

EXPERT INSIGHT

Mastering Kubernetes

Level up your container orchestration skills with
Kubernetes to build, run, secure, and observe
large-scale distributed apps

Third Edition

Gigi Sayfan



Packt

Mastering Kubernetes

Third Edition

Level up your container orchestration skills with Kubernetes to build, run, secure, and observe large-scale distributed apps

Gigi Sayfan



BIRMINGHAM - MUMBAI

Mastering Kubernetes

Third Edition

Copyright © 2020 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Producers: Ben Renow-Clarke, Aarthi Kumaraswamy

Acquisition Editor – Peer Reviews: Suresh Jain

Content Development Editor: Kate Blackham

Technical Editor: Gaurav Gavas

Project Editor: Carol Lewis

Proofreader: Safis Editing

Indexer: Pratik Shirodkar

Presentation Designer: Sandip Tadge

First published: May 2017

Second edition: April 2018

Third edition: June 2020

Production reference: 1260620

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-83921-125-6

www.packt.com



packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Learn better with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.Packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customer@packtpub.com for more details.

At www.Packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Gigi Sayfan has been developing software professionally for more than 20 years in domains as diverse as instant messaging, morphing, chip fabrication process control, embedded multimedia applications for game consoles, brain-inspired machine learning, custom browser development, web services for 3D distributed game platforms, IoT sensors, virtual reality, and genomics. He has written production code in many programming languages, such as Go, Python, C, C++, C#, Java, Delphi, JavaScript, and even Cobol and PowerBuilder for operating systems such as Windows (3.11 through 7), Linux, macOS, Lynx (embedded), and Sony PlayStation. His technical expertise includes databases, low-level networking, distributed systems, unorthodox user interfaces, DevOps, and the general software development life cycle.

Gigi is also a longtime author who has published multiple books and hundreds of technical articles and blogs.

About the reviewer

Onur Yilmaz is a senior software engineer at a multinational enterprise software company. He is a **Certified Kubernetes Administrator (CKA)** and works on Kubernetes and cloud management systems. He is a keen supporter of cutting-edge technologies including Docker, Kubernetes, and cloud-native applications. He is the author of multiple books, including *Introduction to DevOps with Kubernetes*, *Kubernetes Design Patterns and Extensions*, *Serverless Architectures with Kubernetes*, and *Cloud-Native Continuous Integration and Delivery*. He has one master's and two bachelor's degrees in the engineering field.

Table of Contents

Preface	xix
Chapter 1: Understanding Kubernetes Architecture	1
What is Kubernetes?	2
What Kubernetes is not	2
Understanding container orchestration	3
Physical machines, virtual machines, and containers	3
The benefits of containers	3
Containers in the cloud	4
Cattle versus pets	5
Kubernetes concepts	5
Clusters	6
Nodes	6
The master	7
Pods	7
Labels	8
Annotations	8
Label selectors	9
Services	9
Volume	10
Replication controllers and replica sets	10
StatefulSet	10
Secrets	11
Names	11
Namespaces	11
Diving into Kubernetes architecture in depth	12
Distributed system design patterns	12
The sidecar pattern	13

The ambassador pattern	13
The adapter pattern	13
Multi-node patterns	14
The Kubernetes APIs	14
Resource categories	15
Kubernetes components	18
Master components	18
Node components	21
Kubernetes runtimes	22
The container runtime interface (CRI)	23
Docker	25
rkt	27
App container	27
CRI-O	27
Hyper containers	28
Frakti	28
Stackube	28
Continuous integration and deployment	28
What is a CI/CD pipeline?	29
Designing a CI/CD pipeline for Kubernetes	30
Summary	30
Chapter 2: Creating Kubernetes Clusters	31
Overview	31
Creating a single-node cluster with Minikube	32
Meet kubect!l	32
Quick introduction to Minikube	33
Getting ready	33
On Windows	33
On macOS	34
Creating the cluster	35
Troubleshooting	36
Checking out the cluster	37
Doing work	38
Examining the cluster with the dashboard	40
Creating a multi-node cluster with KinD	42
Quick introduction to KinD	42
Installing KinD	42
Creating the cluster with KinD	43
Doing work with KinD	46
Accessing Kubernetes services locally though a proxy	46
Creating a multi-node cluster with k3d	47
Quick introduction to k3s and k3d	48

Installing k3d	48
Creating the cluster with k3d	49
Comparing Minikube, KinD, and k3d	51
Creating clusters in the cloud (GCP, AWS, Azure)	52
The cloud-provider interface	52
GCP	53
AWS	53
Kubernetes on EC2	54
AWS EKS	55
Fargate	55
Azure	56
Other cloud providers	56
Once upon a time in China	57
IBM Kubernetes Service	57
Oracle Container Service	58
Creating a bare-metal cluster from scratch	58
Use cases for bare metal	58
When should you consider creating a bare-metal cluster?	59
Understanding the process	59
Using virtual private cloud infrastructure	60
Building your own cluster with Kubespray	60
Building your cluster with KRIB	60
Building your cluster with RKE	61
Bootkube	61
Summary	61
References	62
Chapter 3: High Availability and Reliability	63
High availability concepts	64
Redundancy	64
Hot swapping	64
Leader election	65
Smart load balancing	65
Idempotency	66
Self-healing	66
High availability best practices	66
Creating highly available clusters	67
Making your nodes reliable	68
Protecting your cluster state	69
Clustering etcd	69
Verifying the etcd cluster	73
Protecting your data	73
Running redundant API servers	74

Running leader election with Kubernetes	74
Making your staging environment highly available	75
Testing high availability	76
High availability, scalability, and capacity planning	77
Installing the cluster autoscaler	78
Considering the vertical pod autoscaler	80
Live cluster updates	80
Rolling updates	81
Complex deployments	83
Blue-green deployments	84
Canary deployments	85
Managing data-contract changes	86
Migrating data	86
Deprecating APIs	87
Large cluster performance, cost, and design trade-offs	88
Availability requirements	88
Best effort	88
Maintenance windows	89
Quick recovery	90
Zero downtime	90
Site reliability engineering	92
Performance and data consistency	93
Summary	93
References	94
Chapter 4: Securing Kubernetes	95
Understanding Kubernetes security challenges	96
Node challenges	96
Network challenges	97
Image challenges	99
Configuration and deployment challenges	100
Pod and container challenges	101
Organizational, cultural, and process challenges	102
Hardening Kubernetes	103
Understanding service accounts in Kubernetes	103
How does Kubernetes manage service accounts?	105
Accessing the API server	105
Authenticating users	106
Authorizing requests	108
Using admission control plugins	110
Securing pods	112
Using a private image repository	112
ImagePullSecrets	112

Specifying a security context	113
Protecting your cluster with AppArmor	114
Pod security policies	116
Authorizing pod security policies via RBAC	117
Managing network policies	118
Choosing a supported networking solution	119
Defining a network policy	119
Limiting egress to external networks	121
Cross-namespace policies	122
Using secrets	122
Storing secrets in Kubernetes	122
Configuring encryption at rest	122
Creating secrets	123
Decoding secrets	124
Using secrets in a container	124
Running a multi-user cluster	125
The case for a multi-user cluster	126
Using namespaces for safe multi-tenancy	126
Avoiding namespace pitfalls	127
Summary	128
References	128
Chapter 5: Using Kubernetes Resources in Practice	129
Designing the Hue platform	129
Defining the scope of Hue	130
Smart reminders and notifications	130
Security, identity, and privacy	130
Hue components	131
Hue microservices	133
Planning workflows	135
Automatic workflows	135
Human workflows	135
Budget-aware workflows	135
Using Kubernetes to build the Hue platform	136
Using kubectl effectively	136
Understanding kubectl resource configuration files	137
ApiVersion	138
Kind	138
Metadata	138
Spec	138
Deploying long-running microservices in pods	139
Creating pods	139
Decorating pods with labels	141
Deploying long-running processes with deployments	142
Updating a deployment	143
Separating internal and external services	144

Deploying an internal service	145
Creating the Hue-reminders service	146
Exposing a service externally	148
Ingress	149
Advanced scheduling	150
Node selector	150
Taints and tolerations	151
Node affinity and anti-affinity	153
Pod affinity and anti-affinity	153
Using namespaces to limit access	154
Using kustomization for hierarchical cluster structures	156
Understanding the basics of kustomize	156
Configuring the directory structure	157
Applying kustomizations	158
Patching	160
Kustomizing the entire staging namespace	160
Launching jobs	162
Running jobs in parallel	163
Cleaning up completed jobs	164
Scheduling cron jobs	164
Mixing non-cluster components	166
Outside-the-cluster-network components	166
Inside-the-cluster-network components	167
Managing the Hue platform with Kubernetes	167
Using liveness probes to ensure your containers are alive	167
Using readiness probes to manage dependencies	168
Employing init containers for orderly pod bring-up	169
Pod readiness and readiness gates	170
Sharing with DaemonSet pods	171
Evolving the Hue platform with Kubernetes	172
Utilizing Hue in an enterprise	172
Advancing science with Hue	173
Educating the kids of the future with Hue	173
Summary	173
References	174
Chapter 6: Managing Storage	175
Persistent volumes walkthrough	175
Volumes	176
Using emptyDir for intra-pod communication	176
Using HostPath for intra-node communication	178
Using local volumes for durable node storage	180
Provisioning persistent volumes	181

Provisioning persistent volumes externally	182
Creating persistent volumes	182
Capacity	183
Volume mode	183
Access modes	183
Reclaim policy	184
Storage class	184
Volume type	185
Mount options	185
Making persistent volume claims	185
Mounting claims as volumes	188
Raw block volumes	189
Storage classes	191
Default storage class	192
Demonstrating persistent volume storage end to end	192
Public cloud storage volume types – GCE, AWS, and Azure	198
Amazon EBS	198
Amazon EFS	199
GCE persistent disk	201
Azure data disk	202
Azure Files	203
GlusterFS and Ceph volumes in Kubernetes	204
Using GlusterFS	205
Creating endpoints	205
Adding a GlusterFS Kubernetes service	206
Creating pods	207
Using Ceph	208
Connecting to Ceph using RBD	208
Connecting to Ceph using CephFS	210
Flocker as a clustered container data volume manager	211
Integrating enterprise storage into Kubernetes	212
Rook – the new kid on the block	213
Projecting volumes	214
Using out-of-tree volume plugins with FlexVolume	215
The Container Storage Interface	216
Volume snapshotting and cloning	217
Volume snapshots	217
Volume cloning	218
Summary	219
Chapter 7: Running Stateful Applications with Kubernetes	221
Stateful versus stateless applications in Kubernetes	221
Understanding the nature of distributed data-intensive apps	222
Why manage state in Kubernetes?	222
Why manage state outside of Kubernetes?	222

Shared environment variables versus DNS records for discovery	223
Accessing external data stores via DNS	223
Accessing external data stores via environment variables	223
Consuming a ConfigMap as an environment variable	224
Using a redundant in-memory state	225
Using DaemonSet for redundant persistent storage	226
Applying persistent volume claims	226
Utilizing StatefulSets	226
Running a Cassandra cluster in Kubernetes	228
Quick introduction to Cassandra	229
The Cassandra Docker image	230
Hooking up Kubernetes and Cassandra	238
Creating a Cassandra headless service	241
Using StatefulSets to create the Cassandra cluster	241
Summary	246
Chapter 8: Deploying and Updating Applications	247
Horizontal pod autoscaling	248
Declaring an HPA	248
Custom metrics	251
Autoscaling with Kubectl	251
Performing rolling updates with autoscaling	254
Handling scarce resources with limits and quotas	257
Enabling resource quotas	258
Resource quota types	258
Compute resource quota	258
Storage resource quota	259
Object count quota	260
Quota scopes	261
Resource quotas and priority classes	261
Requests and limits	262
Working with quotas	262
Using namespace-specific context	262
Creating quotas	262
Using limit ranges for default compute quotas	267
Choosing and managing the cluster capacity	268
Choosing your node types	268
Choosing your storage solutions	269
Trading off cost and response time	269
Using multiple node configurations effectively	270
Benefiting from elastic cloud resources	270
Autoscaling instances	270
Mind your cloud quotas	271
Manage regions carefully	271
Considering container-native solutions	272

Pushing the envelope with Kubernetes	273
Improving the performance and scalability of Kubernetes	274
Caching reads in the API server	274
The pod lifecycle event generator	274
Serializing API objects with protocol buffers	276
etcd3	276
Other optimizations	277
Measuring the performance and scalability of Kubernetes	277
The Kubernetes SLOs	277
Measuring API responsiveness	277
Measuring end-to-end pod startup time	279
Testing Kubernetes at scale	280
Introducing the Kubemark tool	281
Setting up a Kubemark cluster	281
Comparing a Kubemark cluster to a real-world cluster	281
Summary	282
Chapter 9: Packaging Applications	283
Understanding Helm	283
The motivation for Helm	284
The Helm 2 architecture	284
Helm 2 components	284
The Tiller server	284
The Helm client	285
Helm 3	285
Using Helm	285
Installing Helm	286
Installing the Helm client	286
Installing the Tiller server for Helm 2	286
Finding charts	287
Adding repositories	288
Installing packages	290
Checking the installation status	292
Customizing a chart	296
Additional installation options	298
Upgrading and rolling back a release	298
Deleting a release	299
Working with repositories	300
Managing charts with Helm	301
Taking advantage of starter packs	302
Creating your own charts	302
The Chart.yaml file	303
Versioning charts	303
The appVersion field	303
Deprecating charts	304
Chart metadata files	304

Managing chart dependencies	304
Managing dependencies with requirements.yaml	305
Utilizing special fields in requirements.yaml	306
Using templates and values	307
Writing template files	307
Testing and troubleshooting your charts	309
Embedding built-in objects	311
Feeding values from a file	312
Scope, dependencies, and values	313
Summary	315
Chapter 10: Exploring Advanced Networking	317
Understanding the Kubernetes networking model	318
Intra-pod communication (container to container)	318
Inter-pod communication (pod to pod)	318
Pod-to-service communication	319
External access	319
Kubernetes networking versus Docker networking	320
Lookup and discovery	322
Self-registration	322
Services and endpoints	322
Loosely coupled connectivity with queues	323
Loosely coupled connectivity with data stores	323
Kubernetes ingress	324
Kubernetes network plugins	324
Basic Linux networking	324
IP addresses and ports	324
Network namespaces	325
Subnets, netmasks, and CIDRs	325
Virtual Ethernet devices	325
Bridges	325
Routing	325
Maximum transmission unit	326
Pod networking	326
Kubenet	326
Container networking interface	327
Kubernetes networking solutions	332
Bridging on bare metal clusters	332
Contiv	332
Open vSwitch	333
Nuage networks VCS	335
Flannel	335
Calico	337
Romana	337
Weave Net	340
Using network policies effectively	340

Understanding the Kubernetes network policy design	340
Network policies and CNI plugins	341
Configuring network policies	341
Implementing network policies	342
Load balancing options	342
External load balancer	343
Configuring an external load balancer	344
Finding the load balancer IP addresses	344
Preserving client IP addresses	345
Understanding even external load balancing	346
Service load balancing	346
Ingress	347
HAProxy	349
MetalLB	351
Keepalived VIP	351
Traefic	351
Writing your own CNI plugin	352
First look at the loopback plugin	352
Building on the CNI plugin skeleton	356
Reviewing the bridge plugin	359
Summary	362
Chapter 11: Running Kubernetes on Multiple Clouds and Cluster Federation	365
The history of cluster federation on Kubernetes	366
Understanding cluster federation	366
Important use cases for cluster federation	368
Capacity overflow	368
Sensitive workloads	368
Avoiding vendor lock-in	369
Geo-distributing high availability	369
Learning the basics of Kubernetes federation	370
Defining basic concepts	370
Federation building blocks	370
Federation features	372
The KubeFed control plane	372
The federation API server	372
The federation controller manager	372
The hard parts	373
Federated unit of work	374
Location affinity	374
Cross-cluster scheduling	375
Federated data access	376
Federated auto-scaling	376
Managing a Kubernetes Cluster Federation	377
Installing kubefedctl	377

Creating clusters	379
Configuring the Host Cluster	379
Registering clusters with the federation	381
Working with federated API types	381
Federating resources	382
Federating an entire namespace	384
Checking the status of federated resources	384
Using overrides	385
Using placement to control federation	385
Debugging propagation failures	387
Employing higher-order behavior	387
Utilizing multi-cluster Ingress DNS	387
Utilizing multi-cluster Service DNS	388
Utilizing multi-cluster scheduling	389
Introducing the Gardener project	392
Understanding the terminology of Gardener	392
Understanding the conceptual model of Gardener	393
Diving into the Gardener architecture	394
Managing cluster state	394
Managing the control plane	395
Preparing the infrastructure	395
Using the Machine controller manager	395
Networking across clusters	395
Monitoring clusters	395
The gardenctl CLI	396
Extending Gardener	397
Gardener ring	401
Summary	402
Chapter 12: Serverless Computing on Kubernetes	405
Understanding serverless computing	405
Running long-running services on "serverless" infrastructure	406
Running FaaS on "serverless" infrastructure	407
Serverless Kubernetes in the cloud	408
Don't forget the cluster autoscaler	408
Azure AKS and Azure Container Instances	409
AWS EKS and Fargate	410
Google Cloud Run	412
Knative	413
Knative Serving	413
The Knative Service object	414
The Knative Route object	416
The Knative Configuration object	417
The Knative Revision object	420

Knative Eventing	420
Getting familiar with Knative Eventing terminology	420
The architecture of Knative Eventing	422
Taking Knative for a ride	423
Installing Knative	424
Deploying a Knative service	426
Invoking a Knative service	426
Checking the scale-to-zero option in Knative	427
Kubernetes FaaS frameworks	428
Fission	429
Fission Workflows	430
Experimenting with Fission	432
Kubeless	434
Kubeless architecture	434
Playing with Kubeless	435
Using the Kubeless UI	437
Kubeless with the serverless framework	438
Knative and riff	439
Understanding riff runtimes	439
Installing riff with Helm 2	439
Summary	442
Chapter 13: Monitoring Kubernetes Clusters	443
Understanding observability	444
Logging	444
Log format	445
Log storage	445
Log aggregation	445
Metrics	445
Distributed tracing	446
Application error reporting	447
Dashboards and visualization	447
Alerting	448
Logging with Kubernetes	448
Container logs	448
Kubernetes component logs	449
Centralized logging	450
Choosing a log collection strategy	450
Cluster-level central logging	452
Remote central logging	452
Dealing with sensitive log information	453
Using Fluentd for log collection	453
Collecting metrics with Kubernetes	454
Monitoring with the metrics server	455
Exploring your cluster with the Kubernetes dashboard	457

The rise of Prometheus	458
Installing Prometheus	460
Interacting with Prometheus	462
Incorporating kube-state-metrics	462
Utilizing the node exporter	464
Incorporating custom metrics	466
Alerting with Alertmanager	466
Visualizing your metrics with Grafana	468
Considering Loki	470
Distributed tracing with Jaeger	470
What is OpenTracing?	471
OpenTracing concepts	472
Introducing Jaeger	472
Jaeger architecture	473
Installing Jaeger	475
Troubleshooting problems	478
Taking advantage of staging environments	478
Detecting problems at the node level	479
Problem daemons	479
Dashboards versus alerts	480
Logs versus metrics versus error reports	481
Detecting performance and root cause with distributed tracing	482
Summary	482
Chapter 14: Utilizing Service Meshes	483
What is a service mesh?	483
Control plane and data plane	487
Choosing a service mesh	487
Envoy	487
Linkerd 2	487
Kuma	488
AWS App Mesh	488
Maesh	488
Istio	488
Incorporating Istio into your Kubernetes cluster	489
Understanding the Istio architecture	489
Envoy	490
Pilot	490
Mixer	491
Citadel	491
Galley	492
Preparing a minikube cluster for Istio	492
Installing Istio	493
Installing Bookinfo	495

Traffic management	499
Security	502
Istio identity	503
Istio PKI	504
Istio authentication	504
Istio authorization	505
Policies	509
Monitoring and observability	512
Logs	513
Metrics	516
Distributed tracing	519
Visualizing your service mesh with Kiali	522
Summary	523
Chapter 15: Extending Kubernetes	525
Working with the Kubernetes API	525
Understanding OpenAPI	526
Setting up a proxy	526
Exploring the Kubernetes API directly	526
Using Postman to explore the Kubernetes API	528
Filtering the output with HTTPie and jq	529
Creating a pod via the Kubernetes API	530
Accessing the Kubernetes API via the Python client	531
Dissecting the CoreV1API group	532
Listing objects	534
Creating objects	534
Watching objects	535
Invoking Kubectl programmatically	536
Using Python subprocesses to run Kubectl	536
Extending the Kubernetes API	538
Understanding Kubernetes extension points and patterns	539
Extending Kubernetes with plugins	539
Extending Kubernetes with the cloud controller manager	540
Extending Kubernetes with webhooks	541
Extending Kubernetes with controllers and operators	541
Extending Kubernetes scheduling	542
Extending Kubernetes with custom container runtimes	542
Introducing custom resources	542
Developing custom resource definitions	543
Integrating custom resources	545
Dealing with unknown fields	546
Finalizing custom resources	548
Adding custom printer columns	548
Understanding API server aggregation	549
Utilizing the service catalog	550
Writing Kubernetes plugins	552

Writing a custom scheduler	552
Understanding the design of the Kubernetes scheduler	552
Scheduling pods manually	555
Preparing our own scheduler	556
Assigning pods to the custom scheduler	557
Verifying that the pods were scheduled using the correct scheduler	558
Writing Kubectl plugins	559
Understanding Kubectl plugins	559
Managing Kubectl plugins with Krew	560
Creating your own Kubectl plugin	561
Kubectl plugin gotchas	562
Don't forget your shebangs!	562
Naming	562
Overriding existing Kubectl commands	562
Flat namespace for Krew plugins	563
Employing access control webhooks	563
Using an authentication webhook	564
Using an authorization webhook	566
Using an admission control webhook	568
Configuring a webhook admission controller on the fly	568
Providing custom metrics for horizontal pod autoscaling	570
Extending Kubernetes with custom storage	571
Summary	572
Chapter 16: The Future of Kubernetes	575
The Kubernetes momentum	576
The importance of the CNCF	576
Project curation	576
Certification	577
Training	578
Community and education	578
Tooling	578
The rise of managed Kubernetes platforms	579
Public cloud Kubernetes platforms	579
Bare-metal, private clouds, and Kubernetes on the edge	579
Kubernetes Platform as a Service (PaaS)	580
Upcoming trends	580
Security	580
Networking	581
Custom hardware and devices	582
Service mesh	582
Serverless computing	583
Kubernetes on the Edge	583
Native CI/CD	584
Operators	584

Summary	585
References	585
Other Books You May Enjoy	587
Index	591

Preface

Kubernetes is an open source system that automates the deployment, scaling, and management of containerized applications. If you are running more than just a few containers or want to automate the management of your containers, you need Kubernetes. This book focuses on guiding you through the advanced management of Kubernetes clusters.

