ways to create a Service imperatively.

The command `create service` instantiates a new Service. You have to provide the type of the Service as the third, mandatory command-line option. That's also the case for the default type, `ClusterIP`. In addition, you can optionally provide the port mapping, which we'll discuss a little later in this chapter:

```
$ kubectl create service clusterip nginx-service --tcp=80:80
service/nginx-service created
```

Instead of creating a Service as a standalone object, you can also *expose* a Pod or Deployment with a single command. The `run` command provides an optional `--expose` command-line option, which creates a new Pod and a corresponding Service with the correct label selection in place:

```
$ kubectl run nginx --image=nginx --restart=Never --port=80 --expose
service/nginx created
pod/nginx created
```

For an existing Deployment, you can expose the underlying Pods with a Service using the `expose deployment` command:

```
$ kubectl expose deployment my-deploy --port=80 --target-port=80
service/my-deploy exposed
```

The `expose` command and the `--expose` command-line option are welcome shortcuts as a means to creating a new Service during the CKAD exam with a fast turnaround time.

Using the declarative approach, you would define a Service manifest in YAML form as shown in Example 7-1. As you can see, the key of the label selector uses the value `app`. After creating the Service, you will likely have to change the label selection criteria to meet your needs, as the `create service` command does not offer a dedicated command-line option for it.

Example 7-1. A Service defined by a YAML manifest

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: ClusterIP
  selector:
    app: nginx-service
  ports:
```

```
  - port: 80
    targetPort: 80
```

Listing Services

You can observe the most important attributes of a Service when rendering the full list for a namespace. The following command shows the type, the cluster IP, an optional external IP, and the mapped ports:

```
$ kubectl get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
nginx-service   ClusterIP   10.109.125.232   <none>        80/TCP    82m
```

Rendering Service Details

The `describe` command helps with retrieving even more details about a Service. The label selector will be included in the description of the Service, represented by the attribute `Selector`. That's important information when troubleshooting a Service object:

```
$ kubectl describe service nginx-service
Name:              nginx-service
Namespace:         default
Labels:            app=nginx-service
Annotations:       <none>
Selector:          app=nginx-service
Type:              ClusterIP
IP:                10.109.125.232
Port:              80-80  80/TCP
TargetPort:        80/TCP
Endpoints:         <none>
Session Affinity:  None
Events:            <none>
```

Port Mapping

In "Creating Services" on page 3, we only briefly touched on the topic of port mapping. The correct port mapping determines if the incoming traffic actually reaches the application running inside of the Pods that match the label selection criteria of the Service. A Service always defines two different ports: the incoming port accepting traffic and the outgoing port, also called the target port. Their functionality is best illustrated by example.

Figure 7-2 shows a Service that accepts incoming traffic on port 3000. That's the port defined by the attribute `ports.port` in the manifest. Any incoming traffic is then routed toward the target port, represented by `ports.targetPort`. The target port is

the same port as defined by the container running inside of the label-selected Pod. In this case, that's port 80.

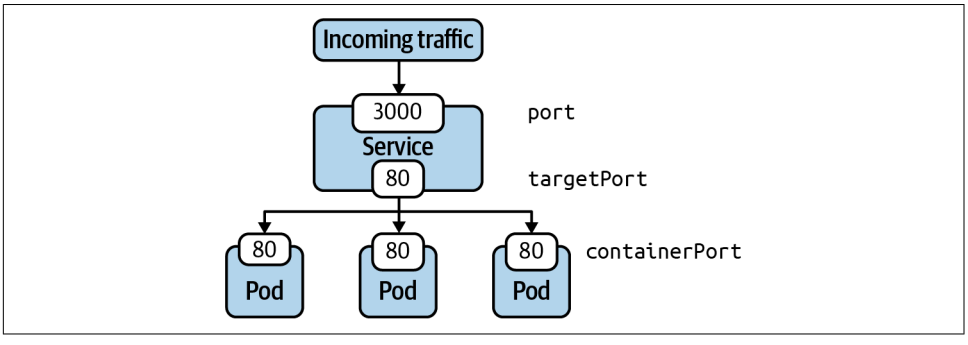


Figure 7-2. Port mapping of a Service to a Pod

Accessing a Service with Type ClusterIP

`ClusterIP` is the default type of Service. It exposes the Service on a cluster-internal IP address. Figure 7-3 shows how to reach a Pod exposed by the `ClusterIP` type from another Pod from within the cluster. You can also create a proxy from outside of the cluster using the `kubectl proxy` command. Using a proxy is not only meant for production environments but can also be helpful for troubleshooting a Service.

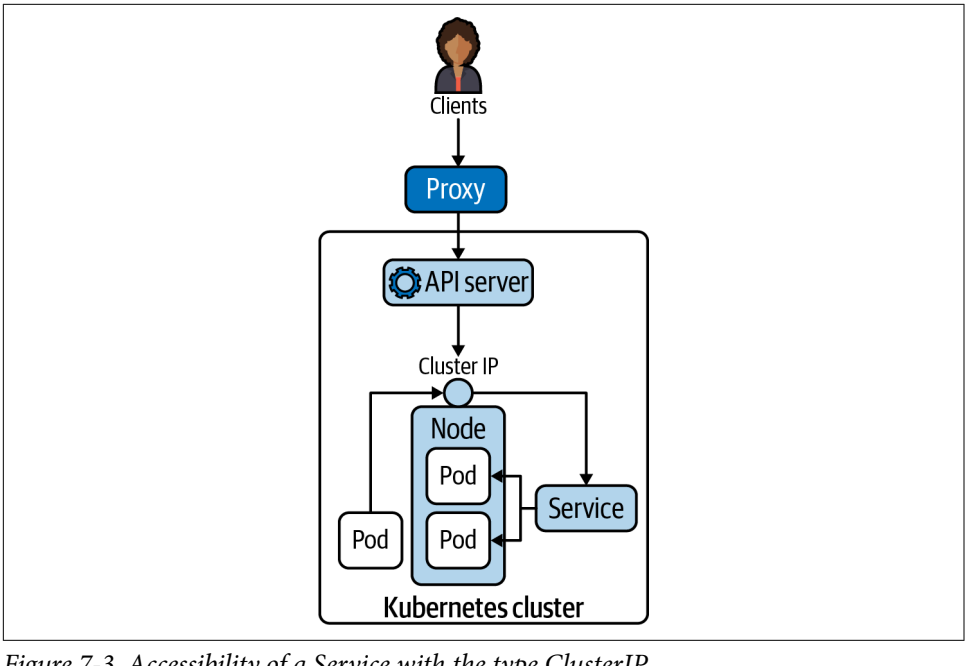


Figure 7-3. Accessibility of a Service with the type ClusterIP

To demonstrate the use case, we'll opt for a quick way to create the Pod and the corresponding Service with the same command. The command automatically takes care of properly mapping labels and ports:

```
$ kubectl run nginx --image=nginx --restart=Never --port=80 --expose
service/nginx created
pod/nginx created
$ kubectl get pod,service
NAME        READY   STATUS    RESTARTS   AGE
pod/nginx   1/1     Running   0          26s

NAME            TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
service/nginx   ClusterIP   10.96.225.204   <none>        80/TCP    26s
```

Remember that the Service of type `ClusterIP` can only be reached from within the cluster. To demonstrate the behavior, we'll create a new Pod running in the same cluster and execute a `wget` command to access the application. Have a look at the cluster IP exposed by the Service—that's `10.96.225.204`. The port is 80. Combined as a single command, you can resolve the application via `wget -O- 10.96.225.204:80` from the temporary Pod:

```
$ kubectl run busybox --image=busybox --restart=Never -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # wget -O- 10.96.225.204:80
Connecting to 10.96.225.204:80 (10.96.225.204:80)
writing to stdout
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and \
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

```
-                    100% |********************************************** \
**********************|   612  0:00:00 ETA
written to stdout
/ # exit
```

The `proxy` command can establish a direct connection to the Kubernetes API server from your localhost. With the following command, we are opening port 9999 on which to run the proxy:

```
$ kubectl proxy --port=9999
Starting to serve on 127.0.0.1:9999
```

After running the command, you will notice that the shell is going to wait until you break out of the operation. To try talking to the proxy, you will have to open another terminal window. Say you have the `curl` command-line tool installed on your machine to make a call to an endpoint of the API server. The following example uses `localhost:9999`—that's the proxy entry point. As part of the URL, you're providing the endpoint to the Service named `nginx` running in the `default` namespace according to the API reference:

```
$ curl -L localhost:9999/api/v1/namespaces/default/services/nginx/proxy
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and \
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Accessing a Service with Type NodePort

Declaring a Service with type `NodePort` exposes access through the node's IP address and can be resolved from outside of the Kubernetes cluster. The node's IP address can be reached in combination with a port number in the range of 30000 and 32767, assigned automatically upon the creation of the Service. Figure 7-4 illustrates the routing of traffic to Pods via a NodePort-typed Service.

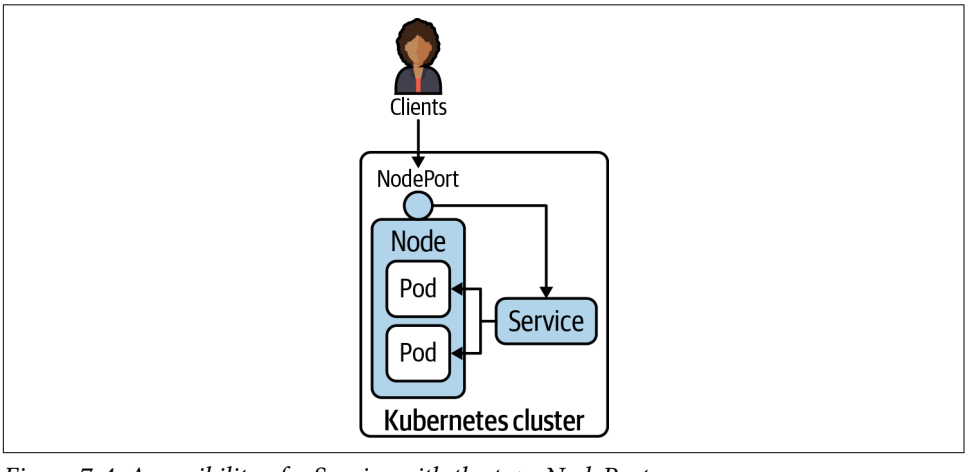


Figure 7-4. Accessibility of a Service with the type NodePort

Let's enhance the example from the previous section. We'll change the existing Service named `nginx` to use the type `NodePort` instead of `ClusterIP`. There are various ways to implement the change. For this example, we'll use the `patch` command. When listing the Service, you will find the changed type and the port you can use to reach the Service. The port that has been assigned in this example is 32300:

```
$ kubectl patch service nginx -p '{ "spec": {"type": "NodePort"} }'
service/nginx patched
$ kubectl get service nginx
NAME    TYPE       CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
nginx   NodePort   10.96.225.204   <none>        80:32300/TCP   3d21h
```

You should now be able to access the Service using the node IP address and the node port. One way to discover the IP address of the node is by first listing all available nodes and then inspecting the relevant ones for details. In the following commands, we are only running on a single-node Kubernetes cluster, which makes things easy:

```
$ kubectl get nodes
NAME       STATUS   ROLES    AGE   VERSION
minikube   Ready    master   91d   v1.18.3
$ kubectl describe node minikube | grep InternalIP:
  InternalIP:  192.168.64.2
$ curl 192.168.64.2:32300
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and \
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.co</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Deployments and Services

In this chapter, we discussed the primitive Service in detail. Many tutorials and examples on the internet explain the concept of a Service in conjunction with a Deployment. I'm sure it became abundantly clear that a Service does not need a Deployment to work *but* they can work in tandem, as shown in Figure 7-5. A Deployment manages Pods and their replication. A Service routes network requests to a set of Pods. Both primitives use label selection to connect with an associated set of Pods.

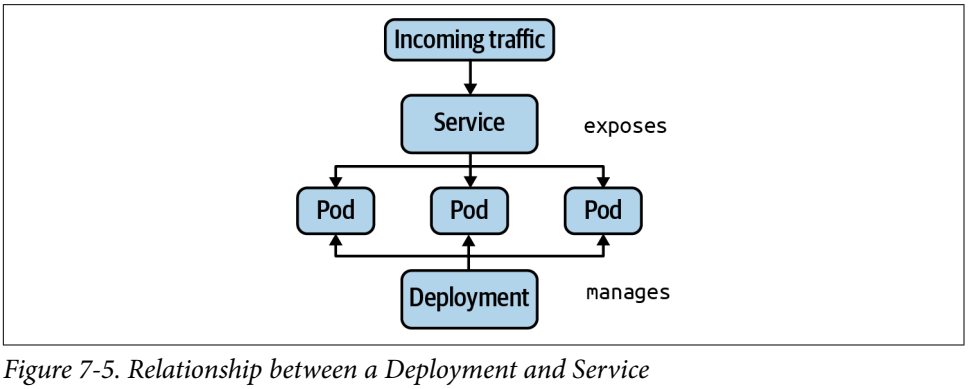


Figure 7-5. Relationship between a Deployment and Service

Understanding Network Policies

Within a Kubernetes cluster, any Pod can talk to any other Pod without restrictions using its IP address or DNS name, even across namespaces. Not only does unrestricted inter-Pod communication pose a potential security risk, it also makes it harder to understand the mental communication model of your architecture. For example, there's no good reason to allow a backend application running in a Pod to directly talk to the frontend application running in another Pod. The communication should be directed from the frontend Pod to the backend Pod. A network policy defines the rules that control traffic from and to a Pod, as illustrated in Figure 7-6.

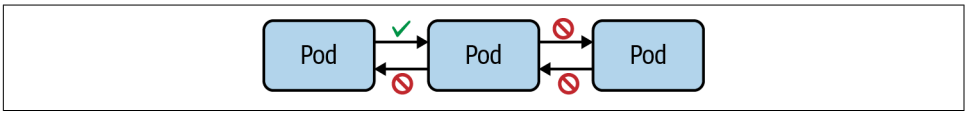


Figure 7-6. Network policies define traffic from and to a Pod

Label selection plays a crucial role in defining which Pods a network policy applies to. We already saw the concept in action in other contexts (e.g., the Deployment and the Service). Furthermore, a network policy defines the direction of the traffic, to allow or disallow. Incoming traffic is called *ingress*, and outgoing traffic is called *egress*. For ingress and egress, you can whitelist the sources of traffic like Pods, IP addresses, or ports.

A network policy defines a couple of important attributes, which together forms its set of rules. I want to discuss them first in Table 7-2, before looking at an exemplary scenario, so you have a rough idea what they mean in essence.

Table 7-2. Configuration elements of a network policy

Attribute	Description
`podSelector`	Selects the Pods in the namespace to apply the network policy to.
`policyTypes`	Defines the type of traffic (i.e., ingress and/or egress) the network policy applies to.
`ingress`	Lists the rules for incoming traffic. Each rule can define `from` and `ports` sections.
`egress`	Lists the rules for outgoing traffic. Each rule can define `to` and `ports` sections.

Creating Network Policies

The creation of network policies is best explained by example. Let's say you're dealing with the following scenario: you're running a Pod that exposes an API to other consumers. For example, it's a Pod that handles the processing of payments for other applications. The company you're working for is in the process of migrating applications from a legacy payment processor to a new one. Therefore, you'll only want to allow access from the applications that are capable of properly communicating with

it. Right now, you have two consumers—a grocery store and a coffee shop—each running their application in a separate Pod. The coffee shop is ready to consume the API of payment processor, but the grocery store isn't. Figure 7-7 shows the Pods and their assigned labels.

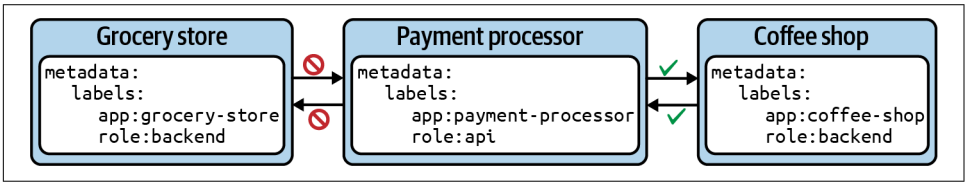


Figure 7-7. Limiting traffic to and from a Pod

You cannot create a new network policy with the imperative `create` command. Instead, you will have to use the declarative approach. The YAML manifest in Example 7-2, stored in the file *networkpolicy-api-allow.yaml*, shows a network policy for the scenario described previously.