The book begins by explaining the fundamentals behind Kubernetes' architecture and covers Kubernetes' design in detail. You will discover how to run complex stateful microservices on Kubernetes, including such advanced features as horizontal pod autoscaling, rolling updates, resource quotas, and persistent storage backends. Using real-world use cases, you will explore the options for network configuration and understand how to set up, operate, secure, and troubleshoot Kubernetes clusters. Finally, you will learn about advanced topics such as custom resources, API aggregation, service meshes, and serverless computing. All the content is up to date and complies with Kubernetes 1.18. By the end of this book, you'll know everything you need to know to go from intermediate to advanced level.

Who this book is for

The book is for system administrators and developers who have intermediate-level knowledge about Kubernetes and are now waiting to master its advanced features. You should also have basic networking knowledge. This advanced-level book provides a pathway to master Kubernetes.

What this book covers

Chapter 1, Understanding Kubernetes Architecture, in this chapter, we will build together the foundation necessary to utilize Kubernetes to its full potential. We will start by understanding what Kubernetes is, what Kubernetes isn't, and what container orchestration means exactly. Then we will cover important Kubernetes concepts that will form the vocabulary we will use throughout the book.

Chapter 2, Creating Kubernetes Clusters, in this chapter, we will roll up our sleeves and build some Kubernetes clusters using minikube, KinD, and k3d. We will discuss and evaluate other tools such as Kubeadm, Kube-spray, bootkube, and stackube. We will also look into deployment environments such as local, cloud, and bare metal.

Chapter 3, High Availability and Reliability, in this chapter, we will dive into the topic of highly available clusters. This is a complicated topic. The Kubernetes project and the community haven't settled on one true way to achieve high-availability nirvana. There are many aspects to highly available Kubernetes clusters, such as ensuring that the control plane can keep functioning in the face of failures, protecting the cluster state in etcd, protecting the system's data, and recovering capacity and/or performance quickly. Different systems will have different reliability and availability requirements.

Chapter 4, Securing Kubernetes, in this chapter, we will explore the important topic of security. Kubernetes clusters are complicated systems composed of multiple layers of interacting components. Isolation and compartmentalization of different layers is very important when running critical applications. To secure the system and ensure proper access to resources, capabilities, and data, we must first understand the unique challenges facing Kubernetes as a general-purpose orchestration platform that runs unknown workloads. Then we can take advantage of various securities, isolation, and access control mechanisms to make sure the cluster, the applications running on it, and the data are all safe. We will discuss various best practices and when it is appropriate to use each mechanism.

Chapter 5, Using Kubernetes Resources in Practice, in this chapter, we will design a fictional massive-scale platform that will challenge Kubernetes' capabilities and scalability. The Hue platform is all about creating an omniscient and omnipotent digital assistant. Hue is a digital extension of you. Hue will help you do anything, find anything, and, in many cases will do a lot on your behalf. It will obviously need to store a lot information, integrate with many external services, respond to notifications and events, and be smart about interacting with you.

Chapter 6, Managing Storage, in this chapter, we'll look at how Kubernetes manages storage. Storage is very different from compute, but at a high level they are both resources. Kubernetes as a generic platform takes the approach of abstracting storage behind a programming model and a set of plugins for storage providers.

Chapter 7, Running Stateful Applications with Kubernetes, in this chapter, we will learn how to run stateful applications on Kubernetes. Kubernetes takes a lot of work out of our hands by automatically starting and restarting pods across the cluster nodes as needed, based on complex requirements and configurations such as namespaces, limits, and quotas. But when pods run storage-aware software, such as databases and queues, relocating a pod can cause the system to break.

Chapter 8, Deploying and Updating Applications, in this chapter, we will explore the automated pod scalability that Kubernetes provides, how it affects rolling updates, and how it interacts with quotas. We will touch on the important topic of provisioning and how to choose and manage the size of the cluster. Finally, we will go over how the Kubernetes team improved the performance of Kubernetes and how they test the limits of Kubernetes with the Kubemark tool.

Chapter 9, Packaging Applications, in this chapter, we are going to look into Helm, the Kubernetes package manager. Every successful and non-trivial platform must have a good packaging system. Helm was developed by Deis (acquired by Microsoft on April 4, 2017) and later contributed to the Kubernetes project directly. It became a CNCF project in 2018. We will start by understanding the motivation for Helm, its architecture, and its components.

Chapter 10, Exploring Advanced Networking, in this chapter, we will examine the important topic of networking. Kubernetes as an orchestration platform manages containers/pods running on different machines (physical or virtual) and requires an explicit networking model.

Chapter 11, Running Kubernetes on Multiple Clouds and Cluster Federation, in this chapter, we'll take it to the next level, with running Kubernetes on multiple clouds, multiple clusters, and cluster federation. A Kubernetes cluster is a closely-knit unit where all the components run in relative proximity and are connected by a fast network (typically a physical data center or cloud provider availability zone). This is great for many use cases, but there are several important use cases where systems need to scale beyond a single cluster.

Chapter 12, Serverless Computing on Kubernetes, in this chapter, we will explore the fascinating world of serverless computing in the cloud. The term "serverless" is getting a lot of attention, but it is a misnomer used to describe two different paradigms. A true serverless application runs as a web application in the user's browser or a mobile app and only interacts with external services. The types of serverless systems we build on Kubernetes are different.

Chapter 13, Monitoring Kubernetes Clusters, in this chapter, we're going to talk about how to make sure your systems are up and running and performing correctly and how to respond when they aren't. In *Chapter 3, High Availability and Reliability*, we discussed related topics. The focus here is about knowing what's going on in your system and what practices and tools you can use.

Chapter 14, Utilizing Service Meshes, in this chapter, we will learn how service meshes allow you to externalize cross-cutting concerns like monitoring and observability from the application code. The service mesh is a true paradigm shift in the way you can design, evolve, and operate distributed systems on Kubernetes. I like to think of it as aspect-oriented programming for cloud-native distributed systems.

Chapter 15, Extending Kubernetes, in this chapter, we will dig deep into the guts of Kubernetes. We will start with the Kubernetes API and learn how to work with Kubernetes programmatically via direct access to the API, the Python client, and automating Kubectl. Then, we'll look into extending the Kubernetes API with custom resources. The last part is all about the various plugins Kubernetes supports. Many aspects of Kubernetes operation are modular and designed for extension. We will examine the API aggregation layer and several types of plugins, such as custom schedulers, authorization, admission control, custom metrics, and volumes. Finally, we'll look into extending Kubectl and adding your own commands.

Chapter 16, The Future of Kubernetes, in this chapter, we'll look at the future of Kubernetes from multiple angles. We'll start with the momentum of Kubernetes since its inception, across dimensions such as community, ecosystem, and mindshare. Spoiler alert: Kubernetes won the container orchestration wars by a land slide. As Kubernetes grows and matures, the battle lines shift from beating competitors to fighting against its own complexity. Usability, tooling, and education will play a major role as container orchestration is still new, fast-moving, and not a well-understood domain. Then we will take a look at some very interesting patterns and trends, and finally, we will review my predictions from the 2nd edition and I will make some new predictions.

To get the most out of this book

To follow the examples in each chapter, you need a recent version of Docker and Kubernetes installed on your machine, ideally Kubernetes 1.18. If your operating system is Windows 10 Professional, you can enable hypervisor mode; otherwise, you will need to install VirtualBox and use a Linux guest OS. If you use macOS then you're good to go.

Download the example code files

You can download the example code files for this book from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at <http://www.packtpub.com>.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the on-screen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for macOS
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Kubernetes-Third-Edition>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781839211256_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: If you chose HyperKit instead of VirtualBox, you need to add the flag `--vm-driver=hyperkit` when starting the cluster.

A block of code is set as follows:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
```

```
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

Any command-line input or output is written as follows:

```
$ k get pods
```

NAME	READY	STATUS	RESTARTS	AGE
echo-855975f9c-r6kj8	1/1	Running	0	2m11s

Bold: Indicates a new term, an important word, or words that you see on the screen, for example, in menus or dialog boxes, also appear in the text like this. For example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com, and mention the book's title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book we would be grateful if you would report this to us. Please visit, <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packtpub.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit <http://authors.packtpub.com>.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packtpub.com.

1

Understanding Kubernetes Architecture

In one sentence, Kubernetes is a platform to orchestrate the deployment, scaling, and management of container-based applications. You have probably read about Kubernetes, and maybe even dipped your toes in and used it in a side project or maybe even at work. But to understand what Kubernetes is all about, how to use it effectively, and what the best practices are requires much more.

Kubernetes is a big open source project and ecosystem with a lot of code and a lot of functionality. Kubernetes came out of Google, but joined the **Cloud Native Computing Foundation (CNCF)** and became the clear leader in the space of container-based applications.

In this chapter, we will build the foundation necessary to utilize Kubernetes to its full potential. We will start by understanding what Kubernetes is, what Kubernetes isn't, and what container orchestration means exactly. Then we will cover important Kubernetes concepts that will form the vocabulary we will use throughout the book. After that, we will dive into the architecture of Kubernetes proper and look at how it enables all the capabilities it provides for its users. Then, we will discuss the various runtimes and container engines that Kubernetes supports (Docker is just one option), and finally, we will discuss the role of Kubernetes in the full continuous integration and deployment pipeline.

At the end of this chapter, you will have a solid understanding of container orchestration, what problems Kubernetes addresses, the rationale of Kubernetes design and architecture, and the different runtimes it supports. You'll also be familiar with the overall structure of the open source repository and be ready to jump in and find answers to any questions.

What is Kubernetes?

Kubernetes is a platform that encompasses a huge number of services and capabilities that keeps growing. The core functionality is scheduling workloads in containers across your infrastructure, but it doesn't stop there. Here are some of the other capabilities Kubernetes brings to the table:

- Mounting storage systems
- Distributing secrets
- Checking application health and readiness
- Replicating application instances
- Using the Horizontal Pod Autoscaler
- Using Cluster Autoscaling
- Naming and service discovery
- Balancing loads
- Rolling updates
- Monitoring resources
- Accessing and ingesting logs
- Debugging applications
- Providing authentication and authorization

We will cover all these capabilities in great detail throughout the book. At this point, just absorb and appreciate how much value Kubernetes can add to your system.

Kubernetes has impressive scope, but it is also important to understand what Kubernetes explicitly doesn't provide.

What Kubernetes is not

Kubernetes is not a **Platform as a Service (PaaS)**. It doesn't dictate many important aspects that are left to you or to other systems built on top of Kubernetes, such as Deis, OpenShift, and Eldarion; for example:

- Kubernetes doesn't require a specific application type or framework
- Kubernetes doesn't require a specific programming language
- Kubernetes doesn't provide databases or message queues
- Kubernetes doesn't distinguish apps from services
- Kubernetes doesn't have a click-to-deploy service marketplace

- Kubernetes doesn't provide a built-in function as a service solution
- Kubernetes doesn't mandate a logging, monitoring, and alerting system

Now that we have a clear idea about the boundaries of Kubernetes, let's dive into its primary responsibility – container orchestration.

Understanding container orchestration

The primary responsibility of Kubernetes is container orchestration. That means making sure that all the containers that execute various workloads are scheduled to run on physical or virtual machines. The containers must be packed efficiently following the constraints of the deployment environment and the cluster configuration. In addition, Kubernetes must keep an eye on all running containers and replace dead, unresponsive, or otherwise unhealthy containers. Kubernetes provides many more capabilities, which you will learn about in the following chapters. In this section, the focus is on containers and their orchestration.

Physical machines, virtual machines, and containers

It all starts and ends with hardware. In order to run your workloads, you need some real hardware provisioned. That includes actual physical machines with certain compute capabilities (CPUs or cores), memory, and some local persistent storage (spinning disks or SSDs). In addition, you will need some shared persistent storage and to hook up all these machines using networking, so they can find and talk to each other. At this point, you run multiple virtual machines on the physical machines or stay at the bare-metal level (real hardware only – no virtual machines). Kubernetes can be deployed on a bare-metal cluster or on a cluster of virtual machines. Kubernetes, in turn, can orchestrate the containers it manages directly on bare metal or on virtual machines. In theory, a Kubernetes cluster can be composed of a mix of bare-metal and virtual machines, but this is not very common.

The benefits of containers

Containers represent a true paradigm shift in the development and operation of large, complicated software systems. Here are some of the benefits compared to more traditional models:

- Agile application creation and deployment
- Continuous development, integration, and deployment
- Dev and Ops separation of concerns

- Environmental consistency across development, testing, staging, and production
- Cloud and OS distribution portability
- Application-centric management (dependencies are packaged with the application)
- Resource isolation (container CPU and memory can be limited)
- Resource utilization (multiple containers can be deployed on the same node)

The benefits of container-based development and deployment are significant in many contexts, but are particularly significant if you deploy your system to the cloud.

Containers in the cloud

Containers are ideal to package microservices because while providing isolation to the microservice, they are very lightweight and you don't incur a lot of overhead when deploying many microservices as you do with virtual machines. That makes containers ideal for cloud deployment, where allocating a whole virtual machine for each microservice would be cost-prohibitive.

All major cloud providers, such as **Amazon Web Services (AWS)**, **Google Cloud Platform (GCP)**, and Microsoft's Azure, provide container hosting services these days. Many other companies have jumped on the Kubernetes wagon and offer managed Kubernetes services, including:

- IBM IKS
- Alibaba Cloud
- DigitalOcean DKS
- Oracle OKS
- OVH Managed Kubernetes
- Rackspace KaaS

The **Google Kubernetes Engine (GKE)** was always based on Kubernetes. Amazon's **Elastic Kubernetes Service (EKS)** was added in addition to the proprietary AWS ECS orchestration solution. Microsoft Azure's container service used to be based on Apache Mesos but later switched to Kubernetes with **Azure Kubernetes Service (AKS)**. You could always deploy Kubernetes on all the cloud platforms, but it wasn't deeply integrated with other services. However, at the end of 2017, all cloud providers announced direct support for Kubernetes. Microsoft's launched AKS, AWS released EKS, and Alibaba Cloud started working on a Kubernetes controller manager to integrate Kubernetes seamlessly.

Cattle versus pets

In the olden days, when systems were small, each server had a name. Developers and users knew exactly what software was running on each machine. I remember that, in many of the companies I worked for, we had multi-day discussions to decide on a naming theme for our servers. For example, composers and Greek mythology characters were popular choices. Everything was very cozy. You treated your servers like beloved pets. When a server died it was a major crisis. Everybody scrambled to try to figure out where to get another server, what was even running on the dead server, and how to get it working on the new server. If the server stored some important data, then hopefully you had an up-to-date backup and maybe you'd even be able to recover it.

Obviously, that approach doesn't scale. When you have tens or hundreds of servers, you must start treating them like cattle. You think about the collective and not individuals. You may still have some pets like your CI/CD machines (although managed CI/CD solutions are becoming more common), but your web servers and backend services are just cattle.

Kubernetes takes the cattle approach to the extreme and takes full responsibility for allocating containers to specific machines. You don't need to interact with individual machines (nodes) most of the time. This works best for stateless workloads. For stateful applications, the situation is a little different, but Kubernetes provides a solution called `StatefulSet`, which we'll discuss soon.

In this section, we covered the idea of container orchestration and discussed the relationships between hosts (physical or virtual) and containers, as well as the benefits of running containers in the cloud, and finished with a discussion about cattle versus pets. In the following section, we will get to know the world of Kubernetes and learn its concepts and terminology.

Kubernetes concepts

In this section, I'll briefly introduce many important Kubernetes concepts and give you some context as to why they are needed and how they interact with other concepts. The goal is to get familiar with these terms and concepts. Later, we'll see how these concepts are woven together and organized into API groups and resource categories to achieve awesomeness. You can consider many of these concepts as building blocks. Some of the concepts, such as nodes and masters, are implemented as a set of Kubernetes components. These components are at a different abstraction level, and I discuss them in detail in a dedicated section later in this chapter – *Kubernetes components*.

Here is the famous Kubernetes architecture diagram:

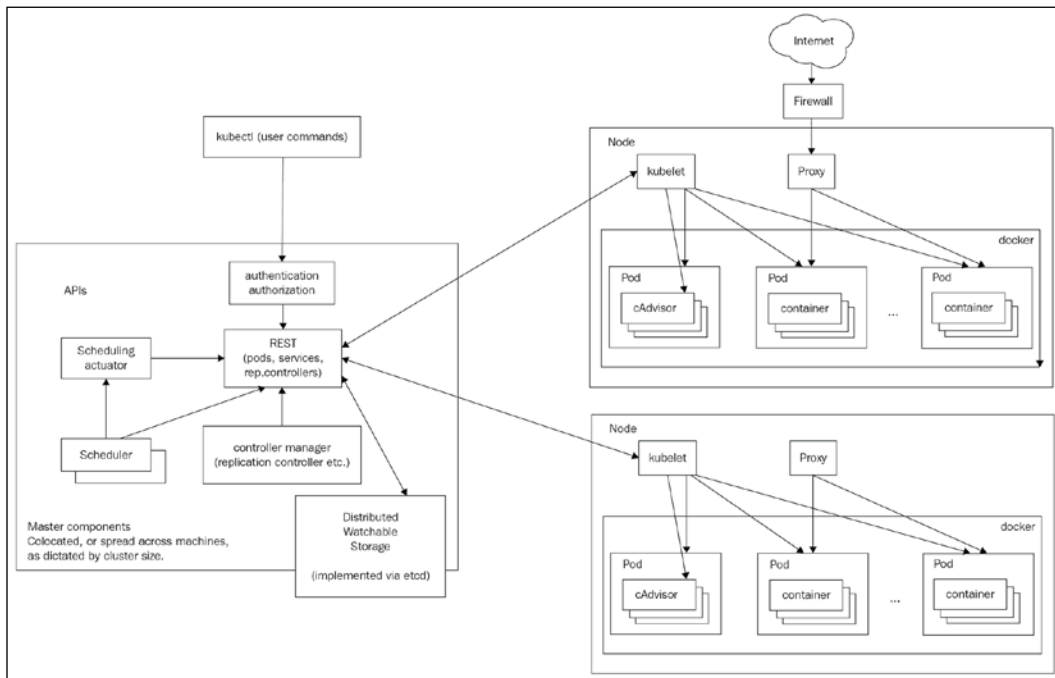


Figure 1.1: Kubernetes architecture diagram

Clusters

A cluster is a collection of hosts (nodes) that provide compute, memory, storage, and networking resources. Kubernetes uses these resources to run the various workloads that comprise your system. Note that your entire system may consist of multiple clusters. We will discuss this advanced use case of federation in detail in *Chapter 11, Running Kubernetes on Multiple Clouds and Cluster Federation*.

Nodes

A node is a single host. It may be a physical or virtual machine. Its job is to run pods. Each Kubernetes node runs several Kubernetes components, such as the kubelet, the container runtime, and kube-proxy. Nodes are managed by a Kubernetes master. The nodes are the worker bees of Kubernetes and shoulder all the heavy lifting. In the past, they were called **minions**. If you read some old documentation or articles, don't get confused. Minions are just nodes.

The master

The master is the control plane of Kubernetes. It consists of several components, such as an API server, a scheduler, and a controller manager. The master is responsible for the global state of the cluster, cluster-level scheduling of pods, and handling of events. Usually, all the master components are set up on a single host. When considering high-availability scenarios or very large clusters, you will want to have master redundancy. We will discuss highly available clusters in detail in *Chapter 4, Securing Kubernetes*.

Pods

A pod is the unit of work in Kubernetes. Each pod contains one or more containers. Containers in pods are always scheduled together (always run on the same machine). All the containers in a pod have the same IP address and port space; they can communicate using localhost or standard inter-process communication. In addition, all the containers in a pod can have access to shared local storage on the node hosting the pod. Containers don't get access to local storage or any other storage by default. Volumes of storage must be mounted into each container inside the pod explicitly. Pods are an important feature of Kubernetes. It is possible to run multiple applications inside a single Docker container by having something like supervisord as the main Docker process that runs multiple processes, but this practice is often frowned upon for the following reasons:

- **Transparency:** Making the containers within the pod visible to the infrastructure enables the infrastructure to provide services to those containers, such as process management and resource monitoring. This facilitates a number of conveniences for users.
- **Decoupling software dependencies:** The individual containers may be versioned, rebuilt, and redeployed independently. Kubernetes may even support live updates of individual containers someday.
- **Ease of use:** Users don't need to run their own process managers, worry about signal and exit-code propagation, and so on.
- **Efficiency:** Because the infrastructure takes on more responsibility, containers can be more lightweight.

Pods provide a great solution for managing groups of closely related containers that depend on each other and need to co-operate on the same host to accomplish their purpose. It's important to remember that pods are considered ephemeral, throwaway entities that can be discarded and replaced at will. Any pod storage is destroyed with its pod. Each pod gets a **unique ID (UID)**, so you can still distinguish between them if necessary.

Labels

Labels are key-value pairs that are used to group together sets of objects – very often pods. This is important for several other concepts, such as replication controllers, replica sets, deployments, and services that operate on dynamic groups of objects and need to identify the members of the group. There is an $N \times N$ relationship between objects and labels. Each object may have multiple labels, and each label may be applied to different objects. There are certain restrictions on labels by design. Each label on an object must have a unique key. The label key must adhere to a strict syntax. It has two parts: prefix and name. The prefix is optional. If it exists then it is separated from the name by a forward slash (/) and it must be a valid DNS sub-domain. The prefix must be 253 characters long at most. The name is mandatory and must be 63 characters long at most. Names must start and end with an alphanumeric character (a-z, A-Z, 0-9) and contain only alphanumeric characters, dots, dashes, and underscores. Values follow the same restrictions as names. Note that labels are dedicated to identifying objects and not for attaching arbitrary metadata to objects. This is what annotations are for.

Annotations

Annotations let you associate arbitrary metadata with Kubernetes objects. Kubernetes just stores the annotations and makes their metadata available. Annotations, like labels, are key-value pairs where the key may have an optional prefix and is separated from the key name by a forward slash (/). The name and prefix (if provided) must follow strict rules. For details, check out <https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/#syntax-and-character-set>.

In my experience, you always need such metadata for complicated systems, and it's nice that Kubernetes recognizes this need and provides it out of the box so you don't have to come up with your own separate metadata store and mapping object for their metadata. While annotations are useful, their lack of structure can pose some problems when trying to process annotations in a generic way. Custom resource definitions are often touted as an alternative. We'll cover those later, in *Chapter 15, Extending Kubernetes*.

Label selectors

Label selectors are used to select objects based on their labels. Equality-based selectors specify a key name and a value. There are two operators, = (or ==) and !=, for equality or inequality based on the value; for example:

```
role = webserver
```

This will select all objects that have that label key and value.

Label selectors can have multiple requirements separated by a comma; for example:

```
role = webserver, application != foo
```

Set-based selectors extend the capabilities, and allow selection based on multiple values:

```
role in (webserver, backend)
```

Services

Services are used to expose some functionality to users or other services. They usually encompass a group of pods, usually identified by – you guessed it – a label. You can have services that provide access to external resources, or to pods you control directly at the virtual IP level. Native Kubernetes services are exposed through convenient endpoints. Note that services operate at layer 3 (TCP/UDP). Kubernetes 1.2 added the Ingress object, which provides access to HTTP objects. More on that later. Services are published or discovered via one of two mechanisms: DNS or environment variables. Services can be load-balanced by Kubernetes. However, developers can choose to manage load balancing themselves in the case of services that use external resources or require special treatment.

There are many gory details associated with IP addresses, virtual IP addresses, and port spaces. We will discuss them in depth in *Chapter 10, Exploring Advanced Networking*.

Volume

Local storage on the pod is ephemeral and goes away with the pod. Sometimes that's all you need if the goal is just to exchange data between containers of the node, but sometimes it's important for the data to outlive the pod, or it's necessary to share data between pods. The volume concept supports that need. Note that, while Docker has a volume concept too, it's quite limited (although getting more powerful). Kubernetes uses its own separate volumes. Kubernetes also supports additional container runtimes, so it can't rely on Docker volumes even in principle.

There are many volume types. Kubernetes currently directly supports many volume types, but the modern approach to extending Kubernetes with more volume types is through the **Container Storage Interface (CSI)**, which we'll discuss in detail later. The built-in volume types will be gradually phased out in favor of out-of-tree plugins available through the CSI.

Replication controllers and replica sets

Replication controllers and replica sets both manage a group of pods identified by a label selector and ensure that a certain number are always up and running. The main difference between them is that replication controllers test for membership by name equality and replica sets can use set-based selection. Replica sets are the way to go as they are a superset of replication controllers. I expect replication controllers to be deprecated at some point. Kubernetes guarantees that you will always have the same number of pods running as you specified in a replication controller or a replica set. Whenever the number drops due to a problem with the hosting node or the pod itself, Kubernetes will fire up new instances. Note that, if you manually start pods and exceed the specified number, the replica set controller will kill some extra pods.

Replication controllers used to be central to many workflows, such as rolling updates and running one-off jobs. As Kubernetes evolved, it introduced direct support for many of these workflows, with dedicated objects such as Deployment, Job, CronJob, and DaemonSet. We will meet them all later.

StatefulSet

Pods come and go, and if you care about their data then you can use persistent storage. That's all good. But sometimes you want Kubernetes to manage a distributed data store such as Cassandra or MySQL Galera. These clustered stores keep the data distributed across uniquely identified nodes. You can't model that with regular pods and services. Enter StatefulSet. If you remember, earlier we discussed pets versus cattle and how the cattle mindset is the way to go.

Well, StatefulSet sits somewhere in the middle. StatefulSet ensures (similar to a replication set) that a given number of instances with unique identities are running at any given time. StatefulSet members have the following properties:

- A stable hostname, available in DNS
- An ordinal index
- Stable storage linked to the ordinal and hostname

StatefulSet can help with peer discovery as well as adding or removing members safely.

Secrets

Secrets are small objects that contain sensitive info such as credentials and tokens. They are stored by default as plaintext in etcd, accessible by the Kubernetes API server, and can be mounted as files in pods (using dedicated secret volumes that piggyback on regular data volumes) that need access to them. The same secret can be mounted in multiple pods. Kubernetes itself creates secrets for its components, and you can create your own secrets. Another approach is to use secrets as environment variables. Note that secrets in a pod are always stored in memory (tmpfs in the case of mounted secrets) for better security.

Names

Each object in Kubernetes is identified by a UID and a name. The name is used to refer to the object in API calls. Names should be up to 253 characters long and use lowercase alphanumeric characters, a dash (-), and a dot (.). If you delete an object, you can create another object with the same name as the deleted object, but the UIDs must be unique across the lifetime of the cluster. The UIDs are generated by Kubernetes, so you don't have to worry about it.

Namespaces

A namespace is a kind of virtual cluster. You can have a single physical cluster that contains multiple virtual clusters segregated by namespaces. By default, pods in one namespace can access pods and services in other namespaces. In multi-tenancy scenarios where it's important to totally isolate namespaces, you can do it with proper network policies. Note that node objects and persistent volumes don't live in a namespace. Kubernetes may schedule pods from different namespaces to run on the same node. Likewise, pods from different namespaces can use the same persistent storage.

When using namespaces, you have to consider network policies and resource quotas to ensure proper access and distribution of the physical cluster resources.

We've covered most of Kubernetes' primary concepts; there are a few more I mentioned briefly. In the next section, we will continue our journey into Kubernetes architecture by looking into its design motivations, the internals and implementation, and we'll even pick at the source code.

Diving into Kubernetes architecture in depth

Kubernetes has very ambitious goals. It aims to manage and simplify the orchestration, deployment, and management of distributed systems across a wide range of environments and cloud providers. It provides many capabilities and services that should work across all that diversity while evolving and remaining simple enough for mere mortals to use. This is a tall order. Kubernetes achieves this by following a crystal-clear, high-level design and well-thought-out architecture that promotes extensibility and pluggability. Many parts of Kubernetes are still hardcoded or environment-aware, but the trend is to refactor them into plugins and keep the core small, generic, and abstract. In this section, we will peel Kubernetes like an onion, starting with various distributed system design patterns and how Kubernetes supports them, then go over the surface of Kubernetes, which is its set of APIs, and then take a look at the actual components that comprise Kubernetes. Finally, we will take a quick tour of the source-code tree to gain an even better insight into the structure of Kubernetes itself.

At the end of this section, you will have a solid understanding of Kubernetes architecture and implementation, and why certain design decisions were made.

Distributed system design patterns

All happy (working) distributed systems are alike, to paraphrase Tolstoy in *Anna Karenina*. That means that to function properly, all well-designed distributed systems must follow some best practices and principles. Kubernetes doesn't want to be just a management system; it wants to support and enable these best practices and provide high-level services to developers and administrators. Let's look at some of those best practices, described as design patterns.

The sidecar pattern

The sidecar pattern is about co-locating another container in a pod in addition to the main application container. The application container is unaware of the sidecar container and just goes about its business. A great example is a central logging agent. Your main container can just log to stdout, but the sidecar container will send all logs to a central logging service where they will be aggregated with the logs from the entire system. The benefits of using a sidecar container versus adding central logging to the main application container are enormous. First, applications are not burdened anymore with central logging, which could be a nuisance. If you want to upgrade or change your central logging policy or switch to a totally new provider, you just need to update the sidecar container and deploy it. None of your application containers change, so you can't break them by accident. The Istio service mesh uses the sidecar pattern to inject its proxies into each pod.

The ambassador pattern

The ambassador pattern is about representing a remote service as if it were local and possibly enforcing some policy. A good example of the ambassador pattern is if you have a Redis cluster with one master for writes and many replicas for reads. A local ambassador container can serve as a proxy and expose Redis to the main application container on the localhost. The main application container simply connects to Redis on localhost:6379 (Redis default port), but it connects to the ambassador running in the same pod, which filters the requests, and sends write requests to the real Redis master and read requests randomly to one of the read replicas. Just like with the sidecar pattern, the main application has no idea what's going on. That can help a lot when testing against a real local Redis. Also, if the Redis cluster configuration changes, only the ambassador needs to be modified; the main application remains blissfully unaware.

The adapter pattern

The adapter pattern is about standardizing output from the main application container. Consider the case of a service that is being rolled out incrementally: it may generate reports in a format that doesn't conform to the previous version. Other services and applications that consume that output haven't been upgraded yet. An adapter container can be deployed in the same pod with the new application container and massage the output to match the old version until all consumers have been upgraded. The adapter container shares the filesystem with the main application container, so it can watch the local filesystem, and whenever the new application writes something, it immediately adapts it.

Multi-node patterns

Single-node patterns are all supported directly by Kubernetes via pods. Multi-node patterns such as leader election, work queues, and scatter-gather are not supported directly, but composing pods with standard interfaces to accomplish them is a viable approach with Kubernetes.

Many tools, frameworks, and add-ons that integrate deeply with Kubernetes utilize these design patterns. The beauty of these patterns is that they are all loosely coupled and don't require Kubernetes to be modified or even be aware of the presence of these integrations. The vibrant ecosystem around Kubernetes is a direct result of its architecture. Let's dig one level deeper and get familiar with the Kubernetes APIs.

The Kubernetes APIs

If you want to understand the capabilities of a system and what it provides, you must pay a lot of attention to its API. The API provides a comprehensive view of what you can do with the system as a user. Kubernetes exposes several sets of REST APIs for different purposes and audiences via API groups. Some of the APIs are used primarily by tools and some can be used directly by developers. An important aspect of the APIs is that they are under constant development. The Kubernetes developers keep it manageable by trying to extend (adding new objects and new fields to existing objects) and avoid renaming or dropping existing objects and fields. In addition, all API endpoints are versioned and often have an alpha or beta notation too; for example:

`/api/v1`

`/api/v2alpha1`

You can access the API through the `kubectl` CLI, via client libraries, or directly through REST API calls. There are elaborate authentication and authorization mechanisms we will explore in a later chapter. If you have the right permissions, you can list, view, create, update, and delete various Kubernetes objects. At this point, let's get a glimpse of the surface area of the APIs. The best way to explore the API is via API groups. Some API groups are enabled by default. Other groups can be enabled/disabled via flags. For example, to disable the batch V1 group and enable the batch V2 Alpha group, you can set the `--runtime-config` flag when running the API server as follows:

`--runtime-config=batch/v1=false,batch/v2alpha=true`

The following resources are enabled by default in addition to the core resources:

- DaemonSets
- Deployments
- HorizontalPodAutoscalers
- Ingress
- Jobs
- ReplicaSets

In addition to API groups, another useful classification of available APIs is by functionality. Enter resource categories...

Resource categories

The Kubernetes API is huge, and breaking it down into categories helps a lot when you're trying to find your way around. Kubernetes defines the following resource categories:

- **Workloads:** Objects you use to manage and run containers in the cluster
- **Discovery and Load Balancing:** Objects you use to expose your workloads to the world as externally accessible, load-balanced services
- **Config and Storage:** Objects you use to initialize and configure your applications, and to persist data that's outside the container
- **Cluster:** Objects that define how the cluster itself is configured; these are typically used only by cluster operators
- **Metadata:** Objects you use to configure the behavior of other resources within the cluster, such as HorizontalPodAutoscaler for scaling workloads

In the following sub-sections, I'll list the resources that belong to each group with the API group they belong to in the following format: <resource name>: <API group>; for example, **Container: core**, where the resource is **Container** and the API group is **core**. I will not specify the version here because APIs move rapidly from alpha to beta to GA (general availability) and from V1 to V2, and so on.

The workloads API

The workloads API contains many resources. Here is a list of all the resources with the API groups they belong to:

- Container: core
- CronJob: batch
- DaemonSet: apps
- Deployment: apps
- Job: batch
- Pod: core
- ReplicaSet: apps
- ReplicationController: core
- StatefulSet: apps

Containers are created by controllers through pods. Pods run containers and provide environmental dependencies such as shared or persistent storage volumes and configuration or secret data injected into the container.

Here is an example of the detailed documentation of one of the most common operations – getting a list of all the pods as a REST API:

```
GET /api/v1/pods
```

It accepts various query parameters (all optional):

- pretty: If true, the output is pretty printed
- labelSelector: A selector expression to limit the result
- watch: If true, watch for changes and return a stream of events
- resourceVersion: With watch, returns only events that occurred after that version
- timeoutSeconds: Timeout for the list or watch operation

The next category of resources deals with high-level networking.

Discovery and Load Balancing

This category is also known as service APIs. By default, workloads are only accessible within the cluster, and they must be exposed externally using either a LoadBalancer or NodePort Service.

For development, internally accessible workloads can be accessed via proxy through the API master using the `kubectl proxy` command:

- Endpoints: `core`
- Ingress: `networking.k8s.io`
- Service: `core`

The next category of resources deals with storage and internal state management.

Config and Storage

Dynamic configuration without redeployment is a cornerstone of Kubernetes and running complex distributed applications on your Kubernetes cluster. Storing data is another paramount concern for any non-trivial system. The config and storage category provides multiple resources to address these concerns:

- ConfigMap: `core`
- CSIDriver: `storage.k8s.io`
- CSINode: `storage.k8s.io`
- Secret: `core`
- PersistentVolumeClaim: `core`
- StorageClass: `storage.k8s.io`
- Volume: `storage.k8s.io`
- VolumeAttachment: `storage.k8s.io`

The next category of resources deals with helper resources that are usually part of other high-level resources.

Metadata

The metadata resources typically show up as sub-resources of the resources of the configuration. For example, a limit range will be part of a pod configuration. You will not interact with these objects directly most of the time. There are many metadata resources – there isn't much point in listing all of them. You can find the complete list here: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.16/#-strong-metadata-apis-strong->.

Clusters

The resources in the cluster category are designed for use by cluster operators as opposed to developers. There are many resources in this category as well. Here are some of the most important resources:

- Namespace: core
- Node: core
- PersistentVolume: core
- ResourceQuota: core
- Role: rbac.authorization.k8s.io
- RoleBinding: rbac.authorization.k8s.io
- ClusterRole: rbac.authorization.k8s.io
- ClusterRoleBinding: rbac.authorization.k8s.io
- NetworkPolicy: networking.k8s.io

Now that we understand how Kubernetes organizes and exposes its capabilities via API groups and resource categories, let's see how it manages the physical infrastructure and keeps it up with the state of the cluster.

Kubernetes components

A Kubernetes cluster has several master components used to control the cluster, as well as node components that run on each worker node. Let's get to know all these components and how they work together.

Master components

The master components can all run on one node, but in a highly available setup or a very large cluster, they may be spread across multiple nodes.

The API server

The Kubernetes API server exposes the Kubernetes REST API. It can easily scale horizontally as it is stateless and stores all the data in the etcd cluster. The API server is the embodiment of the Kubernetes control plane.

Etcd

Etcd is a highly reliable distributed data store. Kubernetes uses it to store the entire cluster state. In a small, transient cluster a single instance of etcd can run on the same node with all the other master components. But for more substantial clusters, it is typical to have a three-node or even five-node etcd cluster for redundancy and high availability.

The Kube controller manager

The Kube controller manager is a collection of various managers rolled up into one binary. It contains the replication controller, the pod controller, the services controller, the endpoints controller, and others. All these managers watch over the state of the cluster via the API and their job is to steer the cluster into the desired state.

Cloud controller managers

When running in the cloud, Kubernetes allows cloud providers to integrate their platform for the purpose of managing nodes, routes, services, and volumes. The cloud provider code interacts with the Kubernetes code. It replaces some of the functionality of the Kube controller manager. When running Kubernetes with a cloud controller manager, you must set the Kube controller manager flag `--cloud-provider` to "external". This will disable the control loops that the cloud controller manager is taking over. The cloud controller manager was introduced in Kubernetes 1.6 and it's being used by multiple cloud providers already, such as:

- GCP
- AWS
- Azure
- Baidu Cloud
- DigitalOcean
- Oracle
- Linode

A quick note about Go to help you parse the code: the method name comes first, followed by the method's parameters in parentheses. Each parameter is a pair, consisting of a name followed by its type. Finally, the return values are specified. Go allows multiple return types. It is very common to return an error object in addition to the actual result. If everything is OK, the error object will be nil.

Here is the main interface of the cloudprovider package:

```
package cloudprovider
import (
    "errors"
    "fmt"
    "strings"
    "k8s.io/api/core/v1"
    "k8s.io/apimachinery/pkg/types"
    "k8s.io/client-go/informers"
    "k8s.io/kubernetes/pkg/controller"
)
// Interface is an abstract, pluggable interface for cloud providers.
type Interface interface {
    Initialize(clientBuilder controller.ControllerClientBuilder)
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
    HasClusterID() bool
}
```

Most of the methods return other interfaces with their own method. For example, here is the LoadBalancer interface:

```
type LoadBalancer interface {
    GetLoadBalancer(clusterName string,
        service *v1.Service) (status *v1.LoadBalancerStatus,
        exists bool,
        err error)

    EnsureLoadBalancer(clusterName string,

        service *v1.Service,
        nodes []*v1.Node) (*v1.LoadBalancerStatus, error)

    UpdateLoadBalancer(clusterName string, service *v1.Service, nodes
        []*v1.Node) error

    EnsureLoadBalancerDeleted(clusterName string, service *v1.Service)
    error
}
```

The cloud controller manager is instrumental in bringing Kubernetes to all the major cloud providers, but the heart and soul of Kubernetes is the scheduler.

kube-scheduler

Kube-scheduler is responsible for scheduling pods into nodes. This is a very complicated task as it needs to consider multiple interacting factors, such as the following:

- Resource requirements
- Service requirements
- Hardware/software policy constraints
- Node affinity and anti-affinity specifications
- Pod affinity and anti-affinity specifications
- Taints and tolerations
- Data locality
- Deadlines

If you need some special scheduling logic not covered by the default kube-scheduler, you can replace it with your own custom scheduler. You can also run your custom scheduler side by side with the default scheduler and have your custom scheduler schedule only a subset of the pods.

DNS

Starting with Kubernetes 1.3, a DNS service is part of the standard Kubernetes cluster. It is scheduled as a regular pod. Every service (except headless services) receives a DNS name. Pods can receive a DNS name too. This is very useful for automatic discovery.

Node components

Nodes in the cluster need a couple of components to interact with the cluster master components, receive workloads to execute, and update the Kubernetes API server regarding their status.

Proxy

Kube-proxy does low-level network housekeeping on each node. It reflects the Kubernetes services locally and can perform TCP and UDP forwarding. It finds cluster IPs via environment variables or DNS.

Kubelet

The kubelet is the Kubernetes representative on the node. It oversees communicating with the master components and manages the running pods. That includes the following:

- Receiving pod specs
- Downloading pod secrets from the API server
- Mounting volumes
- Running the pod's containers (via the configured runtime)
- Reporting the status of the node and each pod
- Running the container startup, liveness, and readiness probes

In this section, we dug into the guts of Kubernetes and explored its architecture from a very high level of vision and supported design patterns, through its APIs and the components used to control and manage the cluster. In the next section, we will take a quick look at the various runtimes that Kubernetes supports.

Kubernetes runtimes

Kubernetes originally only supported Docker as a container runtime engine. But that is no longer the case. Kubernetes now supports several different runtimes:

- Docker (via a CRI shim)
- rkt (direct integration to be replaced with Rktlet)
- CRI-O
- Frakti (Kubernetes on the Hypervisor, previously Hypernetes)
- rktlet (CRI implementation for rkt)
- CRI-containerd

The major design policy is that Kubernetes itself should be completely decoupled from specific runtimes. The **Container Runtime Interface (CRI)** enables it.

In this section, you'll get a closer look at the CRI and get to know the individual runtime engines. At the end of this section, you'll be able to make a well-informed decision about which runtime engine is appropriate for your use case and under what circumstances you may switch or even combine multiple runtimes in the same system.

The container runtime interface (CRI)

The CRI is a collection of a gRPC API, specifications/requirements, and libraries for container runtimes to integrate with a kubelet on a node. In Kubernetes 1.7, the internal Docker integration in Kubernetes was replaced with a CRI-based integration. This is a big deal. It opened the door to multiple implementations that can take advantage of advances in the container world. The kubelet doesn't need to interface directly with multiple runtimes. Instead, it can talk to any CRI-compliant container runtime. The following diagram illustrates the flow:

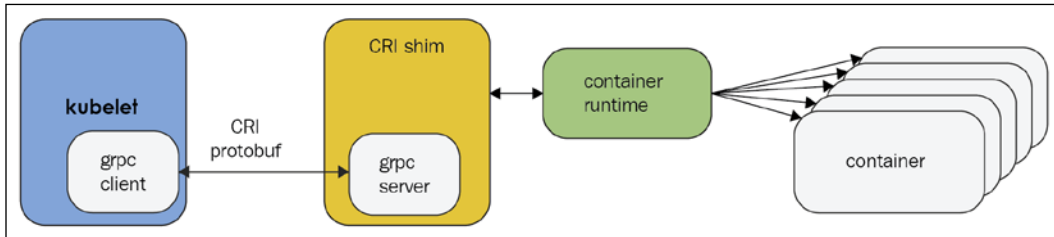


Figure 1.2: The container runtime interface (CRI) flow diagram

There are two gRPC service interfaces, `ImageService` and `RuntimeService`, that CRI container runtimes (or shims) must implement. `ImageService` is responsible for managing images. Here is the gRPC/protobuf interface (this is Google's Protobuf specification language and not Go):

```

service ImageService {

    rpc ListImages(ListImagesRequest) returns (ListImagesResponse) {}

    rpc ImageStatus(ImageStatusRequest) returns (ImageStatusResponse) {}

    rpc PullImage(PullImageRequest) returns (PullImageResponse) {}

    rpc RemoveImage(RemoveImageRequest) returns (RemoveImageResponse) {}

    rpc ImageFsInfo(ImageFsInfoRequest) returns (ImageFsInfoResponse) {}
}
  
```


RuntimeService is responsible for managing pods and containers. Here is the gRPC/protobuf interface:

```
service RuntimeService {

    rpc Version(VersionRequest) returns (VersionResponse) {}

    rpc RunPodSandbox(RunPodSandboxRequest) returns
(RunPodSandboxResponse) {}

    rpc StopPodSandbox(StopPodSandboxRequest) returns
(StopPodSandboxResponse) {}

    rpc RemovePodSandbox(RemovePodSandboxRequest) returns
(RemovePodSandboxResponse) {}

    rpc PodSandboxStatus(PodSandboxStatusRequest) returns
(PodSandboxStatusResponse) {}

    rpc ListPodSandbox(ListPodSandboxRequest) returns
(ListPodSandboxResponse) {}

    rpc CreateContainer(CreateContainerRequest) returns
(CreateContainerResponse) {}

    rpc StartContainer(StartContainerRequest) returns
(StartContainerResponse) {}

    rpc StopContainer(StopContainerRequest) returns
(StopContainerResponse) {}

    rpc RemoveContainer(RemoveContainerRequest) returns
(RemoveContainerResponse) {}

    rpc ListContainers(ListContainersRequest) returns
(ListContainersResponse) {}

    rpc ContainerStatus(ContainerStatusRequest) returns
(ContainerStatusResponse) {}

    rpc UpdateContainerResources(UpdateContainerResourcesRequest)
returns (UpdateContainerResourcesResponse) {}

    rpc ExecSync(ExecSyncRequest) returns (ExecSyncResponse) {}
```

```

    rpc Exec(ExecRequest) returns (ExecResponse) {}

    rpc Attach(AttachRequest) returns (AttachResponse) {}

    rpc PortForward(PortForwardRequest) returns (PortForwardResponse) {}

    rpc ContainerStats(ContainerStatsRequest) returns
(ContainerStatsResponse) {}

    rpc ListContainerStats(ListContainerStatsRequest) returns
(ListContainerStatsResponse) {}

    rpc UpdateRuntimeConfig(UpdateRuntimeConfigRequest) returns
(UpdateRuntimeConfigResponse) {}

    rpc Status(StatusRequest) returns (StatusResponse) {}
}

```

The data types used as arguments and return types are called messages and are also defined as part of the API. Here is one of them:

```

message CreateContainerRequest {
    string pod\_sandbox\_id = 1;
    ContainerConfig config = 2;
    PodSandboxConfig sandbox\_config = 3;
}

```

As you can see, messages can be embedded inside each other. The `CreateContainerRequest` message has one string field and two other fields, which are themselves messages: `ContainerConfig` and `PodSandboxConfig`.