Example 7-2. Declaring a NetworkPolicy with YAML

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: api-allow
spec:
  podSelector:
    matchLabels:
      app: payment-processor
      role: api
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: coffeeshop
```

Before creating the network policy, you'll stand up the Pod that runs the payment processor:

```
$ kubectl run payment-processor --image=nginx --restart=Never \
  -l app=payment-processor,role=api --port 80
pod/payment-processor created
$ kubectl get pods -o wide
NAME              READY STATUS  RESTARTS AGE   IP        NODE     NOMINATED NODE \
 READINESS GATES
payment-processor 1/1   Running 0        6m43s 10.0.0.51 minikube <none> \
 <none>
$ kubectl create -f networkpolicy-api-allow.yaml
networkpolicy.networking.k8s.io/api-allow created
```

Without a network policy controller, network policies won't have any effect. You need to configure a network overlay solution that provides this controller. If you're testing network policies on minikube, you'll have to go through some extra steps to install and enable the network provider Cilium. Without adhering to the proper prerequisites, network policies won't have any effect. You can find guidance on a dedicated page in the Kubernetes documentation.

To verify the correct behavior of the network policy, you'll emulate the grocery store Pod and the coffeshop Pod. As you can see in the following console output, traffic from the grocery store Pod is blocked:

```
$ kubectl run grocery-store --rm -it --image=busybox \
  --restart=Never -l app=grocery-store,role=backend -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.51
Connecting to 10.0.0.51 (10.0.0.51:80)
wget: download timed out
/ # exit
pod "grocery-store" deleted
```

Accessing the payment processor from the coffeeshop Pod works perfectly, as the Pod selector matches the label `app=coffeeshop`:

```
$ kubectl run coffeeshop --rm -it --image=busybox \
  --restart=Never -l app=coffeeshop,role=backend -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.51
Connecting to 10.0.0.51 (10.0.0.51:80)
remote file exists
/ # exit
pod "coffeshop" deleted
```

Listing Network Policies

Listing network policies works the same as any other Kubernetes primitive. Use the `get` command in combination with the resource type `networkpolicy`, or its short-form, `netpol`. For the previous network policy, you see a table that renders the name and Pod selector:

```
$ kubectl get networkpolicy
NAME         POD-SELECTOR                      AGE
api-allow    app=payment-processor,role=api    83m
```

It's unfortunate that the output of the command doesn't give away a lot of information about the rules. To retrieve more information, you have to dig into the details.

Rendering Network Policy Details

You can inspect the details of a network policy using the `describe` command. The output renders all the important information: Pod selector, and ingress and egress rules:

```
$ kubectl describe networkpolicy api-allow
Name:         api-allow
Namespace:    default
Created on:   2020-09-26 18:02:57 -0600 MDT
Labels:       <none>
Annotations:  <none>
Spec:
  PodSelector:     app=payment-processor,role=api
  Allowing ingress traffic:
    To Port: <any> (traffic allowed to all ports)
    From:
      PodSelector: app=coffeeshop
  Not affecting egress traffic
  Policy Types: Ingress
```

The network policy details don't draw a clear picture of the Pods that have been selected based on its rules. It would be extremely useful to be presented with a visual representation. The product Weave Cloud can provide such a visualization to make troubleshooting network policies easier. Remember that you do not have access to this product during the CKAD exam.

Isolating All Pods in a Namespace

The safest approach to writing a new network policy is to define it in a way that disallows all ingress and egress traffic. With those constraints in place, you can define more detailed rules and loosen restrictions gradually. The Kubernetes documentation describes such a default policy as shown in Example 7-3.

Example 7-3. Disallowing all traffic with the default policy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  - Egress
```

The curly braces for `spec.podSelector` mean "apply to all Pods in the namespace." The attribute `spec.policyTypes` defines the types of traffic the rule should apply to.

We can easily verify the correct behavior. Say, we're dealing with a Pod serving up frontend logic and another Pod that provides the backend functionality. The backend functionality is a basic NGINX web server exposing its endpoint on port 80. First, we'll create the backend Pod and connect to it from the frontend Pod running the `busybox` image. We should have no problem connecting to the backend Pod:

```
$ kubectl run backend --image=nginx --restart=Never --port=80
pod/backend created
$ kubectl get pods backend -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP          NODE     \
  NOMINATED NODE   READINESS GATES
backend   1/1     Running   0          16s   10.0.0.61   minikube \
  <none>           <none>
$ kubectl run frontend --rm -it --image=busybox --restart=Never -- /bin/sh
/ # wget --spider --timeout=1 10.0.0.61
Connecting to 10.0.0.61 (10.0.0.61:80)
remote file exists
/ # exit
pod "frontend" deleted
```

Now, we'll go through the same procedure but with the "deny all" network policy put in place. Ingress access to the backend Pod will be blocked:

```
$ kubectl create -f networkpolicy-deny-all.yaml
networkpolicy.networking.k8s.io/default-deny-all created
$ kubectl run frontend --rm -it --image=busybox --restart=Never -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # wget --spider --timeout=1 10.0.0.61
Connecting to 10.0.0.61 (10.0.0.61:80)
wget: download timed out
/ # exit
pod "frontend" deleted
```

Restricting Access to Ports

If not specified by a network policy, all ports are accessible. There are good reasons why you may want to restrict access on the port level as well. Say you're running an application in a Pod that only exposes port 8080 to the outside. While convenient during development, it widens the attack vector on any other port that's not relevant to the application. Port rules can be specified for ingress and egress as part of a network policy. The definition of a network policy in Example 7-4 allows access on port 8080.