Now that you are familiar at the code level with what Kubernetes considers a runtime engine, let's look at the individual runtime engines briefly.

Docker

Docker is, of course, the 800-pound gorilla of containers. Kubernetes was originally designed to manage only Docker containers. The multi-runtime capability was first introduced in Kubernetes 1.3 and the CRI in Kubernetes 1.5. Until then, Kubernetes could only manage Docker containers.

I assume you're very familiar with Docker and what it brings to the table if you are reading this book. Docker enjoys tremendous popularity and growth, but there is also a lot of criticism of it. Critics often mention the following concerns:

- Security
- Difficulty setting up multi-container applications (in particular, networking)
- Development, monitoring, and logging
- The limitations of Docker containers running one command
- Releasing half-baked features too fast

Docker is aware of the criticisms and has addressed some of these concerns. In particular, Docker invested in its Docker Swarm product. Docker Swarm is a Docker-native orchestration solution that competes with Kubernetes. It is simpler to use than Kubernetes, but it's not as powerful or mature.

Starting with Docker 1.12, swarm mode is included in the Docker daemon natively, which upset some people due to bloat and scope creep. As a result, more people turned to CoreOS rkt as an alternative solution.

Starting with Docker 1.11, released in April 2016, Docker has changed the way it runs containers. The runtime now uses **containerd** and **runC** to run **Open Container Initiative (OCI)** images in containers:

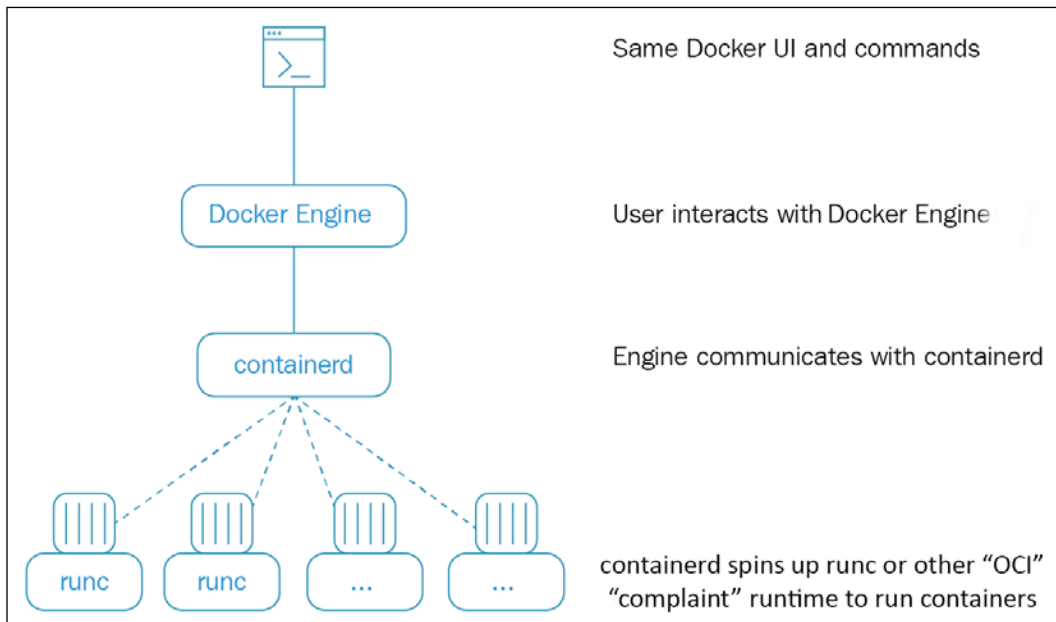


Figure 1.3: Architecture of Docker 1.11 after building it on runC and containerd

rkt

rkt is a container manager from CoreOS (the developers of the CoreOS Linux distro, etcd, flannel, and more). It is not developed anymore as CoreOS was acquired by Red Hat, who was later acquired by IBM. However, the legacy of rkt is the proliferation of multiple container runtimes beyond Docker and pushing Docker toward the standardized OCI effort.

The rkt runtime prides itself on its simplicity and a strong emphasis on security and isolation. It doesn't have a daemon like Docker Engine and relies on the OS init system, such as systemd, to launch the rkt executable. rkt can download images (both App Container (**appc**) images and OCI images), verify them, and run them in containers. Its architecture is much simpler.

App container

CoreOS started a standardization effort in December 2014 called appc. This includes a standard image format (**ACI – Application Container Image**), runtime, signing, and discovery. A few months later, Docker started its own standardization effort with OCI. At this point, it seems these efforts will converge. This is a great thing as tools, images, and runtime will be able to interoperate freely. We're not there yet.

CRI-O

CRI-O is a Kubernetes incubator project. It is designed to provide an integration path between Kubernetes and OCI-compliant container runtimes like Docker. CRI-O provides the following capabilities:

- Support for multiple image formats, including the existing Docker image format
- Support for multiple means to download images, including trust and image verification
- Container image management (managing image layers, overlay filesystems, and so on)
- Container process lifecycle management
- Monitoring and logging required to satisfy the CRI
- Resource isolation as required by the CRI

It supports runc and Kata containers right now, but any OCI-compliant container runtime can be plugged in and be integrated with Kubernetes.

Hyper containers

Hyper containers are another option. A Hyper container has a lightweight VM (its own guest kernel) and it can run on bare metal. Instead of relying on Linux cgroups for isolation, it relies on a hypervisor. This approach presents an interesting mix compared to standard bare-metal clusters, which are difficult to set up, and public clouds, where containers are deployed on heavyweight VMs.

Frakti

Frakti lets Kubernetes use hypervisors via the OCI-compliant runV project to run its pods and containers. It's a lightweight, portable, and secure approach that provides strong isolation with its own kernel compared to the traditional Linux namespace-based approaches, but not as heavyweight as a full-fledged VM.

Stackube

Stackube (previously called Hypernetes) is a multi-tenant distribution that uses Hyper containers as well as some OpenStack components for authentication, persistent storage, and networking. Since containers don't share the host kernel, it is safe to run containers of different tenants on the same physical host. Stackube uses Frakti, of course, as its container runtime.

In this section, we've covered the various runtime engines that Kubernetes supports as well as the trend toward standardization, convergence, and externalizing the runtime support from core Kubernetes. In the next section, we'll take a step back and look at the big picture, and how Kubernetes fits into the CI/CD pipeline.

Continuous integration and deployment

Kubernetes is a great platform for running your microservice-based applications. But, at the end of the day, it is an implementation detail. Users, and often most developers, may not be aware that the system is deployed on Kubernetes. But Kubernetes can change the game and make things that were too difficult before possible.

In this section, we'll explore the CI/CD pipeline and what Kubernetes brings to the table. At the end of this section, you'll be able to design CI/CD pipelines that take advantage of Kubernetes properties such as easy scaling and development-production parity to improve the productivity and robustness of day-to-day development and deployment.

What is a CI/CD pipeline?

A CI/CD pipeline is a set of tools and steps that takes a set of changes by developers or operators that modify the code, data, or configuration of a system, tests them, and deploys them to production (and possibly other environments). Some pipelines are fully automated and some are semi-automated with human checks. In large organizations, there may be test and staging environments that changes are deployed to automatically, but release to production requires manual intervention. The following diagram depicts a typical pipeline:

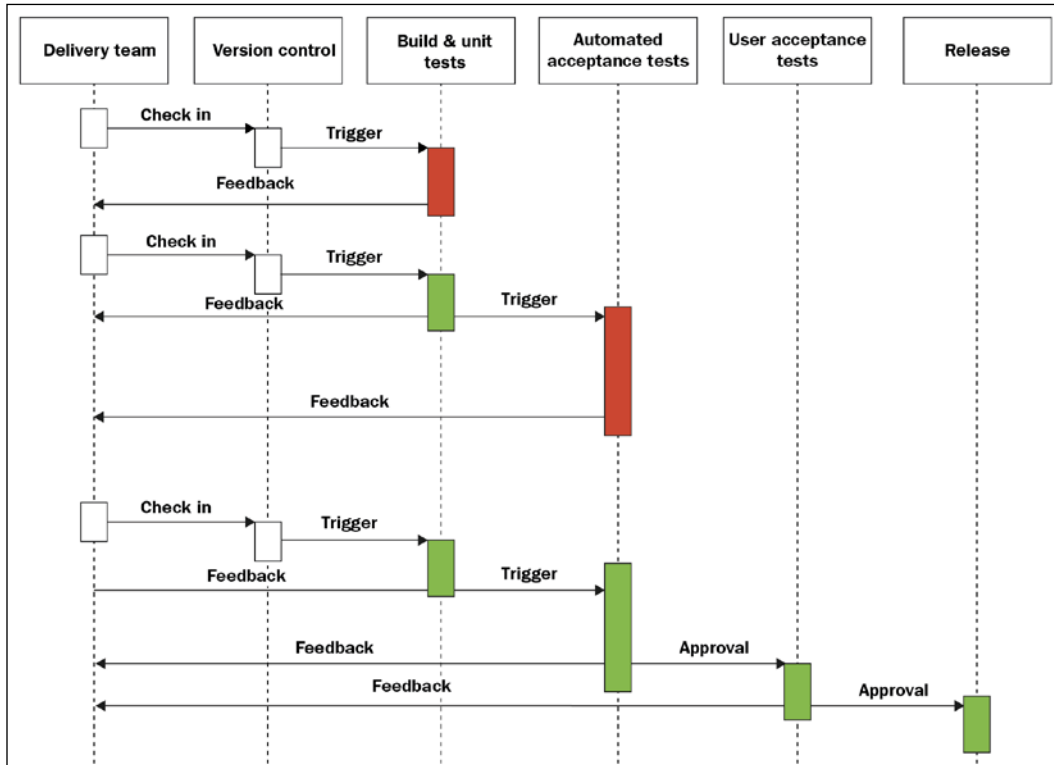


Figure 1.4: Diagram representing CI/CD pipeline

It may be worth mentioning that developers can be completely isolated from production infrastructure. Their interface is just a Git workflow, where a good example is Deis Workflow (PaaS on Kubernetes, similar to Heroku).

Designing a CI/CD pipeline for Kubernetes

When your deployment target is a Kubernetes cluster, you should rethink some traditional practices. For starters, the packaging is different. You need to bake images for your containers. Reverting code changes is super easy and instantaneous by using smart labeling. It gives you a lot of confidence that, if a bad change slips through the testing net somehow, you'll be able to revert to the previous version immediately. But you want to be careful there. Schema changes and data migrations can't be automatically rolled back.

Another unique capability of Kubernetes is that developers can run a whole cluster locally. That takes some work when you design your cluster, but since the microservices that comprise your system run in containers, and those containers interact via APIs, it is possible and practical to do. As always, if your system is very data-driven, you will need to accommodate that and provide data snapshots and synthetic data that your developers can use.

There are many commercial CI/CD solutions that support Kubernetes, but there are also several Kubernetes-native solutions, such as Tekton, Argo CD, and Jenkins X.

A Kubernetes-native CI/CD solution runs inside your cluster, is specified using Kubernetes CRDs, and uses containers to execute the steps. By using a Kubernetes-native CI/CD solution, you get to benefit from Kubernetes managing and easily scaling your CI/CD pipelines, which is otherwise often a non-trivial task.

Summary

In this chapter, we covered a lot of ground, and you got to understand the design and architecture of Kubernetes. Kubernetes is an orchestration platform for microservice-based applications running as containers. Kubernetes clusters have master and worker nodes. Containers run within pods. Each pod runs on a single physical or virtual machine. Kubernetes directly supports many concepts, such as services, labels, and persistent storage. You can implement various distributed system design patterns on Kubernetes. Container runtimes just need to implement the CRI. Docker, rkt, hyper containers, and more are supported.

In *Chapter 2, Creating Kubernetes Clusters*, we will explore the various ways to create Kubernetes clusters, discuss when to use different options, and build a multi-node cluster.

2

Creating Kubernetes Clusters

Overview

In the previous chapter, we learned what Kubernetes is all about, how it is designed, what concepts it supports, its runtime engines, and how it fits within the CI/CD pipeline.

Creating a Kubernetes cluster from scratch is a non-trivial task. There are many options and tools to select from. There are many factors to consider. In this chapter, we will roll our sleeves up and build us some Kubernetes clusters using Minikube, KinD, and K3d. We will discuss and evaluate other tools such as Kubeadm, Kubespray, KRIB, RKE, and bootkube. We will also look into deployment environments such as local, cloud, and bare metal. The topics we will cover are as follows:

- Creating a single-node cluster with Minikube
- Creating a multi-node cluster with KinD
- Creating a multi-node cluster using k3d
- Creating clusters in the cloud
- Creating bare-metal clusters from scratch
- Reviewing other options for creating Kubernetes clusters

At the end of this chapter, you will have a solid understanding of the various options to create Kubernetes clusters and knowledge of the best-of-breed tools to support the creation of Kubernetes clusters. You will have also built several clusters, both single-node and multi-node.

Creating a single-node cluster with Minikube

In this section, we will create a local single-node cluster using Minikube. Local clusters are the most useful for developers that want quick edit-test-deploy-debug cycles on their machine, before committing their changes. Local clusters are very useful for DevOps and operators that want to play with Kubernetes locally, without concerns about breaking a shared environment. While Kubernetes is typically deployed on Linux in production, many developers work on Windows PCs or Macs. That said, there are not too many differences if you do want to install Minikube on Linux:



Figure 2.1: The minikube logo

Meet kubectl

Before we start creating clusters, let us talk about kubectl. It is the official Kubernetes CLI and it interacts with your Kubernetes cluster's API server via its API. It is configured by default using the `~/.kube/config` file, which is a YAML file that contains metadata, connection info, and authentication tokens or certificates for one or more clusters. Kubectl provides commands you can use to view your configuration and switch between clusters if it contains more than one. You can also point kubectl at a different config file by setting the `KUBECONFIG` environment variable. I prefer a third approach, which is keeping separate config file for each cluster and copying the active cluster's config file to `~/.kube/config` (symlinks do not work).

We will discover together what kubectl can do along the way. The purpose here is just to avoid confusion when working with different clusters and configuration files.

Quick introduction to Minikube

Minikube is the most mature local Kubernetes cluster. It runs the latest stable Kubernetes release and it supports Windows, macOS, and Linux. It supports:

- LoadBalancer service type via Minikube tunnel
- NodePort service type via Minikube service
- Multiple clusters
- Filesystem mounts
- GPU support for machine learning
- RBAC
- Persistent volumes
- Ingress
- Dashboard via Minikube dashboard
- Custom container runtimes via the `start --container-runtime` flag
- Configuration API server and kubelet options via command-line flags
- Add-ons

Getting ready

There are some prerequisites to install before you can create the cluster itself. These include VirtualBox, the `kubectl` command-line interface to Kubernetes, and, of course, Minikube itself. Here is a list of the latest versions at the time of writing:

- VirtualBox: <https://www.virtualbox.org/wiki/Downloads>
- Kubectl: <https://kubernetes.io/docs/tasks/tools/install-kubectl/>
- Minikube: <https://kubernetes.io/docs/tasks/tools/install-minikube/>

On Windows

Install VirtualBox and make sure `kubectl` and Minikube are on your path. I personally just throw all command-line programs I use into `c:.` You may prefer another approach. I use the excellent ConEMU to manage multiple consoles, terminals, and SSH sessions. It works with Command Prompt, PowerShell, PuTTY, Cygwin, `msys`, and `Git-Bash`. It does not get much better than that on Windows.

With Windows 10 Pro, you have the option to use the Hyper-V hypervisor. This is technically a better solution than VirtualBox, but it requires the Pro version of Windows and is completely Windows-specific. By using VirtualBox, these instructions are universal and will be easy to adapt to other versions of Windows, or other operating systems altogether. If you have Hyper-V enabled, you must disable it because VirtualBox cannot coexist with Hyper-V.

I recommend using PowerShell in administrator mode. You can add the following alias and function to your PowerShell profile:

```
Set-Alias -Name k -Value kubectl
function mk
{
    minikube-windows-amd64 `
    --show-libmachine-logs `
    --also-log-to-stderr `
    @args
}
```

On macOS

On macOS, you have the option of using HyperKit instead of VirtualBox:

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/docker-
machine-driver-hyperkit \
&& chmod +x docker-machine-driver-hyperkit \
&& sudo mv docker-machine-driver-hyperkit /usr/local/bin/ \
&& sudo chown root:wheel /usr/local/bin/docker-machine-driver-hyperkit \
&& sudo chmod u+s /usr/local/bin/docker-machine-driver-hyperkit
```

You can add aliases to your `.bashrc` file (similar to the PowerShell alias and function on Windows):

```
alias k='kubectl'
alias mk='/usr/local/bin/minikube'
```

If you chose HyperKit instead of VirtualBox, you need to add the flag `--vm-driver=hyperkit` when starting the cluster.

It is also important to disable any VPN when using HyperKit.

Now, you can use `k` and `mk` and type less. The flags to Minikube in the `mk` function provide better logging and direct it to the console in addition to files (similar to `tee`).

Type `mk version` to verify Minikube is correctly installed and functioning:

```
$ mk version
minikube version: v1.10.1
```

Type `k version` to verify `kubectl` is correctly installed and functioning:

```
$ k version
Client Version: version.Info{Major:"1", Minor:"18", GitVersion:"v1.18.3",
GitCommit:"641856db18352033a0d96dbc99153fa3b27298e5", GitTreeState:"clean",
BuildDate:"2020-05-20T12:52:00Z", GoVersion:"go1.13.9", Compiler:"gc",
Platform:"darwin/amd64"}
The connection to the server localhost:8080 was refused – did you specify
the right host or port?
Unable to connect to the server: dial tcp 192.168.99.100:8443: getsockopt:
operation timed out
```

Do not worry about the error on the last line. There is no cluster running, so `kubectl` cannot connect to anything. That is expected.

You can explore the available commands and flags for both Minikube and `kubectl`. I will not go over each command, only the commands I use.





Creating the cluster

The Minikube tool supports multiple versions of Kubernetes. At the time of writing, the latest version is 1.18.0, which is also the default:

```
$ mk start
🐻 minikube v1.10.1 on darwin (amd64)
🔥 Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
🐳 Configuring environment for Kubernetes v1.18.0 on Docker 19.03.8
🚗 Pulling images ...
🚀 Launching Kubernetes ...
🕒 Verifying: apiserver proxy etcd scheduler controller dns
🎉 Done! kubectl is now configured to use "minikube"
```

When you restart an existing stopped cluster, you will see the following output:

```
$ mk start
🐻 minikube v1.10.1 on darwin (amd64)
💡 Tip: Use 'minikube start -p <name>' to create a new cluster, or
'minikube delete' to delete this one.
🔄 Restarting existing virtualbox VM for "minikube" ...
🕒 Waiting for SSH access ...
```

-  **Configuring environment for Kubernetes v1.18.0 on Docker 19.03.8**
-  **Relaunching Kubernetes v1.18.0 using kubeadm ...**
-  **Verifying: apiserver proxy etcd scheduler controller dns**
-  **Done! kubectl is now configured to use "minikube"**

Let us review what Minikube did behind the curtains for you. You will need to do a lot of it when creating a cluster from scratch:

1. Start a VirtualBox VM
2. Create certificates for the local machine and the VM
3. Download images
4. Set up networking between the local machine and the VM
5. Run the local Kubernetes cluster on the VM
6. Configure the cluster
7. Start all the Kubernetes control plane components
8. Configure kubectl to talk to the cluster

Troubleshooting

If something goes wrong during the process, try to follow the error messages. You can add the `--alsologtostderr` flag to get detailed error info to the console. Everything Minikube does is organized neatly under `~/minikube`. Here is the directory structure:

```
$ tree ~/.minikube -L 2
/Users/gigi.sayfan/.minikube
├── addons
├── apiserver.crt
├── apiserver.key
├── ca.crt
├── ca.key
├── ca.pem
├── cache
│   ├── images
│   ├── iso
│   └── v1.15.0
├── cert.pem
└── certs
    ├── ca-key.pem
    ├── ca.pem
    ├── cert.pem
    └── key.pem
```


Disconnect from the VM via *Ctrl + D* or by typing:

```
$ logout
```

We will cover many of the `kubectl` commands throughout our journey. First, let us check the cluster status using `cluster-info`:

```
$ k cluster-info
```

```
Kubernetes master is running at https://192.168.99.103:8443
```

```
KubeDNS is running at https://192.168.99.103:8443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
```

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

You can see that the master is running properly. To see a much more detailed view of all the objects in the cluster as JSON, type `k cluster-info dump`. The output can be a little daunting, so let us use more specific commands to explore the cluster.

Let us check out the nodes in the cluster using `get nodes`:

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
minikube	Ready	master	28m	v1.16.3

So, we have one node called `minikube`. To get a lot more information about it, type `k describe node minikube`.

The output is verbose; I will let you try it yourself.

Doing work

We have a nice empty cluster up and running (well, not completely empty as the DNS service and dashboard run as pods in the `kube-system` namespace). It is time to deploy some pods:

```
$ k create deployment echo --image=gcr.io/google_containers/echoserver:1.8  
deployment.apps/echo created
```

Let us check out the pod that was created:

```
$ k get pods
```

NAME	READY	STATUS	RESTARTS	AGE
echo-855975f9c-r6kj8	1/1	Running	0	2m11s

To expose our pod as a service, type the following:

```
$ k expose deployment echo --type=NodePort --port=8080
service/echo exposed
```

Exposing the service as type NodePort means that it is exposed to the host on some port. But it is not the 8080 port we ran the pod on. Ports get mapped in the cluster. To access the service, we need the cluster IP and the exposed port:

```
$ mk ip
192.168.99.103
```

```
$ k get service echo --output='jsonpath='{.spec.ports[0].nodePort}''
31800
```

Now, we can access the echo service, which returns a lot of information.

Replace the IP address and port with the results of the previous commands:

```
$ curl http://192.168.99.103:31800/hi
  Hostname: echo-855975f9c-r6kj8

Pod Information:
  -no pod information available-

Server values:
  server_version=nginx: 1.13.3 - lua: 10008

Request Information:
```



```
client_address=172.17.0.1
method=GET
real_path=/hi
query=
request_version=1.1
request_uri=http://192.168.99.103:8080/hi
```

Request Headers:

```
accept=/*/*
host=192.168.99.103:31800
user-agent=curl/7.64.0
```

Request Body:

```
-no body in request-
```

Congratulations! You just created a local Kubernetes cluster, deployed a service, and exposed it to the world.

Examining the cluster with the dashboard

Kubernetes has a very nice web interface, which is deployed, of course, as a service in a pod. The dashboard is well designed and provides a high-level overview of your cluster. It also lets you drill down into individual resources, view logs, edit resource files, and more. It is the perfect weapon when you want to check out your cluster manually. To launch it, type:

```
$ mk dashboard
```

```
🔧 Enabling dashboard ...
😬 Verifying dashboard health ...
🚀 Launching proxy ...
😬 Verifying proxy health ...
🍦 Opening http://127.0.0.1:56853/api/v1/namespaces/kube-system/services/
http:kubernetes-dashboard:/proxy/ in your default browser...
```

Minikube will open a browser window with the dashboard UI. Note that, on Windows, Microsoft Edge cannot display the dashboard. I had to run it myself on a different browser.

Here is the workloads view, which displays deployments, replica sets, replication controllers, and pods:

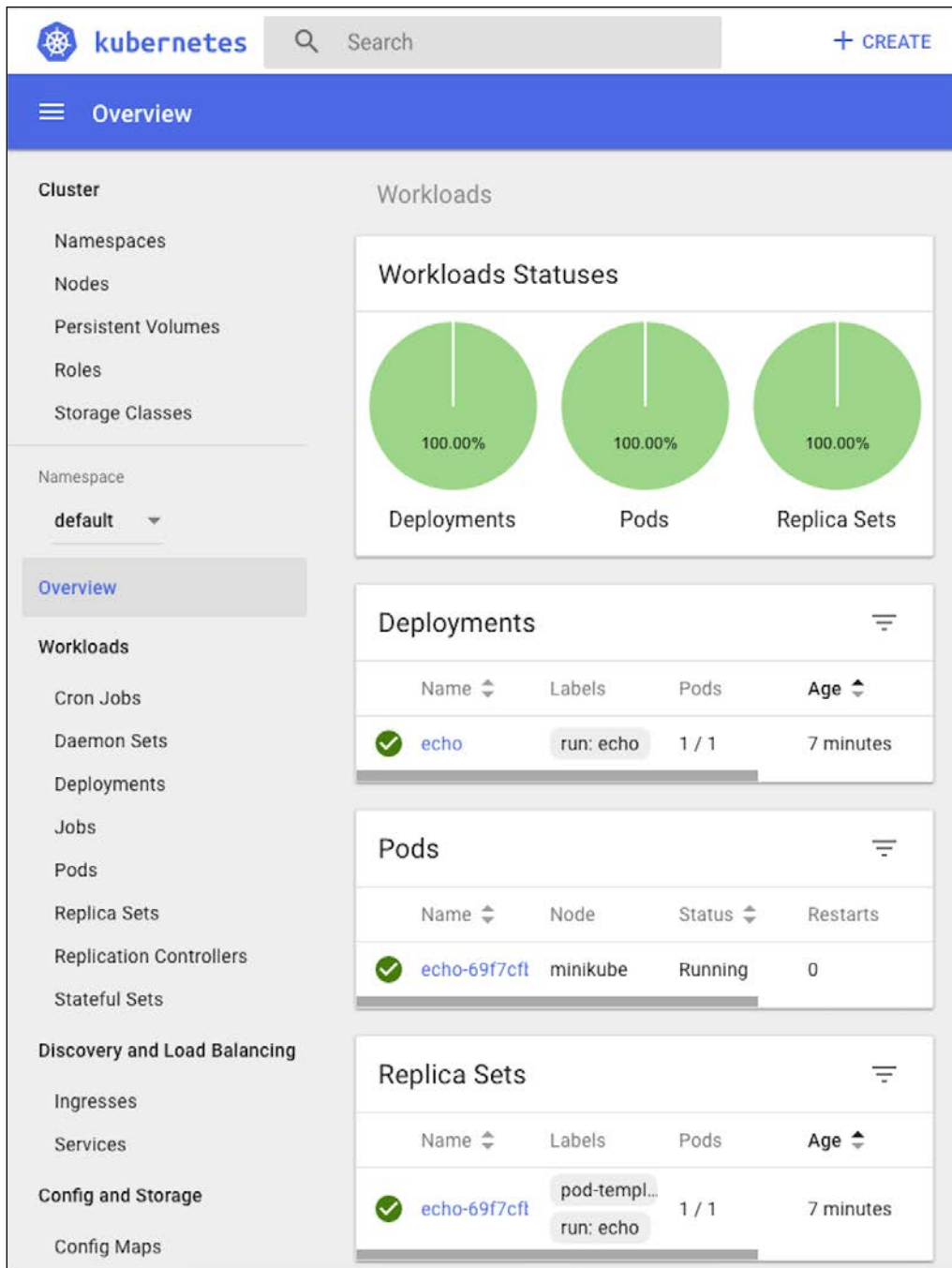


Figure 2.2: Kubernetes dashboard UI

It can also display daemon sets, stateful sets, and jobs, but we do not have any in this cluster.

In this section, we created a local single-node Kubernetes cluster on Windows, explored it a little bit using kubectl, deployed a service, and played with the web UI. In the next section, we will move on to a multi-node cluster.

Creating a multi-node cluster with KinD

In this section, we will create a multi-node cluster using KinD. We will also repeat the deployment of the echo server we deployed on Minikube and observe the differences. Spoiler alert – everything will be faster and easier!

Quick introduction to KinD

KinD stands for Kubernetes in Docker. It is a tool for creating ephemeral clusters (no persistent storage). It was built primarily for running the Kubernetes conformance tests. It supports Kubernetes 1.11+. Under the cover, it uses kubeadm to bootstrap Docker containers as nodes in the cluster. KinD is a combination of a library and a CLI. You can use the library in your code for testing or other purposes. KinD can create highly available clusters with multiple master nodes. Finally, KinD is a CNCF conformant Kubernetes installer. It better be if it is used for the conformance tests of Kubernetes itself :-).

KinD is super fast to start, but it has some limitations too: no persistent storage and no support for alternative runtimes yet, only Docker.

Let's install KinD and get going.

Installing KinD

You must have Docker installed as KinD is literally running as a Docker container. If you have Go 1.11+ installed, you can install the KinD CLI via:

```
$ GO111MODULE="on" go get sigs.k8s.io/kind@v0.8.1
```

Otherwise, on macOS, type:

```
$ curl -Lo ./kind-darwin-amd64 https://github.com/kubernetes-sigs/kind/releases/download/v0.8.1/kind-darwin-amd64
$ chmod +x ./kind-darwin-amd64
$ mv ./kind-darwin-amd64 /usr/local/bin/kind
```

On Windows, type (in PowerShell):

```
c:\> curl.exe -Lo kind-windows-amd64.exe https://github.com/kubernetes-
sigs/kind/releases/download/v0.8.1/kind-windows-amd64
c:\> Move-Item .\kind-windows-amd64.exe c:\windows\kind.exe
```

Creating the cluster with KinD

Creating a cluster is super easy:

```
$ kind create cluster
```

```
Creating cluster "kind" ...
```

- ✓ Ensuring node image (kindest/node:v1.16.3) 
- ✓ Preparing nodes 
- ✓ Creating kubeadm config 
- ✓ Starting control-plane 

Cluster creation complete. You can now use the cluster with:

```
export KUBECONFIG="$(kind get kubeconfig-path --name="kind")"
kubectl cluster-info
```

KinD suggests that you export KUBECONFIG, but as I mentioned earlier, I prefer to copy the config file to `~/.kube/config` so I do not have to export again if I want to access the cluster from another terminal window:

```
$ cp $(kind get kubeconfig-path --name="kind") ~/.kube/config
```

Now, we can access the cluster using `kubectl`:

```
$ k cluster-info
```

```
Kubernetes master is running at https://localhost:58560
```

```
KubeDNS is running at https://localhost:58560/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
```

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

However, this creates a single-node cluster:

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	Ready	master	11m	v1.16.3

Let us delete it and create a multi-node cluster:

```
$ kind delete cluster
```

```
Deleting cluster "kind" ...
```

To create a multi-node cluster, we need to provide a configuration file with the specification of our nodes. Here is a configuration file that will create a cluster with one control-plane node and two worker nodes:

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: worker
- role: worker
```

Let us save the configuration file as `kind-multi-node-config.yaml` and create the cluster:

```
$ kind create cluster --config kind-multi-node-config.yaml
```

Creating cluster "kind" ...

- ✓ Ensuring node image (kindest/node:v1.16.3) 
- ✓ Preparing nodes   
- ✓ Creating kubeadm config 
- ✓ Starting control-plane 
- ✓ Joining worker nodes 

Cluster creation complete. You can now use the cluster with:

```
export KUBECONFIG="$(kind get kubeconfig-path --name="kind")"
kubectl cluster-info
```

Yeah, it works! We have a local three-node cluster now:

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	Ready	master	12m	v1.16.3
kind-worker	NotReady	<none>	11m	v1.16.3
kind-worker2	NotReady	<none>	11m	v1.16.3

KinD is also kind enough (see what I did there) to let us create **highly available (HA)** clusters with multiple control plane nodes for redundancy. Let us give it a try and see what it looks like with two control-plane nodes and two worker nodes:

```
kind: Cluster
apiVersion: kind.sigs.k8s.io/v1alpha3
nodes:
- role: control-plane
- role: control-plane
- role: worker
```

```
- role: worker
```

Let us save the configuration file as `kind-ha-multi-node-config.yaml`, delete the current cluster, and create a new HA cluster:

```
$ kind delete cluster
Deleting cluster "kind" ...
$ kind create cluster --config kind-ha-multi-node-config.yaml
Creating cluster "kind" ...
  ✓ Ensuring node image (kindest/node:v1.16.3) 🖼️
  ✓ Preparing nodes 📦 📦 📦 📦 📦 📦
  ✓ Starting the external load balancer 🏗️
  ✓ Creating kubeadm config 📄
  ✓ Starting control-plane 🎮
  ✓ Joining more control-plane nodes 🚧
  ✓ Joining worker nodes 🚗
Cluster creation complete. You can now use the cluster with:
export KUBECONFIG="$(kind get kubeconfig-path --name="kind")"
kubectl cluster-info
```

Hmmm... there is something new here. Now, KinD creates an external load balancer, as well as join more control-plane nodes before joining the worker nodes. The load balancer is necessary to distribute requests across all the control-plane nodes.

Note that the external load balancer does not show as a node using `kubectl`:

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	Ready	master	8m31s	v1.16.3
kind-control-plane2	Ready	master	8m14s	v1.16.3
kind-worker	Ready	<none>	7m35s	v1.16.3
kind-worker2	Ready	<none>	7m35s	v1.16.3

However, KinD has its own `get nodes` command, where you can see the load balancer:

```
$ kind get nodes
kind-control-plane2
kind-worker
kind-control-plane
kind-worker2
kind-external-load-balancer
```

Doing work with KinD

Let us deploy our echo service on the KinD cluster. It starts the same:

```
$ k create deployment echo --image=gcr.io/google_containers/echoserver:1.8
deployment.apps/echo created
```

```
$ k expose deployment echo --type=NodePort --port=8080
service/echo exposed
```

Checking our services, we can see the echo service front and center:

```
$ k get svc echo
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
echo	NodePort	10.105.48.21	<none>	8080:31550/TCP	3m5s

However, there is no external IP to the service. With Minikube, we got the IP of the Minikube node itself via `$(minikube ip)`, and we can use it in combination with the node port to access the service. That is not an option with KinD clusters. Let us see how to use a proxy to access the echo service.

Accessing Kubernetes services locally though a proxy

In this section, we will go into a lot of detail about networking, services, and how to expose them outside the cluster.

Here, I am just showing how to get it done and keeping you in suspense for now. First, we need to run the `kubectl proxy` command, which exposes the API server, pods, and services on localhost:

```
$ k proxy &
[1] 10653
Starting to serve on 127.0.0.1:8001
```

Then, we can access the echo service though a specially crafted proxy URL that includes the exposed port (8080) and NOT the node port.

I use **Httpie** here. You can use **curl** too. To install Httpie, follow the instructions here: <https://httpie.org/doc#installation>:

```
$ http http://localhost:8001/api/v1/namespaces/default/services/echo:8080/
proxy/
HTTP/1.1 200 OK
```

Content-Length: 534
Content-Type: text/plain
Date: Thu, 28 May 2020 21:27:56 GMT
Server: echoserver

Hostname: echo-74545d499-wqkn9

Pod Information:
-no pod information available-

Server values:
server_version=nginx: 1.13.3 - lua: 10008

Request Information:
client_address=10.40.0.0
method=GET
real path=/
query=
request_version=1.1
request_uri=http://localhost:8080/

Request Headers:
accept=/*/*
accept-encoding=gzip, deflate
host=localhost:8001
user-agent=HTTPIe/0.9.9
x-forwarded-for=127.0.0.1, 172.17.0.1
x-forwarded-uri=/api/v1/namespaces/default/services/echo:8080/proxy/

Request Body:
-no body in request-

We will deep dive into exactly what is going on in a future chapter (*Chapter 12, Serverless Computing on Kubernetes*). Let us check out my favorite local cluster solution: k3d.

Creating a multi-node cluster with k3d

In this section, we'll create a multi-node cluster using k3d from Rancher. We will not repeat the deployment of the echo server because it is identical to the KinD cluster, including accessing it though a proxy. Spoiler alert – it is even faster and more user-friendly than KinD!

Quick introduction to k3s and k3d

Rancher created k3s, which is a lightweight Kubernetes distribution. Rancher says that k3s is five less than k8s, if that makes any sense. The basic idea is to remove features and capabilities that most people don't need, such as:

- Non-default features
- Legacy features
- Alpha features
- In-tree storage drivers
- In-tree cloud providers

However, the big ticket item is that k3s removed Docker and uses containerd instead. You can still bring Docker back if you depend on it.

Another major change is that k3s stores its state in a SQLite DB instead of etcd.

For networking and DNS, k3s uses Flannel and CoreDNS.

k3s also added a simplified installer that takes care of SSL and certificate provisioning.

The end result is astonishing – a single binary (less than 40 MB) that needs only 512 MB of memory.

Unlike Minikube and KinD, k3s is actually designed for production. The primary use case is for edge computing, IoT, and CI systems. It is optimized for ARM devices.

OK. That's k3s, but what's k3d? k3d takes all the goodness that is k3s, packages it in Docker (similar to KinD), and adds a friendly CLI to manage it.

Installing k3d

Installing k3d is as simple as:

```
$ curl -s https://raw.githubusercontent.com/rancher/k3d/master/install.sh |  
bash
```

The usual disclaimer is in effect – make sure to read the installation script before downloading and piping it to bash.

Creating the cluster with k3d

Are you ready to be amazed? Creating a single-node cluster with k3d takes less than 2 seconds!

```
$ time k3d create --workers 1
2020/05/28 17:07:36 Created cluster network with ID
f09fde83314b059d1a442ec1d01fcd62e522e5f1d838121528c5a1ae582e3cbf
2020/05/28 17:07:36 Creating cluster [k3s-default]
2020/05/28 17:07:36 Creating server using docker.io/rancher/k3s:v1.17.3-
k3s1...
2020/05/28 17:07:36 Booting 1 workers for cluster k3s-default
2020/05/28 17:07:37 Created worker with ID
8a6bd47f7a5abfbac5c396c45f13db04c7e18749ff4d2e054e737fe7f7843010
2020/05/28 17:07:37 SUCCESS: created cluster [k3s-default]
2020/05/28 17:07:37 You can now use the cluster with:
```

```
export KUBECONFIG="$(k3d get-kubeconfig --name='k3s-default')"
kubectl cluster-info
```

```
real    0m1.896s
user    0m0.009s
sys     0m0.011s
```

What about a multi-node cluster? We saw that KinD was much slower, especially when creating a HA cluster with multiple control-plane nodes and an external load balancer.

Let's delete the single-node cluster first:

```
$ k3d delete
2020/05/28 17:08:42 Removing cluster [k3s-default]
2020/05/28 17:08:42 ...Removing 1 workers
2020/05/28 17:08:43 ...Removing server
2020/05/28 17:08:45 SUCCESS: removed cluster [k3s-default]
```

Now, let's create a cluster with three worker nodes. That takes a little over 5 seconds:

```
$ time k3d create --workers 3
2020/05/28 17:09:16 Created cluster network with ID
5cd1e01434edb1facdab28e563b78b605af416e2ad062dc121400c3f8a5d166c
2020/05/28 17:09:16 Creating cluster [k3s-default]
2020/05/28 17:09:16 Creating server using docker.io/rancher/k3s:v1.17.3-
k3s1...
2020/05/28 17:09:17 Booting 3 workers for cluster k3s-default
```

```
2020/05/28 17:09:19 Created worker with ID
4b442116f8df7debecc9d70cee8ae8fb8f16783c0a8f111268be531f71dd54fa
2020/05/28 17:09:20 Created worker with ID
369879f1a38d60935908705f56b34a95caf6a44970beeb509c0cfb2047cd503a
2020/05/28 17:09:20 Created worker with ID
d531937996fd25490276e32150b69aa2356c90cfcdb480ab77ec3d2be08a2f6
2020/05/28 17:09:20 SUCCESS: created cluster [k3s-default]
2020/05/28 17:09:20 You can now use the cluster with:
```

```
export KUBECONFIG="$(k3d get-kubeconfig --name='k3s-default')"
kubectl cluster-info
```

```
real    0m5.164s
user    0m0.011s
sys     0m0.019s
```

Let's verify the cluster works as expected:

```
$ export KUBECONFIG="$(k3d get-kubeconfig --name='k3s-default')"
$ kubectl cluster-info
Kubernetes master is running at https://localhost:6443
CoreDNS is running at https://localhost:6443/api/v1/namespaces/kube-system/
services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info
dump'.
```

Here are the nodes. Note that there is just one master called k3d-k3s-default-server:

```
$ k get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
k3d-k3s-default-server	Ready	<none>	14h	v1.17.3-k3s1
k3d-k3s-default-worker-0	Ready	<none>	14h	v1.17.3-k3s1
k3d-k3s-default-worker-1	Ready	<none>	14h	v1.17.3-k3s1
k3d-k3s-default-worker-2	Ready	<none>	14h	v1.17.3-k3s1

You can stop and start clusters, create multiple clusters, and list existing clusters using the k3d CLI. Here are all the commands. Feel free to explore them further:

```
$ k3d
NAME:
  k3d - Run k3s in Docker!
```

```
USAGE:
  k3d [global options] command [command options] [arguments...]
```

VERSION:**v1.7.0****AUTHORS:****Thorsten Klein iwilltry42@gmail.com****Rishabh Gupta r.g.gupta@outlook.com****Darren Shepherd****COMMANDS:**

check-tools, ct	Check if docker is running
shell	Start a subshell for a cluster
create, c	Create a single- or multi-node k3s cluster in docker containers
delete, d, del	Delete cluster
stop	Stop cluster
start	Start a stopped cluster
list, ls, l	List all clusters
get-kubeconfig	Get kubeconfig location for cluster
help, h	Shows a list of commands or help for one command

GLOBAL OPTIONS:

--verbose	Enable verbose output
--help, -h	show help
--version, -v	print the version

You can repeat the steps for deploying, exposing, and accessing the echo service on your own. It works just like KinD.

OK. We created clusters using Minikube, KinD, and k3d. Let's compare them so that you can decide which one works for you.

Comparing Minikube, KinD, and k3d

Minikube is the official local Kubernetes release. It is part of Kubernetes; it's very mature and very full featured. That said, it requires a VM and is both slow to install and to start. It can also get into trouble with networking at arbitrary times and sometimes the only remedy is deleting the cluster and rebooting. Also, Minikube supports a single node only. I suggest using Minikube only if it supports some feature that you need that is not available in either KinD or k3d.

KinD is much faster than Minikube and is used for Kubernetes conformance tests, so by definition, it is a conformant Kubernetes distribution. It is the only local cluster solution that provides HA clusters with multiple control-plane nodes. It is also designed to be used as a library, which I don't find as a big attraction because it is very easy to automate CLIs from code. The main downside of KinD for local development is that it is ephemeral. I recommend using KinD if you contribute to Kubernetes itself and want to test against it.

k3d is the clear winner for me. It's lightning fast, supports multiple clusters, and supports multiple worker nodes per cluster. It's also easy to stop and start clusters without losing state.

Alright. Let's take a look at the cloud.

Creating clusters in the cloud (GCP, AWS, Azure)

Creating clusters locally is fun. It's also important during development and when trying to troubleshoot problems locally. But, in the end, Kubernetes is designed for cloud-native applications (applications that run in the cloud). Kubernetes doesn't want to be aware of individual cloud environments because that doesn't scale. Instead, Kubernetes has the concept of a cloud-provider interface. Every cloud provider can implement this interface and then host Kubernetes. Note that, as of version 1.5, Kubernetes still maintains implementations for many cloud providers in its tree, but in the future, they will be refactored out.

The cloud-provider interface

The cloud-provider interface is a collection of Go data types and interfaces. It is defined in a file called `cloud.go`, available at <https://github.com/kubernetes/cloud-provider/blob/master/cloud.go>.

Here is the main interface:

```
type Interface interface {
    Initialize(clientBuilder controller.ControllerClientBuilder)
    LoadBalancer() (LoadBalancer, bool)
    Instances() (Instances, bool)
    Zones() (Zones, bool)
    Clusters() (Clusters, bool)
    Routes() (Routes, bool)
    ProviderName() string
}
```

```
HasClusterID() bool
}
```

This is very clear. Kubernetes operates in terms of instances, zones, clusters, and routes, and also requires access to a load balancer and provider name. The main interface is primarily a gateway. Most methods return yet other interfaces.

For example, the `Clusters` interface is very simple:

```
type Clusters interface {
    ListClusters() ([]string, error)
    Master(clusterName string) (string, error)
}
```

The `ListClusters()` method returns cluster names. The `Master()` method returns the IP address or DNS name of the master node.

The other interfaces are not much more complicated. The entire file is 214 lines long (at the time of writing), including lots of comments. The take-home point is that it is not too complicated to implement a Kubernetes provider if your cloud utilizes those basic concepts.

GCP

The **Google Cloud Platform (GCP)** supports Kubernetes out of the box. The so-called **Google Kubernetes Engine (GKE)** is a container management solution built on Kubernetes. You don't need to install Kubernetes on GCP, and you can use the Google Cloud API to create Kubernetes clusters and provision them. The fact that Kubernetes is a built-in part of the GCP means it will always be well integrated and well tested, and you don't have to worry about changes in the underlying platform breaking the cloud-provider interface.

All in all, if you plan to base your system on Kubernetes and you don't have any existing code on other cloud platforms, then GCP is a solid choice. It leads the pack in terms of maturity, polish, and depth of integration to GCP services, and is usually the first to update to newer versions of Kubernetes.

AWS

AWS has its own container management service called **Elastic Container Service (ECS)** that is not based on Kubernetes. It also has a managed Kubernetes service called **Elastic Kubernetes Service (EKS)**. However, you can run Kubernetes yourself on AWS EC2 instances.

In fact, most of the production Kubernetes deployments in the world run on AWS EC2. Let's talk about how to roll your own Kubernetes first and then we'll discuss EKS.

Kubernetes on EC2

AWS was a supported cloud provider from the get-go. There is a lot of documentation on how to set it up. While you could provision some EC2 instances yourself and use `kubeadm` to create a cluster, I recommend using the Kops (Kubernetes Operations) project. Kops is a Kubernetes project available on GitHub: <https://github.com/kubernetes/kops/blob/master/docs/aws.md>. It is not part of the core Kubernetes repository, but it is developed and maintained by the Kubernetes developers.

It supports the following features:

- Automated Kubernetes cluster CRUD for the cloud (AWS).
- Highly available Kubernetes clusters.
- Uses a state-sync model for dry-run and automatic idempotency.
- Custom support for `kubectl` add-ons.
- Kops can generate Terraform configuration.
- Based on a simple meta-model defined in a directory tree.
- Easy command-line syntax.
- Community support.

To create a cluster, you need to do some minimal DNS configuration via `route53`, set up a S3 bucket to store the cluster configuration, and then run a single command:

```
kops create cluster --cloud=aws --zones=us-east-1c ${NAME}
```

The complete instructions can be found here: https://github.com/kubernetes/kops/blob/master/docs/getting_started/aws.md.

At the end of 2017, AWS joined the CNCF and made two big announcements regarding Kubernetes: its own Kubernetes-based container orchestration solution (EKS) and a container-on-demand solution (Fargate).

AWS EKS

AWS EKS is a fully managed and highly available Kubernetes solution. It has three masters running in three AZs. EKS also takes care of upgrades and patching. The great thing about EKS is that it runs a stock Kubernetes. This means you can use all the standard plugins and tools developed by the community. It also opens the door to convenient cluster federation with other cloud providers and/or your own on-premise Kubernetes clusters.

EKS provides deep integration with AWS infrastructure like how IAM authentication is integrated with Kubernetes **role-based access control (RBAC)**.

You can also use **AWS PrivateLink** if you want to access your Kubernetes masters directly from your own **Amazon Virtual Private Cloud (Amazon VPC)**. With PrivateLink, your Kubernetes masters and the Amazon EKS service endpoint appear as an elastic network interface with private IP addresses in your Amazon VPC.

Another important piece of the puzzle is a special CNI plugin that lets your Kubernetes components talk to each other using AWS networking.

EKS keeps getting better and Amazon demonstrated that it is committed to keeping it up to date and improving it. If you are an AWS shop and getting into Kubernetes, I recommend starting with EKS as opposed to building your own cluster.

The `eksctl` tool is a great CLI for creating and managing EKS clusters and node groups. I successfully created, deleted, and added nodes to several Kubernetes clusters on AWS using `eksctl`. Check out <https://eksctl.io/>.

Fargate

AWS Fargate lets you run containers directly without worrying about provisioning hardware. It eliminates a huge part of the operational complexity at the cost of losing some control. When using Fargate, you package your application into a container, specify CPU and memory requirements, define networking and IAM policies, and you're off to the races. Fargate can run on top of ECS at the moment and EKS in the future. It is a very interesting member in the serverless camp, although it's not directly related to Kubernetes.

Azure

Azure used to have its own container management service. You could use the Mesos-based DC/OS or Docker Swarm to manage them. But you can also use Kubernetes, of course. You could also provision the cluster yourself (for example, using Azure's desired state configuration) and then create the Kubernetes cluster using kubeadm. Azure doesn't have a Kops equivalent, but the Kubespray project is a good option.

However, in the second half of 2017, Azure jumped on the Kubernetes bandwagon too and introduced the **Azure Kubernetes Service (AKS)**. It is similar to Amazon EKS, although it's a little further ahead in its implementation.

AKS provides a REST API as well as a CLI to manage your Kubernetes cluster. However, you can use kubectl and any other Kubernetes tooling directly.

Here are some of the benefits of using AKS:

- Automated Kubernetes version upgrades and patching
- Easy cluster scaling
- Self-healing hosted control plane (masters)
- Cost savings – pay only for running agent pool nodes

AKS also offers integration with **Azure Container Instances (ACI)**, which is similar to AWS Fargate. This means that not only the control plane of your Kubernetes cluster is managed, but also the worker nodes.

Another interesting feature of AKS is AKS-Engine: <https://github.com/Azure/aks-engine>. AKS-Engine is an open source project, which is the core of AKS. One of the downsides of using a managed service is that you have to accept the choices of the cloud provider. If you have special requirements, then the other option is to create your own cluster, which is a big undertaking. With AKS Engine, you get to take the work the AKS team did and customize just the parts that are important to you.

Other cloud providers

GCP, AWS, and Azure are leading the pack, but there are quite a few other companies that offer managed Kubernetes services. In general, I recommend using these providers if you already have significant business connections or integrations.

Once upon a time in China

If you operate in China with its special constraints and limitations, you should probably use a Chinese cloud platform. There are three big ones: Alibaba, Tencent, and Huawei.

The Chinese **Alibaba Cloud** is an up and comer on the cloud platform scene. It mimics AWS pretty closely, although its English documentation leaves a lot to be desired. I deployed some production application on Ali baba cloud, but not Kubernetes clusters. The Alibaba cloud supports Kubernetes in several ways via its **Alibaba container service for Kubernetes (ACK)**:

- Run your own dedicated Kubernetes cluster (you must create three master nodes and upgrade and maintain them)
- Use the managed Kubernetes cluster (you're just responsible for the worker nodes)
- Use the serverless Kubernetes cluster via **Elastic container instances (ECIs)**, which is like Fargate and ACI

ACK is a CNCF certified Kubernetes distribution. If you need to deploy cloud-native applications in China, then ACK looks like a solid option.

Tencent is another large Chinese company with its own cloud platform and Kubernetes support. **Tencent Kubernetes engine (TKE)** seems less mature than ACK.

Finally, the Huawei cloud platform offers the **Cloud Container Engine (CCE)**, which is built on Kubernetes. It supports VMs, bare metal, and GPU accelerated instances.

IBM Kubernetes Service

IBM is investing heavily in Kubernetes. It acquired RedHat at the end of 2018. RedHat was, of course, a major player in the Kubernetes world, building its OpenShift Kubernetes-based platform and contributing RBAC to Kubernetes. IBM has its own cloud platform and offers a managed Kubernetes cluster. You can try it out for free with \$200 credit.

IBM is also involved in the development of Istio and Knative, so IKS will likely have deep integration with those up and coming technologies.

Oracle Container Service

Oracle also has a cloud platform and, of course, offers a managed Kubernetes service too, with high availability, bare-metal instances, and multi-AZ support.

In this section, we covered the cloud-provider interface and looked at the recommended ways to create Kubernetes clusters on various cloud providers. The scene is still young, and the tools are evolving quickly. I believe convergence will happen soon. Tools and projects like Kargo and Kubernetes-anywhere have already been deprecated or merged into other projects. Kubeadm has matured and is the underlying foundation of many other tools to bootstrap and create Kubernetes clusters on and off the cloud. Now, let's consider what it takes to create bare-metal clusters where you have to provision the hardware and low-level networking too.

Creating a bare-metal cluster from scratch

In the previous section, we looked at running Kubernetes on cloud providers. This is the dominant deployment story for Kubernetes. But there are strong use cases for running Kubernetes on bare metal. I won't focus on hosted versus on-premises here. This is yet another dimension. If you already manage a lot of servers on-premises, you are in the best position to decide.

Use cases for bare metal

Bare-metal clusters are a bear, especially if you manage them yourself. There are companies that provide commercial support for bare-metal Kubernetes clusters, such as Platform 9, but the offerings are not mature yet. A solid open source option is Kubespray, which can deploy industrial-strength Kubernetes clusters on bare metal, AWS, GCE, Azure, and OpenStack.

Here are some use cases where it makes sense:

- **Price:** If you already manage large-scale bare clusters, it may be much cheaper to run Kubernetes clusters on your physical infrastructure.
- **Low network latency:** If you must have low latency between your nodes, then the VM overhead might be too much.
- **Regulatory requirements:** If you must comply with regulations, you may not be allowed to use cloud providers.
- **You want total control over hardware:** Cloud providers give you many options, but you may have special needs.

When should you consider creating a bare-metal cluster?

The complexities of creating a cluster from scratch are significant. A Kubernetes cluster is not a trivial beast. There is a lot of documentation on the web on how to set up bare-metal clusters, but as the whole ecosystem moves forward, many of these guides get out of date quickly.

You should consider going down this route if you have the operational capability to debug problems at every level of the stack. Most of the problems will probably be networking-related, but filesystems and storage drivers can bite you too, as well as general incompatibilities and version mismatches between components such as Kubernetes itself, Docker (or other runtimes, if you use them), images, your OS, your OS kernel, and the various add-ons and tools you use. If you opt for using VMs on top of bare metal, then you add another layer of complexity.

Understanding the process

There is a lot to do. Here is a list of some of the concerns you'll have to address:

- Implementing your own cloud-provider interface or sidestepping it
- Choosing a networking model and how to implement it (CNI plugin, direct compile)
- Whether or not to use network policies
- Select images for system components
- Security model and SSL certificates
- Admin credentials
- Templates for components such as API Server, replication controller, and scheduler
- Cluster services: DNS, logging, monitoring, and GUI

I recommend the following guide from the Kubernetes site to get a deeper understanding of what it takes to create a HA cluster from scratch using kubeadm: <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/>.

Using virtual private cloud infrastructure

If your use case falls under the bare-metal use cases, but you don't have the necessary skilled manpower or the inclination to deal with the infrastructure challenges of bare metal, you have the option to use a private cloud such as OpenStack with Stackube: <https://github.com/openstack/stackube>. If you want to aim a little higher in the abstraction ladder, then Mirantis offers a cloud platform built on top of OpenStack and Kubernetes.

Let's review a few more tools for building Kubernetes clusters on bare metal. Some of these tools support OpenStack as well.

Building your own cluster with Kubespray

Kubespray is a project for deploying production-ready highly available Kubernetes clusters. It uses Ansible and can deploy Kubernetes on a large number of targets, such as:

- AWS
- GCE
- Azure
- OpenStack
- vSphere
- Packet (bare metal)
- Oracle Cloud Infrastructure (experimental)

And also to plain bare metal.

It is highly customizable and support multiple operating systems for the nodes, multiple CNI plugins for networking, and multiple container runtimes.

If you want to test it locally, it can deploy to a multi-node vagrant setup too. If you're an Ansible fan, Kubespray may be a great choice for you.

Building your cluster with KRIB

KRIB is a Kubernetes installer for bare metal clusters that are provisioned using **Digital Rebar Provision (DRP)**. DRP is a single Golang executable that takes care of a lot of the heavy lifting like DHCP in terms of bare-metal provisioning (PXE/iPXE), and workflow automation. KRIB drives kubeadm to ensure it ends up with a valid Kubernetes cluster. The process involves:

- Server discovery
- Installation of the KRIE Content and Certificate Plugin
- Starting the cluster deployment
- Monitoring the deployment
- Accessing the cluster

See <https://kubernetes.io/docs/setup/production-environment/tools/krie/> for more details.

Building your cluster with RKE

Rancher Kubernetes Engine (RKE) is a friendly Kubernetes installer that can install Kubernetes on bare-metal as well as virtualized servers. RKE aims to address the complexity of installing Kubernetes. It is open source and has great documentation. Check it out here: <http://rancher.com/docs/rke/v0.1.x/en/>.

Bootkube

Bootkube is very interesting too. It can launch self-hosted Kubernetes clusters. Self-hosted means that most of the cluster components run as regular pods and can be managed, monitored, and upgraded using the same tools and processes you use for your containerized applications. There are significant benefits to this approach that simplify the development and operation of Kubernetes clusters.

It is a Kubernetes incubator project, but it doesn't seem very active. Check it out here: <https://github.com/kubernetes-incubator/bootkube>.

In this section, we considered the option to build a bare-metal cluster Kubernetes cluster. We looked into the use cases that require it and highlighted the challenges and difficulties.

Summary

In this chapter, we got into some hands-on cluster creation. We created a single-node cluster using Minikube and a multi-node cluster using KinD and k3d. Then, we looked at the many options to create Kubernetes clusters on cloud providers. Finally, we touched on the complexities of creating Kubernetes clusters on bare metal. The current state of affairs is very dynamic. The basic components are changing rapidly, the tooling is getting better, and there are different options for each environment. Kubeadm is now the cornerstone of most installation options, which is great for consistency and consolidation of effort.

It's still not completely trivial to stand up a Kubernetes cluster on your own, but with some effort and attention to detail, you can get it done quickly.

In the next chapter, we will explore the important topics of scalability and high availability. Once your cluster is up and running, you need to make sure it stays that way, even as the volume of requests increases. This requires ongoing attention and building the ability to recover from failures, as well as adjusting to changes in traffic.

References

- <https://github.com/kubernetes/minikube>
- <https://kind.sigs.k8s.io/>
- <https://k3s.io/>
- <https://github.com/rancher/k3d>
- <https://kubespray.io/#/>
- <https://www.alibabacloud.com/product/kubernetes>
- <https://www.ibm.com/cloud/container-service>

3

High Availability and Reliability

In *Chapter 2, Creating Kubernetes Clusters*, we learned how to create Kubernetes clusters in different environments, experimented with different tools, and created a couple of clusters. Creating a Kubernetes cluster is just the beginning of the story. Once the cluster is up and running, you need to make sure it stays operational.

In this chapter, we will dive into the topic of highly available clusters. This is a complicated topic. The Kubernetes project and the community haven't settled on one true way to achieve high availability nirvana. There are many aspects to highly available Kubernetes clusters, such as ensuring that the control plane can keep functioning in the face of failures, protecting the cluster state in etcd, protecting the system's data, and recovering capacity and/or performance quickly. Different systems will have different reliability and availability requirements. How to design and implement a highly available Kubernetes cluster will depend on those requirements.

At the end of this chapter, you will understand the various concepts associated with high availability and be familiar with Kubernetes high availability best practices and when to employ them. You will be able to upgrade live clusters using different strategies and techniques, and you will be able to choose between multiple possible solutions based on trade-offs between performance, cost, and availability.

High availability concepts

In this section, we will start our journey into high availability by exploring the concepts and building blocks of reliable and highly available systems. The million (trillion?) dollar question is, how do we build reliable and highly available systems from unreliable components? Components will fail; you can take that to the bank. Hardware will fail; networks will fail; configuration will be wrong; software will have bugs; people will make mistakes. Accepting that, we need to design a system that can be reliable and highly available even when components fail. The idea is to start with redundancy, detect component failure, and replace bad components quickly.

Redundancy

Redundancy is the foundation of reliable and highly available systems at the hardware and data levels. If a critical component fails and you want the system to keep running, you must have another identical component ready to go. Kubernetes itself takes care of your stateless pods via replication controllers and replica sets. But, your cluster state in etcd and the master components themselves need redundancy to function when some components fail. In addition, if your system's stateful components are not backed up by redundant storage (for example, on a cloud platform), then you need to add redundancy to prevent data loss.

Hot swapping

Hot swapping is the concept of replacing a failed component on the fly without taking the system down, with minimal (ideally, zero) interruption for users. If the component is stateless (or its state is stored in separate redundant storage), then hot swapping a new component to replace it is easy and just involves redirecting all clients to the new component. But, if it stores local state, including in memory, then hot swapping is not trivial. There are two main options:

- Give up on in-flight transactions
- Keep a hot replica in sync

The first solution is much simpler. Most systems are resilient enough to cope with failures. Clients can retry failed requests and the hot-swapped component will service them.

The second solution is more complicated and fragile, and will incur a performance overhead because every interaction must be replicated to both copies (and acknowledged). It may be necessary for some parts of the system.

Leader election

Leader or master election is a common pattern in distributed systems. You often have multiple identical components that collaborate and share the load, but one component is elected as the leader and certain operations are serialized through the leader. You can think of distributed systems with leader election as a combination of redundancy and hot swapping. The components are all redundant and, when the current leader fails or becomes unavailable, a new leader is elected and hot-swapped in.

Smart load balancing

Load balancing is about distributing the workload across multiple replicas that service incoming requests. This is useful for scaling up and down under heavy load by adjusting the number of replicas. When some replicas fail, the load balancer will stop sending requests to failed or unreachable components. Kubernetes will provision new replicas, restore capacity, and update the load balancer. Kubernetes provides great facilities to support this via services, endpoints, replica sets, labels, and ingress controllers.

Idempotency

Many types of failure can be temporary. This is most common with networking issues or with too-stringent timeouts. A component that doesn't respond to a health check will be considered unreachable and another component will take its place. Work that was scheduled for the presumably failed component may be sent to another component. But the original component may still be working and complete the same work. The end result is that the same work may be performed twice. It is very difficult to avoid this situation. To support exactly-once semantics, you need to pay a heavy price in overhead, performance, latency, and complexity. Thus, most systems opt to support at-least-once semantics, which means it is OK for the same work to be performed multiple times without violating the system's data integrity. This property is called idempotency. Idempotent systems maintain their state even if an operation is performed multiple times.

Self-healing

When component failures occur in dynamic systems, you usually want the system to be able to heal itself. Kubernetes replication controllers and replica sets are great examples of self-healing systems. But failure can extend well beyond pods. Self-healing starts with the automated detection of problems followed by an automated resolution. Quotas and limits help create checks and balances to ensure automated self-healing doesn't run amok due to unpredictable circumstances such as DDOS attacks. Self-healing systems deal very well with transient failures by retrying failed operations and escalating failures only when it's clear there is no other option. Some self-healing systems have fallback paths including serving cached content if up-to-date content is unavailable. Self-healing systems attempt to degrade gracefully and keep working until the core issue can be fixed.

In this section, we considered various concepts involved in creating reliable and highly available systems. In the next section, we will apply them and demonstrate best practices for systems deployed on Kubernetes clusters.

High availability best practices

Building reliable and highly available distributed systems is a non-trivial endeavor. In this section, we will check some of the best practices that enable a Kubernetes-based system to function reliably and be available in the face of various failure categories. We will also dive deep and see how to go about constructing your own highly available clusters.

Note that you should roll your own highly available Kubernetes cluster only in very special cases. Tools such as **Kubespray** provide battle-tested ways to create highly available clusters. You should take advantage of all the work and effort that went into these tools.

Creating highly available clusters

To create a highly available Kubernetes cluster, the master components must be redundant. That means etcd must be deployed as a cluster (typically across three or five nodes) and the Kubernetes API server must be redundant. Auxiliary cluster-management services such as **Heapster** storage may be deployed redundantly too, if necessary. The following diagram depicts a typical reliable and highly available Kubernetes cluster in a stacked etcd topology. There are several load-balanced master nodes, each one containing whole master components as well as an etcd component:

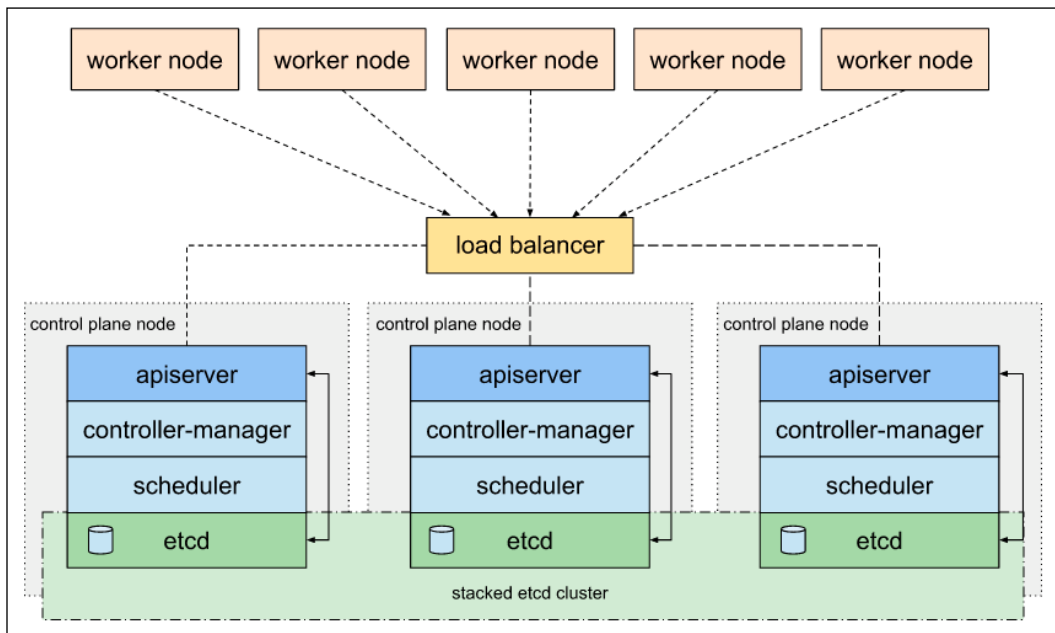


Figure 3.1: A highly available cluster configuration

This is not the only way to configure highly available clusters. You may prefer, for example, to deploy a standalone etcd cluster to optimize the machines to their workload or if you require more redundancy for your etcd cluster than the rest of the master nodes.