Example 7-4. Definition of a network policy allowing ingress access on port 8080

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: port-allow
```

```
spec:
  podSelector:
    matchLabels:
      app: backend
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend
    ports:
    - protocol: TCP
      port: 8080
```

Summary

Pod-to-Pod communication should be through a Service. Services provide networking rules for a select set of Pods. Any traffic routed through the Service will be forwarded to Pods. You can assign one of the following types to a Service: `ClusterIP`, the default type; `NodePort`; `LoadBalancer`; or `ExternalName`. The selected type determines how the Pods are made available—for example, only from within the cluster or accessible from outside of the cluster. In practice, you'll commonly see a Deployment and a Service working together, though they serve different purposes and can operate independently.

Intra-Pod communication or communication between two containers of the same Pod is completely unrestricted in Kubernetes. Network policies instate rules to control the network traffic either from or to a Pod. You can think of network policies as firewall rules for Pods. It's best practice to start with a "deny all traffic" rule to minimize the attack vector. From there, you can open access as needed. Learning about the intricacies of network policies requires a bit of hands-on practice, as it is not directly apparent if the rules work as expected.

Exam Essentials

Understand the purpose of a Service
: The key takeaway for wanting to create a Service is that Pods expose an IP address but virtual and dyanmic IP address can't be relied upon. The IP address automatically changes whenever the Pod needs to be restarted—for example, as a result of a liveness probe identifying that the application doesn't work properly or a node drain/failure event. A Service creates a unified network interface and can expose a set of Pods associated with a label selector.

Practice the implications of different Service types
: Reading about the theoretical effect of assigning specific Service types won't be sufficient to prepare for the CKAD exam. You will need to practically experience

their impact by making Pods accessible from within or outside of the cluster. Spend extra time on the differences between `ClusterIP` and `NodePort`.

Know the basics about network policies
: The CKAD curriculum doesn't clearly state the depth of knowledge you need to have about network policies. I'd recommend going deeper than you would expect for the exam. Network policies come with a couple of basic rules. Once you understand those, it should be relatively easy to grasp their influence on accessibility. To explore common scenarios, have a look at the GitHub repository named "Kubernetes Network Policy Recipes". The repository comes with a visual representation for each scenario and walks you through the steps to set up the network policy and the involed Pods. This is a great practicing resource.

Sample Exercises

Solutions to these exercises are available in the Appendix.

1. Create a new Pod named `frontend` that uses the image `nginx`. Assign the labels `tier=frontend` and `app=nginx`. Expose the container port 80.
2. Create a new Pod named `backend` that uses the image `nginx`. Assign the labels `tier=backend` and `app=nginx`. Expose the container port 80.
3. Create a new Service named `nginx-service` of type `ClusterIP`. Assign the port 9000 and the target port 80. The label selector should use the criteria `tier=backend` and `deployment=app`.
4. Try to access the set of Pods through the Service from within the cluster. Which Pods does the Service select?
5. Fix the Service assignment to properly select the `backend` Pod and assign the correct target port.
6. Expose the Service to be accessible from outside of the cluster. Make a call to the Service.
7. Assume an application stack that defines three different layers: a frontend, a backend, and a database. Each of the layers runs in a Pod. You can find the definition in the YAML file *app-stack.yaml*:

```
kind: Pod
apiVersion: v1
metadata:
  name: frontend
  namespace: app-stack
  labels:
    app: todo
    tier: frontend
spec:
```

```
  containers:
  - name: frontend
    image: nginx

---

kind: Pod
apiVersion: v1
metadata:
  name: backend
  namespace: app-stack
  labels:
    app: todo
    tier: backend
spec:
  containers:
  - name: backend
    image: nginx