The following diagram shows a Kubernetes cluster where etcd is deployed as an external cluster:

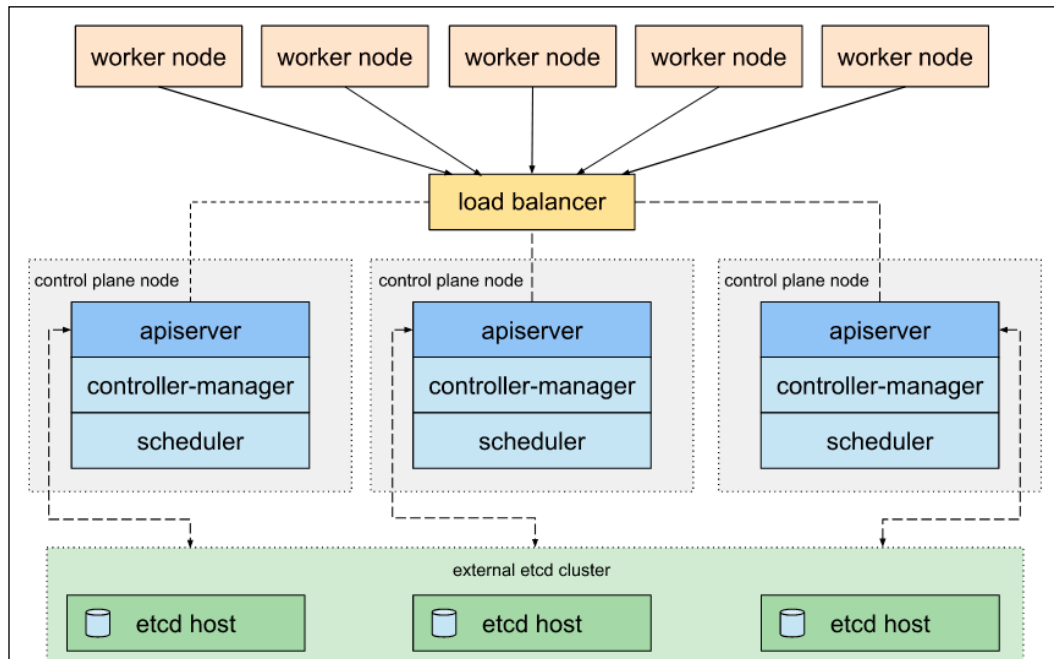


Figure 3.2: etcd used as an external cluster

Self-hosted Kubernetes clusters, where control plane components are deployed as pods and stateful sets in the cluster, are a great approach to simplify the robustness, disaster recovery, and self-healing of the control plane components by applying Kubernetes to Kubernetes.

Making your nodes reliable

Nodes will fail, or some components will fail, but many failures are transient. The basic guarantee is to make sure that the Docker daemon (or whatever the CRI implementation is) and the kubelet restart automatically in the event of a failure.

If you run CoreOS, a modern Debian-based OS (including Ubuntu ≥ 16.04), or any other OS that uses `systemd` as its `init` mechanism, then it's easy to deploy Docker and the kubelet as self-starting daemons:

```
systemctl enable docker
systemctl enable kublet
```

For other operating systems, the Kubernetes project selected `monit` for their high-availability example, but you can use any process monitor you prefer. The main requirement is to make sure that those two critical components will restart in the event of failure, without external intervention.

Protecting your cluster state

The Kubernetes cluster state is stored in `etcd`. The `etcd` cluster was designed to be super reliable and distributed across multiple nodes. It's important to take advantage of these capabilities for a reliable and highly available Kubernetes cluster.

Clustering `etcd`

You should have at least three nodes in your `etcd` cluster. If you need more reliability and redundancy, you can go for five, seven, or any other odd number of nodes. The number of nodes must be odd to have a clear majority in the event of a network split.

In order to create a cluster, the `etcd` nodes should be able to discover each other. There are several methods to accomplish this. I recommend using the excellent `etcd` operator from CoreOS:



Figure 3.3: The Kubernetes `etcd` operator logo

The operator takes care of many complicated aspects of `etcd` operation, such as the following:

- Create and destroy
- Resizing
- Failovers
- Rolling upgrades
- Backup and restore

Installing the etcd operator

The easiest way to install the etcd operator is using Helm – the Kubernetes package manager. If you don't have Helm installed yet, follow the instructions here: <https://github.com/kubernetes/helm#install>.

Next, save the following YAML to `helm-rbac.yaml`:

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: tiller
  namespace: kube-system
```

This creates a service account for Tiller and gives it a cluster admin role:

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

Then initialize Helm with the Tiller service account:

```
$ helm2 init --service-account tiller
$HELM_HOME has been configured at /Users/gigi.sayfan/.helm.
```

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.

To prevent this, run 'helm init' with the `--tiller-tls-verify` flag.
 For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation

Don't worry about the warnings at this point. We will dive deep into Helm in *Chapter 9, Packaging Applications*.

Now, we can finally install the etcd operator. I use x as a short release name to make the output less verbose. You may want to use more meaningful names:

```
$ helm2 install stable/etcd-operator --name x
```

```
NAME:      x
```

```
LAST DEPLOYED: Thu May 28 17:33:16 2020
```

```
NAMESPACE: default
```

```
STATUS: DEPLOYED
```

```
RESOURCES:
```

```
==> v1/Pod(related)
```

NAME	READY	STATUS
RESTARTS AGE		
x-etcd-operator-etcd-backup-operator-dffcbd97-hfsnc	0/1	Pending 0
0s		
x-etcd-operator-etcd-operator-669975754b-vhhq5	0/1	Pending 0
0s		
x-etcd-operator-etcd-restore-operator-6b787cc5c-6dk77	0/1	Pending 0
0s		

```
==> v1/Service
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
etcd-restore-operator	ClusterIP	10.43.182.231	<none>	19999/TCP 0s

```
==> v1/ServiceAccount
```

NAME	SECRETS	AGE
x-etcd-operator-etcd-backup-operator	1	0s
x-etcd-operator-etcd-operator	1	0s
x-etcd-operator-etcd-restore-operator	1	0s

```
==> v1beta1/ClusterRole
```

NAME	AGE
x-etcd-operator-etcd-operator	0s

```
==> v1beta1/ClusterRoleBinding
```

NAME	AGE
------	-----


```
x-etcd-operator-etcd-backup-operator 0s
x-etcd-operator-etcd-operator         0s
x-etcd-operator-etcd-restore-operator 0s
```

==> v1beta2/Deployment

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
x-etcd-operator-etcd-backup-operator	0/1	1	0	0s
x-etcd-operator-etcd-operator	0/1	1	0	0s
x-etcd-operator-etcd-restore-operator	0/1	1	0	0s

NOTES:

1. etcd-operator deployed.

If you would like to deploy an etcd-cluster set `cluster.enabled` to `true` in `values.yaml`

Check the etcd-operator logs

```
export POD=$(kubectl get pods -l app=x-etcd-operator-etcd-operator
--namespace default --output name)
kubectl logs $POD --namespace=default
```

Now that the operator is installed, we can use it to create the etcd cluster.

Creating the etcd Cluster

Save the following to `etcd-cluster.yaml`:

```
apiVersion: "etcd.database.coreos.com/v1beta2"
kind: "EtcdCluster"
metadata:
  name: "example-etcd-cluster"
spec:
  size: 3
  version: "3.2.13"
```

To create the cluster type, use the following command:

```
$ k create -f etcd-cluster.yaml
etcdcluster.etcd.database.coreos.com/etcd-cluster created
```

Let's verify the cluster pods were created properly:

```
$ k get pods -o wide | grep etcd-cluster
etcd-cluster-2fs2lpz7p7    1/1    Running    0    2m53s    10.42.2.4
k3d-k3s-default-worker-1
```

etcd-cluster-58547r5f6x k3d-k3s-default-worker-0	1/1	Running	0	3m49s	10.42.1.5
etcd-cluster-z7s4bfksdl k3d-k3s-default-worker-2	1/1	Running	0	117s	10.42.3.5

As you can see, each etcd pod was scheduled to run on a different node. This is exactly what we want with a redundant datastore like etcd.

The `-o wide` format for `kubectl's get` command provides additional information for the `get pods` command the node for the pod is scheduled on.

Verifying the etcd cluster

Once the etcd cluster is up and running, you can access it with the `etcdctl` tool to check on the cluster status and health. Kubernetes lets you execute commands directly inside pods or containers via the `exec` command (similar to `docker exec`).

Here is how to check if the cluster is healthy:

```
$ k exec etcd-cluster-2fs2lpz7p7 -- etcdctl cluster-health
```

```
member 1691519f36d795b7 is healthy: got healthy result from http://etcd-
cluster-2fs2lpz7p7.etcd-cluster.default.svc:2379
member 1b67c8cb37fca67e is healthy: got healthy result from http://etcd-
cluster-58547r5f6x.etcd-cluster.default.svc:2379
member 3d4cbb73aeb3a077 is healthy: got healthy result from http://etcd-
cluster-z7s4bfksdl.etcd-cluster.default.svc:2379
cluster is healthy
```

Here is to how to set and get key-value pairs:

```
$ k exec etcd-cluster-2fs2lpz7p7 -- etcdctl set test "Yeah, it works"
Yeah, it works
$ k exec etcd-cluster-2fs2lpz7p7 -- etcdctl get test
Yeah, it works
```

Protecting your data

Protecting the cluster state and configuration is great, but even more important is protecting your own data. If somehow the cluster state gets corrupted, you can always rebuild the cluster from scratch (although the cluster will not be available during the rebuild). But if your own data is corrupted or lost, you're in deep trouble. The same rules apply: redundancy is king. But while the Kubernetes cluster state is very dynamic, much of your data may be less dynamic.

For example, a lot of historic data is often important and can be backed up and restored. Live data might be lost, but the overall system may be restored to an earlier snapshot and suffer only temporary damage.

You should consider **Velero** as a solution for backing up your entire cluster, including your own data. Heptio (now part of VMware) developed Velero, which is open source and may be a life-saver for critical systems.

Check it out at <https://velero.io>.

Running redundant API servers

API servers are stateless, fetching all the necessary data on the fly from the etcd cluster. This means that you can easily run multiple API servers without needing to coordinate between them. Once you have multiple API servers running, you can put a load balancer in front of them to make it transparent to clients.

Running leader election with Kubernetes

Some master components, such as the scheduler and the controller manager, can't have multiple instances active at the same time. This would be chaos, as multiple schedulers would try to schedule the same pod into multiple nodes or multiple times into the same node. The correct way to have a highly scalable Kubernetes cluster is to have these components run in leader election mode. This means that multiple instances are running, but only one is active at a time and if it fails, another one is elected as leader and takes its place.

Kubernetes supports this mode via the `--leader-elect` flag. The scheduler and the controller manager can be deployed as pods by copying their respective manifests to `/etc/kubernetes/manifests`.

Here is a snippet from a scheduler manifest that shows the use of the flag:

```
command:
- /bin/sh
- -c
- /usr/local/bin/kube-scheduler --master=127.0.0.1:8080 --v=2
--leader-elect=true 1>>/var/log/kube-scheduler.log
2>&1
```

Here is a snippet from a controller manager manifest that shows the use of the flag:

```
- command:
- /bin/sh
```

```

- -C
- /usr/local/bin/kube-controller-manager --master=127.0.0.1:8080
--cluster-name=e2e-test-bburns
  --cluster-cidr=10.245.0.0/16 --allocate-node-cidrs=true --cloud-
provider=gce --service-account-private-key-file=/srv/kubernetes/server.
key
  --v=2 --leader-elect=true 1>>/var/log/kube-controller-manager.log
2>&1
  image: gcr.io/google/_containers/kube-controller-manager:fda24638d5
1a48baa13c35337fcd4793

```

There are several other flags to control leader election. All of them have reasonable defaults:

<code>--leader-elect-lease-duration</code> duration	Default: 15s
<code>--leader-elect-renew-deadline</code> duration	Default: 10s
<code>--leader-elect-resource-lock</code> endpoints ("configmaps" is the other option)	Default: "endpoints"
<code>--leader-elect-retry-period</code> duration	Default: 2s

Note that it is not possible to have these components restarted automatically by Kubernetes like other pods because these are exactly the Kubernetes components responsible for restarting failed pods, so they can't restart themselves if they fail. There must be a ready-to-go replacement already running.

Making your staging environment highly available

High availability is not trivial to set up. If you go to the trouble of setting up high availability, it means there is a business case for a highly available system. It follows that you want to test your reliable and highly available cluster before you deploy it to production (unless you're Netflix, where you test in production). Also, any change to the cluster may, in theory, break your high availability without disrupting other cluster functions. The essential point is that, just like anything else, if you don't test it, assume it doesn't work.

We've established that you need to test reliability and high availability. The best way to do this is to create a staging environment that replicates your production environment as closely as possible. This can get expensive. There are several ways to manage the cost:

- **An ad hoc highly available staging environment:** Create a large highly available cluster only for the duration of high availability testing.

- **Compress time:** Create interesting event streams and scenarios ahead of time, feed the input, and simulate the situations in rapid succession.
- **Combine high availability testing with performance and stress testing:** At the end of your performance and stress tests, overload the system and see how the reliability and high availability configuration handles the load.

Testing high availability

Testing high availability takes planning and a deep understanding of your system. The goal of every test is to reveal flaws in the system's design and/or implementation, and to provide good enough coverage that, if the tests pass, you'll be confident that the system behaves as expected.

In the realm of reliability, self-healing, and high availability, it means you need to figure out ways to break the system and watch it put itself back together.

This requires several elements, as follows:

- A comprehensive list of possible failures (including reasonable combinations)
- For each possible failure, it should be clear how the system should respond
- A way to induce the failure
- A way to observe how the system reacts

None of the elements are trivial. The best approach in my experience is to do it incrementally and try to come up with a relatively small number of generic failure categories and generic responses, rather than an exhaustive, ever-changing list of low-level failures.

For example, a generic failure category is node-unresponsive; the generic response could be rebooting the node; the way to induce the failure could be stopping the virtual machine (VM) of the node (if it's a VM). The observation should be that, while the node is down, the system still functions properly based on standard acceptance tests; the node is eventually up, and the system gets back to normal. There may be many other things you want to test, such as whether the problem was logged, whether relevant alerts went out to the right people, and whether various stats and reports were updated.

But beware of over-generalizing. In the case of the generic unresponsive node failure, a key component is detecting that the node is unresponsive. If your method of detection is faulty, then your system will not react properly. Use best practices like health checks and readiness checks.

Note that, sometimes, a failure can't be resolved in a single response. For example, in our unresponsive node case, if it's a hardware failure, then rebooting will not help. In this case, a second line of response comes into play and maybe a new node is provisioned to replace the failed node. In this case, you can't be too generic and you may need to create tests for specific types of pod/role that were on the node (such as etcd, master, worker, database, and monitoring).

If you have high quality requirements, be prepared to spend much more time setting up the proper testing environments and the tests than even the production environment.

One last important point is to try to be as unintrusive as possible. That means that, ideally, your production system will not have testing features that allow shutting down parts of it or cause it to be configured to run at reduced capacity for testing. The reason is that it increases the attack surface of your system and it could be triggered by accident by mistakes in configuration. Ideally, you can control your testing environment without resorting to modifying the code or configuration that will be deployed in production. With Kubernetes, it is usually easy to inject pods and containers with custom test functionality that can interact with system components in the staging environment, but will never be deployed in production.

In this section, we looked at what it takes to actually have a reliable and highly available cluster, including etcd, the API server, the scheduler, and the controller manager. We considered best practices for protecting the cluster itself, as well as your data, and paid special attention to the issue of starting environments and testing.

High availability, scalability, and capacity planning

Highly available systems must also be scalable. The load on most complicated distributed systems can vary dramatically based on the time of day, weekdays versus weekends, seasonal effects, marketing campaigns, and many other factors. Successful systems will have more users over time and accumulate more and more data. That means that the physical resources of the clusters – mostly nodes and storage – will have to grow over time too. If your cluster is under-provisioned, it will not be able to satisfy all the demand and it will not be available because requests will time out or be queued up and not processed fast enough.

This is the realm of capacity planning. One simple approach is to over-provision your cluster. Anticipate the demand and make sure you have enough of a buffer for spikes of activity. But be aware that this approach suffers from several deficiencies:

- For highly dynamic and complicated distributed systems, it's difficult to forecast the demand even approximately.
- Over-provisioning is expensive. You spend a lot of money on resources that are rarely or never used.
- You have to periodically redo the whole process because the average and peak load on the system changes over time.

A much better approach is to use intent-based capacity planning where high-level abstraction is used and the system adjusts itself accordingly. In the context of Kubernetes, there is the **horizontal pod autoscaler (HPA)** that can grow and shrink the number of pods needed to handle requests for a particular service. But, that works only to change the ratio of resources allocated to different services. When the entire cluster approaches saturation, you simply need more resources. This is where the cluster autoscaler comes into play. It is a Kubernetes project that became available with Kubernetes 1.8. It works particularly well in cloud environments where additional resources can be provisioned via programmatic APIs.

When the **cluster autoscaler (CA)** determines that pods can't be scheduled (that is, they are in the pending state), it provisions a new node for the cluster. It can also remove nodes from the cluster if it determines that the cluster has more nodes than necessary to handle the load. The CA will check for pending pods every 30 seconds. It will remove nodes only after 10 minutes of not being used, to avoid thrashing.

Here are some issues to consider:

- A cluster may require more nodes even if the total CPU or memory utilization is low due to control mechanisms like affinity, anti-affinity, taints, tolerations, pod priorities, and pod disruption budgets.
- In addition to the built-in delays in triggering the scaling up or down of nodes, there is an additional delay of several minutes when provisioning a new node from the cloud provider.
- The interactions between the HPA and the CA can be subtle.

Installing the cluster autoscaler

Note that you can't test the CA locally. You must have a Kubernetes cluster running on one of the following supported cloud providers:

- GCE
- GKE
- AWS EKS
- Azure
- Alibaba Cloud
- Baidu Cloud

I have installed the CA successfully on GKE as well as AWS EKS.

The `eks-cluster-autoscaler.yaml` file contains all the Kubernetes resources needed to install the CA on EKS. It involves creating a service account and giving it various RBAC permissions because it needs to monitor node usage across the cluster and be able to act on it. Finally, there is a deployment that actually deploys the CA image itself with a command-line interface that includes the range of nodes (that is, the minimum and maximum number) it should maintain, and in the case of EKS, a node group is needed too. The maximum number is important to prevent a situation where an attack or error causes the CA to just add more and more nodes uncontrollably, racking up a huge bill. Here is a snippet from the pod template:

```
spec:
  serviceAccountName: cluster-autoscaler
  containers:
    - image: k8s.gcr.io/cluster-autoscaler:v1.2.2
      name: cluster-autoscaler
      resources:
        limits:
          cpu: 100m
          memory: 300Mi
        requests:
          cpu: 100m
          memory: 300Mi
      command:
        - ./cluster-autoscaler
        - --v=4
        - --stderrthreshold=info
        - --cloud-provider=aws
        - --skip-nodes-with-local-storage=false
        - --nodes=2:5:eksctl-project-nodegroup-ng-name-NodeGroup-suffix
      env:
        - name: AWS_REGION
          value: us-east-1
      volumeMounts:
        - name: ssl-certs
          mountPath: /etc/ssl/certs/ca-certificates.crt
          readOnly: true
      imagePullPolicy: "Always"
  volumes:
    - name: ssl-certs
      hostPath:
        path: "/etc/ssl/certs/ca-bundle.crt"
```


The combination of the HPA and CA provides a truly elastic cluster where the HPA ensures that services use the proper amount of pods to handle the load per service, and the CA makes sure that the number of nodes matches the overall load on the cluster.

Considering the vertical pod autoscaler

The **vertical pod autoscaler (VPA)** is another autoscaler that operates on pods. Its job is to provide additional resources (CPU and memory) to pods that have too low limits. It is designed primarily for stateful services, but can work for stateless services too. It is based on a CRD (custom resource definition) and has three components:

- **Recommender:** Watches CPU and memory usage and provides recommendations for new values for CPU and memory requests
- **Updater:** Kills managed pods whose CPU and memory requests don't match the recommendations made by the recommender
- **Admission plugin:** Sets the CPU and memory requests for new or recreated pods based on recommendations

The VPA is still in beta. Here are some of the main limitations:

- Unable to update running pods (hence the updater kills pods to get them restarted with the correct requests)
- Can't evict pods that aren't managed by a controller
- The VPA is incompatible with the HPA

This section covered the interactions between auto-scalability and high availability and looked at different approaches for scaling Kubernetes clusters and the applications running on those clusters.

Live cluster updates

One of the most complicated and risky tasks involved in running a Kubernetes cluster is a live upgrade. The interactions between different parts of the system in different versions are often difficult to predict, but in many situations, it is required. Large clusters with many users can't afford to be offline for maintenance. The best way to attack complexity is to divide and conquer. Microservice architecture helps a lot here. You never upgrade your entire system. You just constantly upgrade several sets of related microservices, and if APIs have changed, then you upgrade their clients, too. A properly designed upgrade will preserve backward-compatibility at least until all clients have been upgraded, and then deprecate old APIs across several releases.

In this section, we will discuss how to go about updating your cluster using various strategies such as rolling updates, blue-green deployments, and canary deployments. We will also discuss when it's appropriate to introduce breaking upgrades versus backward-compatible upgrades. Then we will get into the critical topic of schema and data migrations.