---

kind: Pod
apiVersion: v1
metadata:
  name: database
  namespace: app-stack
  labels:
    app: todo
    tier: database
spec:
  containers:
  - name: database
    image: mysql
    env:
    - name: MYSQL_ROOT_PASSWORD
      value: example
```

Create the namespace and the Pods using the file *app-stack.yaml*.

8. Create a network policy in the file *app-stack-network-policy.yaml*. The network policy should allow incoming traffic from the backend to the database but disallow incoming traffic from the frontend.
9. Reconfigure the network policy to only allow incoming traffic to the database on TCP port 3306 and no other port.

CHAPTER 8
State Persistence

Each container running in a Pod provides a temporary filesystem. Applications running in the container can read from it and write to it. A container's temporary filesystem is isolated from any other container or Pod and is not persisted beyond a Pod restart. The "State Persistence" section of the CKAD curriculum addresses the technical abstraction in Kubernetes responsible for persisting data beyond a container or Pod restart.

A Volume is a Kubernetes capability that persists data beyond a Pod restart. Essentially, a Volume is a directory that's shareable between multiple containers of a Pod. You will learn about the different Volume types and the process for defining and mounting a Volume in a container.

Persistent Volumes are a specific category of the wider concept of Volumes. The mechanics for Persistent Volumes are slightly more complex. The Persistent Volume is the resource that actually persists the data to an underlying physical storage. The Persistent Volume Claim represents the connecting resource between a Pod and a Persistent Volume responsible for requesting the storage. Finally, the Pod needs to *claim* the Persistent Volume and mount it to a directory path available to the containers running inside of the Pod.

At a high level, this chapter covers the following concepts:

- Volume
- Persistent Volume
- Persistent Volume Claim

Understanding Volumes

Applications running in a container can use the temporary filesystem to read and write files. In case of a container crash or a cluster/node restart, the kubelet will restart the container. Any data that had been written to the temporary filesystem is lost and cannot be retrieved anymore. The container effectively starts with a clean slate again.

There are many uses cases for wanting to mount a Volume in a container. We already saw one of the most prominent use cases in Chapter 4 that uses a Volume to exchange data between a main application container and a sidecar. Figure 8-1 illustrates the differences between the temporary filesystem of a container and the use of a Volume.

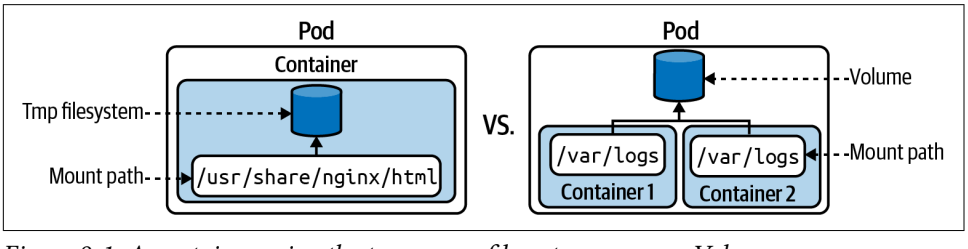


Figure 8-1. A container using the temporary filesystem versus a Volume

Volume Types

Every Volume needs to define a type. The type determines the medium that backs the Volume and its runtime behavior. The Kubernetes documentation offers a long list of Volume types. Some of the types—for example, `azureDisk`, `awsElasticBlockStore`, or `gcePersistentDisk`—are only available when running the Kubernetes cluster in a specific cloud provider. Table 8-1 shows a reduced list of Volume types that I deem to be most relevant to the CKAD exam.

Table 8-1. Volume types relevant to CKAD exam

Type	Description
`emptyDir`	Empty directory in Pod with read/write access. Only persisted for the lifespan of a Pod. A good choice for cache implementations or data exchange between containers of a Pod.
`hostPath`	File or directory from the host node's filesystem.
`configMap`, `secret`	Provides a way to inject configuration data. For practical examples, see Chapter 3.
`nfs`	An existing NFS (Network File System) share. Preserves data after Pod restart.
`persistentVolumeClaim`	Claims a Persistent Volume. Fore more information, see "Creating PersistentVolumeClaims" on page 24.

Creating and Accessing Volumes

Defining a Volume for a Pod requires two steps. First, you need to declare the Volume itself using the attribute `spec.volumes`. As part of the definition, you provide the name and the type. Just declaring the Volume won't be sufficient, though. Second, the Volume needs to be mounted to a path of the consuming container via `spec.containers.volumeMounts`. The mapping between the Volume and the Volume mount occurs by the matching name.

In Example 8-1, stored in the file *pod-with-volume.yaml* here, you can see the definition of a Volume with type `emptyDir`. The Volume has been mounted to the path */var/logs* inside of the container named `nginx`:

Example 8-1. A Pod defining and mounting a Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: business-app
spec:
  volumes:
  - name: logs-volume
    emptyDir: {}
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/logs
      name: logs-volume
```

Let's create the Pod and see if we can interact with the mounted Volume. The following commands open an interactive shell after the Pod's creation, then navigate to the mount path. You can see that the Volume type `emptyDir` initializes the mount path as an empty directory. New files and directories can be created as needed without limitations:

```
$ kubectl create -f pod-with-volume.yaml
pod/business-app created
$ kubectl get pod business-app
NAME           READY   STATUS    RESTARTS   AGE
business-app   1/1     Running   0          43s
$ kubectl exec business-app -it -- /bin/sh
# cd /var/logs
# pwd
/var/logs
# ls
# touch app-logs.txt
# ls
app-logs.txt
```

For an illustrative use case of the `emptyDir` Volume type mounted by more than one container, see Chapter 4.

Understanding Persistent Volumes

Data stored on Volumes outlive a container restart. In many applications, the data lives far beyond the lifecycles of the applications, container, Pod, nodes, and even the clusters themselves. Data persistence ensures the lifecycles of the data are decoupled from the lifecycles of the cluster resources. A typical example would be data persisted by a database. That's the responsibility of a Persistent Volume. Kubernetes models persist data with the help of two primitives: the PersistentVolume and the PersistentVolumeClaim.

The PersistentVolume is the storage device in a Kubernetes cluster. The PersistentVolume is completely decoupled from the Pod and therefore has its own lifecycle. The object captures the source of the storage (e.g., storage made available by a cloud provider). A PersistentVolume is either provided by a Kubernetes administrator or assigned dynamically by mapping to a storage class.

The PersistentVolumeClaim requests the resources of a PersistentVolume—for example, the size of the storage and the access type. In the Pod, you will use the type `persistentVolumeClaim` to mount the abstracted PersistentVolume by using the PersistentVolumeClaim.

Figure 8-2 shows the relationship between the Pod, the PersistentVolumeClaim, and the PersistentVolume.

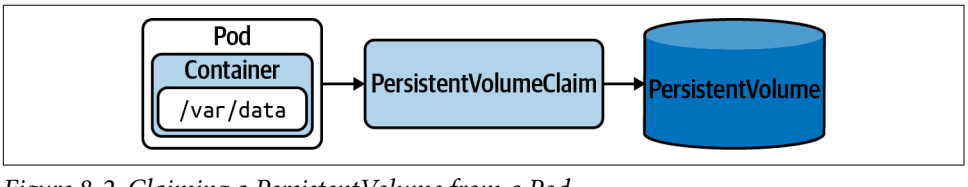


Figure 8-2. Claiming a PersistentVolume from a Pod

Static Versus Dynamic Provisioning

A PersistentVolume can be created statically or dynamically. If you go with the static approach, then you need to create storage device first and reference it by explicitly creating an object of kind PersistentVolume. The dynamic approach doesn't require you to create a PersistentVolume object. It will be automatically created from the PersistentVolumeClaim by setting a storage class name using the attribute `spec.storageClassName`.

A storage class is an abstraction concept that defines a class of storage device (e.g., storage with slow or fast performance) used for different application types. It's usually

the job of a Kubernetes administrator to set up storage classes. Minikube already creates a default storage class named `standard`, which you can query with the following command:

```
$ kubectl get storageclass
NAME                 PROVISIONER                RECLAIMPOLICY    VOLUMEBINDINGMODE \
  ALLOWVOLUMEEXPANSION   AGE
standard (default)   k8s.io/minikube-hostpath   Delete           Immediate \
  false                  108d
```

A deeper discussion on storage classes is out of scope for this book. For the CKAD exam, you will only need to understand the purpose of a storage class, how to set it, and its dynamic creation behavior of a PersistentVolume.

Creating PersistentVolumes

A PersistentVolume can only be created using the mainfest-first approach. At this time, `kubectl` does not allow the creation of a PersistentVolume using the `create` command. Every PersistentVolume needs to define the storage capacity using `spec.capacity` and an access mode set via `spec.accessModes`. Table 8-2 provides a high-level overview of the available access modes.

Table 8-2. PersistentVolume access modes

Type	Description
`ReadWriteOnce`	Read/write access by a single node.
`ReadOnlyMany`	Read-only access by many nodes.
`ReadWriteMany`	Read/write access by many nodes.

Example 8-2 creates a PersistentVolume named `db-pv` with a storage capacity of 1Gi and read/write access by a single node. The attribute `hostPath` mounts the directory */data/db* from the host node's filesystem. We'll store the YAML mainfest in the file *db-pv.yaml*.

Example 8-2. YAML manifest defining a PersistentVolume

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
```

```
  hostPath:
    path: /data/db
```

Upon inspection of the created PersistentVolume, you'll find most of the information you provided in the manifest. The status `Available` indicates that the object is ready to be claimed. The reclaim policy determines what should happen with the PersistentVolume after it has been released from its claim. By default, the object will be retained. The following example uses the short-form command `pv` to avoid having to type `persistentvolume`:

```
$ kubectl create -f db-pv.yaml
persistentvolume/db-pv created
$ kubectl get pv db-pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS      CLAIM   STORAGECLASS \
  REASON   AGE
db-pv  1Gi        RWO            Retain           Available \
           10s
```

Creating PersistentVolumeClaims

The next object we'll need to create is the PersistentVolumeClaim. Its purpose is to bind the PersistentVolume to the Pod. Let's have a look at the YAML manifest stored in the file *db-pvc.yaml*, as shown in Example 8-3.

Example 8-3. Definition of a PersistentVolumeClaim

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: db-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 512m
```

What we're saying here is, "Give me a PersistentVolume that can fulfill the resource request of 512m and provides the access mode `ReadWriteOnce`." The binding to an appropriate PersistentVolume happens automatically based on those criteria.

After creating the PersistentVolumeClaim, the status is set as `Bound`, which means that the binding to the PersistentVolume was successful. The following `get` command uses the short-form `pvc` instead of `persistentvolumeclaims`:

```
$ kubectl create -f db-pvc.yaml
persistentvolumeclaim/db-pvc created
$ kubectl get pvc db-pvc
```

```
NAME    STATUS  VOLUME                                     CAPACITY  ACCESS MODES \
  STORAGECLASS   AGE
db-pvc  Bound   pvc-c9e9aa9e-890d-4fd1-96e6-64072366f78d  512m      RWO \
  standard       7s
```

The PersistentVolume has not been mounted by a Pod yet. Therefore, inspecting the details of the object shows `<none>`. Using the `describe` command is a good way to verify if the PersistentVolumeClaim was mounted properly:

```
$ kubectl describe pvc db-pvc
...
Mounted By:    <none>
...
```

Mounting PersistentVolumeClaims in a Pod

All that's left is mounting the PersistentVolumeClaim in the Pod that wants to consume it. You already learned how to mount a Volume in a Pod. The big difference here, shown in Example 8-4, is using `spec.volumes.persistentVolumeClaim` and providing the name of the PersistentVolumeClaim.

Example 8-4. A Pod referencing a PersistentVolumeClaim

```
apiVersion: v1
kind: Pod
metadata:
  name: app-consuming-pvc
spec:
  volumes:
    - name: app-storage
      persistentVolumeClaim:
        claimName: db-pvc
  containers:
  - image: alpine
    name: app
    command: ["/bin/sh"]
    args: ["-c", "while true; do sleep 60; done;"]
    volumeMounts:
      - mountPath: "/mnt/data"
        name: app-storage
```

Let's assume we stored the configuration in the file *app-consuming-pvc.yaml*. After creating the Pod from the manifest, you should see the Pod transitioning into the `Ready` state. The `describe` command will provide additional information on the Volume:

```
$ kubectl create -f app-consuming-pvc.yaml
pod/app-consuming-pvc created
$ kubectl get pods
```

```
NAME                READY   STATUS    RESTARTS   AGE
app-consuming-pvc   1/1     Running   0          3s
$ kubectl describe pod app-consuming-pvc
...
Volumes:
  app-storage:
    Type:       PersistentVolumeClaim (a reference to a PersistentVolumeClaim \
                in the same namespace)
    ClaimName:  db-pvc
    ReadOnly:   false
...
```

The PersistentVolumeClaim now also shows the Pod that mounted it:

```
$ kubectl describe pvc db-pvc
...
Mounted By:    app-consuming-pvc
...
```

You can now go ahead and open an interactive shell to the Pod. Navigating to the mount path at */mnt/data* gives you access to the underlying PersistentVolume:

```
$ kubectl exec app-consuming-pvc -it -- /bin/sh
/ # cd /mnt/data
/mnt/data # ls -l
total 0
/mnt/data # touch test.db
/mnt/data # ls -l
total 0
-rw-r--r--    1 root     root             0 Sep 29 23:59 test.db
```

Summary

Containers store data in a temporary filesystem, which is empty each time a new Pod is started. Application developers need to persist data beyond the lifecycles of the containers, Pods, node, and cluster. Typical examples include persistent log files or data in a database.

Kubernetes offers the concept of a Volume to implement the use case. A Pod mounts a Volume to a path in the container. Any data written to the mounted storage will be persisted beyond a container restart. Kubernetes offers a wide range of Volume types to fulfill different requirements.

PersistentVolumes even store data beyond a Pod or cluster/node restart. Those objects are decoupled from the Pod's lifecycle and are therefore represented by a Kubernetes primitive. The PersistentVolumeClaim abstracts the underlying implementation details of a PersistentVolume and acts as an intermediary between Pod and PersistentVolume.

Exam Essentials

Understand the need and use cases for a Volume
: Many production-ready application stacks running in a cloud native environment need to persist data. Read up on common use cases and explore recipes that describe typical scenarios. You can find some examples in the O'Reilly books *Kubernetes Patterns*, *Kubernetes Best Practices*, and *Cloud Native DevOps with Kubernetes*.

Practice defining and consuming Volumes
: Volumes are a cross-cutting concept applied in different areas of the CKAD exam. Know where to find the relevant documentation for defining a Volume and the multitude of ways to consume a Volume from a container. Definitely revisit Chapter 3 for a deep dive on how to mount ConfigMaps and Secrets as a Volume, and Chapter 4 for coverage on sharing a Volume between two containers.

Internalize the mechanics of defining and consuming a PersistentVolume
: Creating a PersistentVolume involves a couple of moving parts. Understand the configuration options for PeristentVolumes and PersistentVolumeClaims and how they play together. Try to emulate situations that prevent a successful binding of a PersistentVolumeClaim. Then fix the situation by taking counteractions. Internalize the short-form commands `pv` and `pvc` to save precious time during the exam.

Sample Exercises

Solutions to these exercises are available in the Appendix.

1. Create a Pod YAML file with two containers that use the image `alpine:3.12.0`. Provide a command for both containers that keep them running forever.
2. Define a Volume of type `emptyDir` for the Pod. Container 1 should mount the Volume to path */etc/a*, and container 2 should mount the Volume to path */etc/b*.
3. Open an interactive shell for container 1 and create the directory *data* in the mount path. Navigate to the directory and create the file *hello.txt* with the contents "Hello World." Exit out of the container.
4. Open an interactive shell for container 2 and navigate to the directory */etc/b/data*. Inspect the contents of file *hello.txt*. Exit out of the container.
5. Create a PersistentVolume named `logs-pv` that maps to the `hostPath` */var/logs*. The access mode should be `ReadWriteOnce` and `ReadOnlyMany`. Provision a storage capacity of 5Gi. Ensure that the status of the PersistentVolume shows `Available`.

6. Create a PersistentVolumeClaim named `logs-pvc`. The access it uses is `ReadWriteOnce`. Request a capacity of 2Gi. Ensure that the status of the PersistentVolume shows `Bound`.
7. Mount the PersistentVolumeClaim in a Pod running the image `nginx` at the mount path */var/log/nginx*.
8. Open an interactive shell to the container and create a new file named *my-nginx.log* in */var/log/nginx*. Exit out of the Pod.
9. Delete the Pod and re-create it with the same YAML manifest. Open an interactive shell to the Pod, navigate to the directory */var/log/nginx*, and find the file you created before.

About the Author

Benjamin Muschko is a software engineer, consultant, and trainer with more than 15 years of experience in the industry. He's passionate about project automation, testing, and continuous delivery. Ben is an author, a frequent speaker at conferences, and an avid open source advocate. He holds the CKAD certification.

Software projects sometimes feel like climbing a mountain. In his free time, Ben loves hiking Colorado's 14ers and enjoys conquering long-distance trails.

Colophon

The animal on the cover of *Certified Kubernetes Application Developer (CKAD) Study Guide* is a common porpoise (*Phocoena phocoena*). It is the smallest of the seven species of porpoise and one of the smallest marine mammals. Adults are 4.5 to 6 feet long and weigh between 130 and 170 pounds. They are dark gray with lightly speckled sides and white undersides. Females are larger than males.

The common porpoise lives in the coastal waters of the North Atlantic, North Pacific, and Black Sea. They are also known as harbor porpoises since they inhabit fjords, bays, estuaries, and harbors. These marine mammals eat very small schooling fish and will hunt several hundred fish per hour throughout the day. They are usually solitary hunters but will occasionally form small packs.

Porpoises use ultrasonic clicks for echolocation (for both navigation and hunting) and social communication. A mass of adipose tissue in the skull, known as a melon, focuses and modulates their vocalizations.

Porpoises are conscious breathers, so if they are unconscious for a long time, they may drown. In captivity, they have been known to sleep with one side of their brain at a time so that they can still swim and breathe consciously.

The conservation status of the common porpoise is of least concern. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black and white engraving from *British Quadrupeds*. The cover fonts are Gilroy Semibold 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.