Rolling updates

Rolling updates are updates where you gradually update components from the current version to the next. This means that your cluster will run current and new components at the same time. There are two different cases to consider here:

- New components are backward-compatible
- New components are not backward-compatible

If the new components are backward-compatible, then the upgrade should be very easy. In earlier versions of Kubernetes, you had to manage rolling updates very carefully with labels and change the number of replicas gradually for both the old and new versions (although `kubectl rolling-update` is a convenient shortcut for replication controllers). But, the Deployment resource introduced in Kubernetes 1.2 makes it much easier and supports replica sets. It has the following capabilities built in:

- Running server side (it keeps going if your machine disconnects)
- Versioning
- Multiple concurrent rollouts
- Updating deployments
- Aggregating status across all pods
- Rollbacks
- Canary deployments
- Multiple upgrade strategies (rolling upgrade is the default)

Here is a sample manifest for a deployment that deploys three nginx pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
```

```
replicas: 3   selector:
  matchLabels:
    app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9           ports:
      - containerPort: 80
```

The resource kind is Deployment and it's got the name `nginx-deployment`, which you can use to refer to this deployment later (for example, for updates or rollbacks). The most important part is, of course, the spec, which contains a pod template. The replicas determine how many pods will be in the cluster, and the template spec has the configuration for each container. In this case, this is just a single container.

To start the rolling update, create the Deployment resource and check that it rolled out successfully:

```
$ k create -f nginx-deployment.yaml
deployment.apps/nginx-deployment created
```

```
$ k rollout status deployment/nginx-deployment
deployment "nginx-deployment" successfully rolled out
```

Deployments have an update strategy, which defaults to `rollingUpdate`:

```
$ k get deployment nginx-deployment -o yaml | grep strategy -A 4
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
```

The following diagram illustrates how a rolling update works:

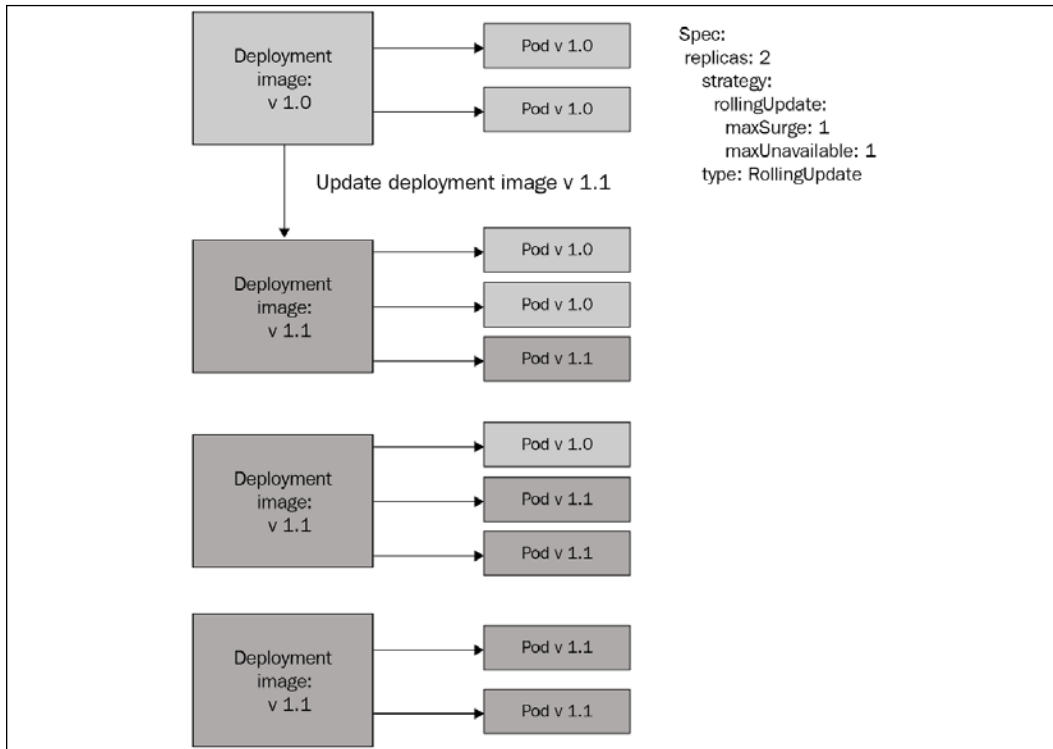


Figure 3.4: How a rolling update progresses

Complex deployments

The Deployment resource is great when you just want to upgrade one pod, but you may often need to upgrade multiple pods, and those pods sometimes have version inter-dependencies. In those situations, you must sometimes forgo a rolling update or introduce a temporary compatibility layer. For example, suppose service A depends on service B. Service B now has a breaking change. The v1 pods of service A can't interoperate with the pods from service B v2. It is also undesirable from both reliability and change-management points of view to make the v2 pods of service B support the old and new APIs. In this case, the solution may be to introduce an adapter service that implements the v1 API of the B service. This service will sit between A and B, and will translate requests and responses across versions. This adds complexity to the deployment process and requires several steps, but the benefit is that the A and B services themselves are simple. You can do rolling updates across incompatible versions and all indirection will go away once everybody upgrades to v2 (all A pods and all B pods).

But rolling updates are not always the answer.

Blue-green deployments

Rolling updates are great for availability, but sometimes the complexity involved in managing a proper rolling update is considered too high, or it adds a significant amount of work that pushes back more important projects. In these cases, blue-green releases provide a great alternative. With a blue-green release, you prepare a full copy of your production environment with the new version. Now you have two copies, old (blue) and new (green). It doesn't matter which one is blue and which one is green. The important thing is that you have two fully independent production environments. Currently, blue is active and services all requests. You can run all your tests on green. Once you're happy, you flip the switch and green becomes active. If something goes wrong, rolling back is just as easy; just switch back from green to blue.

The following diagram illustrates how blue-green deployments work using two deployments, two labels, and a single service that uses a label selector to switch from the blue deployment to the green deployment:

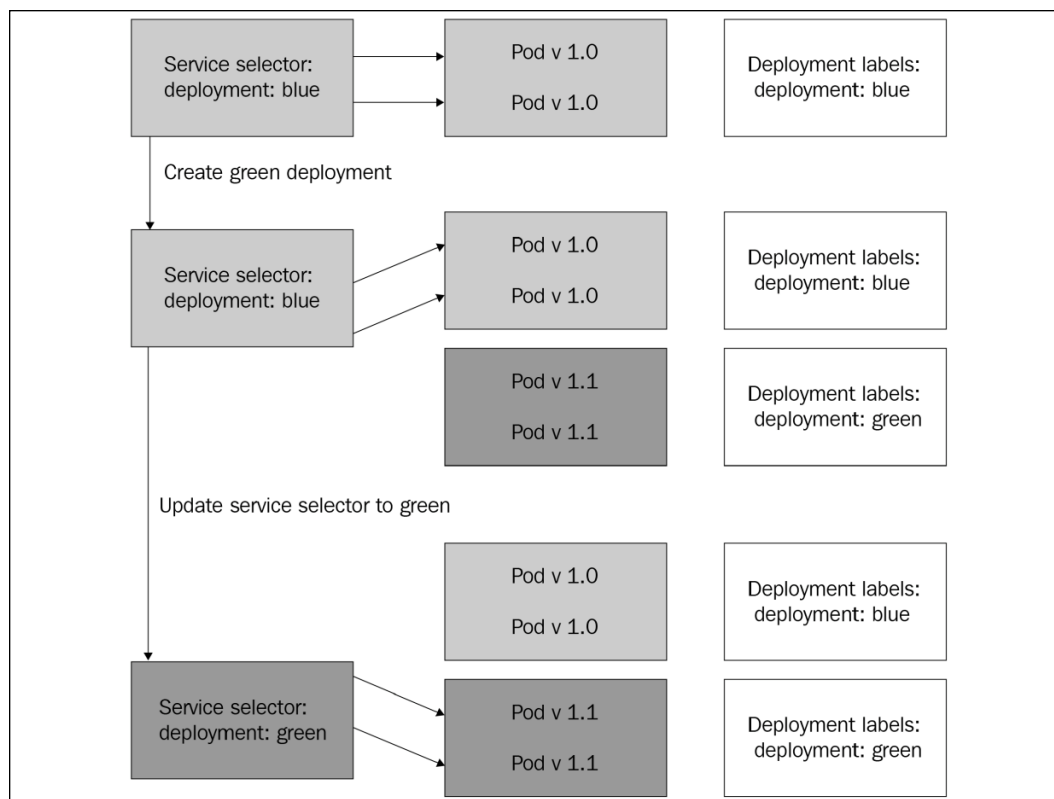


Figure 3.5: Blue-green deployment in practice

I totally ignored the storage and in-memory state in the previous discussion. This immediate switch assumes that blue and green are composed of stateless components only and share a common persistence layer.

If there were storage changes or breaking changes to the API accessible to external clients, then additional steps need to be taken. For example, if blue and green have their own storage, then all incoming requests may need to be sent to both blue and green, and green may need to ingest historical data from blue to get in sync before switching.

Canary deployments

Blue-green deployments are cool. However, there are times where a more nuanced approach is needed. Suppose you are responsible for a large distributed system with many users. The developers plan to deploy a new version of their service. They tested the new version of the service in the test and staging environments. But, the production environment is too complicated to be replicated one to one for testing purposes. This means there is a risk that the service will misbehave in production. That's where canary deployments shine.

The basic idea is to test the service in production, but in a limited capacity. This way, if something is wrong with the new version, only a small fraction of your users or a small fraction of requests will be impacted. This can be implemented very easily in Kubernetes at the pod level. If a service is backed up by 10 pods, then you deploy the new version and then only 10% of the requests will be routed by the service load balancer to the canary pod, while 90% of the requests will still be serviced by the current version.

The following diagram illustrates this approach:

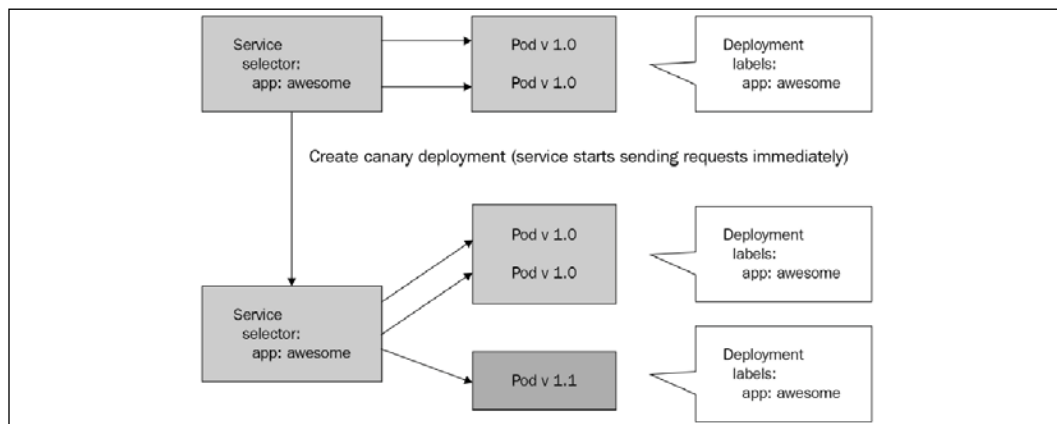


Figure 3.6: Canary deployment in practice

There are more sophisticated ways to route traffic to a canary deployment using a service mesh. We will examine that in a later chapter (*Chapter 14, Utilizing Service Meshes*).

Let's address the hard problem of managing data-contract changes.

Managing data-contract changes

Data contracts describe how data is organized. It's an umbrella term for structure metadata. The most common example is a relational database schema. Other examples include network payloads, file formats, and even the content of string arguments or responses. If you have a configuration file, then this configuration file has both a file format (JSON, YAML, TOML, XML, INI, or a custom format) and some internal structure that describes what kind of hierarchy, keys, values, and data types are valid. Sometimes the data contract is explicit and sometimes it's implicit. Either way, you need to manage it carefully, or else you'll get runtime errors when code that's reading, parsing, or validating encounters data with an unfamiliar structure.

Migrating data

Data migration is a big deal. Many systems these days manage staggering amounts of data measured in terabytes, petabytes, or more. The amount of collected and managed data will continue to increase for the foreseeable future. The pace of data collection exceeds the pace of hardware innovation. The essential point is that if you have a lot of data and you need to migrate it, it can take a while. In a previous company, I oversaw a project to migrate close to 100 terabytes of data from one Cassandra cluster of a legacy system to another Cassandra cluster.

The second Cassandra cluster had a different schema and was accessed by a Kubernetes cluster 24/7. The project was very complicated, and thus it kept getting pushed back when urgent issues popped up. The legacy system was still in place side by side with the next-gen system long after the original estimate.

There were a lot of mechanisms in place to split the data and send it to both clusters, but then we ran into scalability issues with the new system and we had to address those before we could continue. The historical data was important, but it didn't have to be accessed with the same service level as recent hot data. So, we embarked on yet another project to send historical data to cheaper storage. That meant, of course, that client libraries or frontend services had to know how to query both stores and merge the results. When you deal with a lot of data, you can't take anything for granted. You run into scalability issues with your tools, your infrastructure, your third-party dependencies, and your processes. Moving to a large scale is not just a quantity change; it often means qualitative change as well. Don't expect it to go smoothly. It is much more than copying some files from A to B.

Deprecating APIs

API deprecation comes in two flavors: internal and external. Internal APIs are APIs used by components that are fully controlled by you and your team or organization. You can be sure that all API users will upgrade to the new API within a short time. External APIs are used by users or services outside your direct sphere of influence. There are a few gray-area situations when you work for a huge organization (think Google), and even internal APIs may need to be treated as external APIs. If you're lucky, all your external APIs are used by self-updating applications or through a web interface you control. In those cases, the API is practically hidden and you don't even need to publish it.

If you have a lot of users (or a few very important users) using your API, you should consider deprecation very carefully. Deprecating an API means you force your users to change their application to work with you or stay locked to an earlier version.

There are a few ways you can mitigate the pain:

- Don't deprecate. Extend the existing API or keep the previous API active. It is sometimes pretty simple, although it adds to the testing burden.
- Provide client libraries in all relevant programming languages to your target audience. This is always a good practice. It allows you to make many changes to the underlying API without disrupting users (as long as you keep the programming language interface stable).
- If you have to deprecate, explain why, allow ample time for users to upgrade, and provide as much support as possible (for example, an upgrade guide with examples). Your users will appreciate it.

Large cluster performance, cost, and design trade-offs

In the previous section, we looked at live cluster upgrades and application updates. We explored various techniques and how Kubernetes supports them. We also discussed difficult problems such as breaking changes, data contract changes, data migration, and API deprecation. In this section, we will consider the various options and configurations of large clusters with different reliability and high availability properties. When you design your cluster, you need to understand your options and choose wisely based on the needs of your organization.

The topics we will cover include various availability requirements, from best effort all the way to the holy grail of zero downtime. Finally, we will settle down on the practical site-reliability engineering approach. For each category of availability, we will consider what it means from the perspectives of performance and cost.

Availability requirements

Different systems have very different requirements for reliability and availability. Moreover, different sub-systems have very different requirements. For example, billing systems are always a high priority because if the billing system is down, you can't make money. But, even within the billing system, if the ability to dispute charges is sometimes unavailable, it may be OK from a business point of view.

Best effort

Best effort means, counter-intuitively, no guarantee whatsoever. If it works, great! If it doesn't work – oh well, what are you going to do? This level of reliability and availability may be appropriate for internal components that change often so the effort to make them robust is not worth it. As long as the services or clients that invoke the unreliable services are able to handle the occasional errors or outages, then all is well. It may also be appropriate for services released in the wild as beta.

Best effort is great for developers. Developers can move fast and break things. They are not worried about the consequences and they don't have to go through a gauntlet of rigorous tests and approvals. The performance of best-effort services may be better than more robust services because the best-effort service can often skip expensive steps such as verifying requests, persisting intermediate results, and replicating data. But, on the other hand, more robust services are often heavily optimized and their supporting hardware is fine-tuned to their workload. The cost of best-effort services is usually lower because they don't need to employ redundancy, unless the operators neglect to do basic capacity planning and just over-provision needlessly.

In the context of Kubernetes, the big question is whether all the services provided by the cluster are best effort. If this is the case, then the cluster itself doesn't have to be highly available. You could probably have a single master node with a single instance of etcd, and Heapster or another monitoring solution may not need to be deployed. This is typically appropriate for local development clusters only. Even a shared development cluster that multiple developers use should have a decent level of reliability and robustness or else all the developers will be twiddling their thumbs whenever the cluster goes down unexpectedly.

Maintenance windows

In a system with maintenance windows, special times are dedicated for performing various maintenance activities, such as applying security patches, upgrading software, pruning log files, and database cleanups. With a maintenance window, the system (or a sub-system) becomes unavailable. This is planned off-time and, often, users are notified. The benefit of maintenance windows is that you don't have to worry how your maintenance actions are going to interact with live requests coming into the system. It can drastically simplify operations. System administrators and operators love maintenance windows just as much as developers love best-effort systems.

The downside, of course, is that the system is down during maintenance. This may only be acceptable for systems where user activity is limited to certain times (such as US office hours or weekdays only).

With Kubernetes, you can set up maintenance windows by redirecting all incoming requests via the load balancer to a web page (or JSON response) that notifies users about the maintenance window.

But in most cases, the flexibility of Kubernetes should allow you to do live maintenance. In extreme cases, such as upgrading the Kubernetes version, or the switch from etcd v2 to etcd v3, you may want to resort to a maintenance window. Blue-green deployment is another alternative. But the larger the cluster, the more expansive the blue-green alternative, because you must duplicate your entire production cluster, which is both costly and can cause you to run into problems like insufficient quota.

Quick recovery

Quick recovery is another important aspect of highly available clusters. Something will go wrong at some point. Your unavailability clock starts running. How quickly can you get back to normal?

Sometimes it's not up to you. For example, if your cloud provider has an outage (and you didn't implement a federated cluster, as we will discuss later), then you just have to sit and wait until they sort it out. But the most likely culprit is a problem with a recent deployment. There are, of course, time-related issues, and even calendar-related issues. Do you remember the leap-year bug that took down Microsoft Azure on February 29, 2012?

The poster boy of quick recovery is, of course, blue-green deployment – if you keep the previous version running when the problem is discovered. But, that's usually good for problems that happen during deployment or shortly after. If a sneaky bug lays dormant and is discovered only hours after the deployment, then you will have torn down your blue deployment already and you will not be able to revert to it.

On the other hand, rolling updates mean that if the problem is discovered early, then most of your pods will still run the previous version.

Data-related problems can take a long time to reverse, even if your backups are up to date and your restore procedure actually works (definitely test this regularly).

Tools like Heptio Velero can help in some scenarios by creating snapshot backups of your cluster that you can just restore if something goes wrong and you're not sure how to fix it.

Zero downtime

Finally, we arrive at the zero-downtime system. There is no such thing as a zero-downtime system. All systems fail and all software systems definitely fail. Sometimes the failure is serious enough that the system or some of its services will be down. Think about zero downtime as a best-effort distributed system design. You design for zero downtime in the sense that you provide a lot of redundancy and mechanisms to address expected failures without bringing the system down. As always, remember that, even if there is a business case for zero downtime, it doesn't mean that every component must have zero downtime. Reliable (within reason) systems can be constructed from highly unreliable components.

The plan for zero downtime is as follows:

- **Redundancy at every level:** This is a required condition. You can't have a single point of failure in your design because when it fails, your system is down.
- **Automated hot swapping of failed components:** Redundancy is only as good as the ability of the redundant components to kick into action as soon as the original component has failed. Some components can share the load (for example, stateless web servers), so there is no need for explicit action. In other cases, such as the Kubernetes scheduler and controller manager, you need a leader election in place to make sure the cluster keeps humming along.

- **Tons of metrics, monitoring, and alerts to detect problems early:** Even with careful design, you may miss something or some implicit assumption might invalidate your design. Often, such subtle issues creep up on you and with enough attention, you may discover it before it becomes an all-out system failure. For example, suppose there is a mechanism in place to clean up old log files when disk space is over 90% full, but for some reason, it doesn't work. If you set an alert for when disk space is over 95% full, then you'll catch it and be able to prevent the system failure.
- **Tenacious testing before deployment to production:** Comprehensive tests have proven themselves as a reliable way to improve quality. It is hard work to have comprehensive tests for something as complicated as a large Kubernetes cluster running a massive distributed system, but you need it. What should you test? Everything. That's right. For zero downtime, you need to test both the application and the infrastructure together. Your 100% passing unit tests are a good start, but they don't provide much confidence that when you deploy your application on your production Kubernetes cluster, it will still run as expected. The best tests are, of course, on your production cluster after a blue-green deployment or identical cluster. In lieu of a full-fledged identical cluster, consider a staging environment with as much fidelity as possible to your production environment. Here is a list of tests you should run. Each of these tests should be comprehensive because if you leave something untested, it might be broken:
 - Unit tests
 - Acceptance tests
 - Performance tests
 - Stress tests
 - Rollback tests
 - Data restore tests
 - Penetration tests

Does that sound crazy? Good. Zero-downtime, large-scale systems are hard. There is a reason why Microsoft, Google, Amazon, Facebook, and other big companies have tens of thousands of software engineers (combined) just working on infrastructure, operations, and making sure things are up and running.

- **Keep the raw data:** For many systems, the data is the most critical asset. If you keep the raw data, you can recover from any data corruption and processed data loss that happens later. This will not really help you with zero downtime because it can take a while to re-process the raw data, but it will help with zero data loss, which is often more important. The downside to this approach is that the raw data is often huge compared to the processed data. A good option may be to store the raw data in cheaper storage compared to the processed data.
- **Perceived uptime as a last resort:** OK. Some part of the system is down. You may still be able to maintain some level of service. In many situations, you may have access to a slightly stale version of the data or can let the user access some other part of the system. It is not a great user experience, but technically the system is still available.

Site reliability engineering

Site reliability engineering (SRE) is a real-world approach for operating reliable distributed systems. SRE embraces failures and works with **service-level indicators (SLIs)**, **service-level objectives (SLOs)**, and **service-level agreements (SLAs)**. Each service has an objective, such as latency below 50 milliseconds for 95% of requests. If a service violates its objectives, then the team focuses on fixing the issue before going back to work on new features and capabilities.

The beauty of SRE is that you get to play with the knobs for cost and performance. If you want to invest more in reliability, then be ready to pay for it with resources and development time.

Performance and data consistency

When you develop or operate distributed systems, the CAP theorem should always be in the back of your mind. CAP stands for consistency, availability, and partition tolerance.

Consistency means that every read receives the most recent write or an error. Availability means that every request receives a non-error response (but the response may be stale). Partition tolerance means the system continues to operate even when an arbitrary number of messages between nodes are dropped or delayed by the network.

The theorem says that you can have, at most, two out of the three. Since any distributed system can suffer from a network partition, in practice you can choose between CP or AP. CP means that in order to remain consistent, the system will not be available in the event of a network partition. AP means that the system will always be available but might not be consistent. For example, reads from different partitions might return different results because one of the partitions didn't receive a write. In this section, we will focus on highly available systems, which means AP. To achieve high availability, we must sacrifice consistency. But that doesn't mean that our system will have corrupt or arbitrary data. The keyword is eventual consistency. Our system may be a little bit behind and provide access to somewhat stale data, but eventually, you'll get what you expect.

When you start thinking in terms of eventual consistency, it opens the door to potentially significant performance improvements. For example, if some important value is updated frequently (for example, every second), but you send its value only every minute, you have reduced your network traffic by a factor of 60 and you're on average only 30 seconds behind real-time updates. This is very significant. This is huge. You have just scaled your system to handle 60 times more users or requests with the same amount of resources.

Summary

In this chapter, we looked at reliable and highly available large-scale Kubernetes clusters. This is arguably the sweet spot for Kubernetes. While it is useful to be able to orchestrate a small cluster running a few containers, it is not necessary, but at scale, you must have an orchestration solution in place you can trust to scale with your system, and provide the tools and the best practices to do that.

You now have a solid understanding of the concepts of reliability and high availability in distributed systems. You delved into the best practices for running reliable and highly available Kubernetes clusters. You explored the nuances of live Kubernetes cluster upgrades and you can make wise design choices regarding levels of reliability and availability, as well as their performance and cost.

In the next chapter, we will address the important topic of security in Kubernetes. We will also discuss the challenges of securing Kubernetes and the risks involved. We will learn all about namespaces, service accounts, admission control, authentication, authorization, and encryption.

References

- <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/ha-topology/>
- <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/high-availability/>
- <https://medium.com/magalix/kubernetes-autoscaling-101-cluster-autoscaler-horizontal-pod-autoscaler-and-vertical-pod-2a441d9ad231>

4

Securing Kubernetes

In *Chapter 3, High Availability and Reliability*, we looked at reliable and highly available Kubernetes clusters, the basic concepts, the best practices, how to do live updates, and the many design trade-offs regarding performance and cost.

In this chapter, we will explore the important topic of security. Kubernetes clusters are complicated systems composed of multiple layers of interacting components. Isolation and compartmentalization of different layers is very important when running critical applications. To secure the system and ensure proper access to resources, capabilities, and data, we must first understand the unique challenges facing Kubernetes as a general-purpose orchestration platform that runs unknown workloads. Then we can take advantage of various securities, isolation, and access control mechanisms to make sure the cluster, the applications running on it, and the data are all safe. We will discuss various best practices and when it is appropriate to use each mechanism.

At the end of this chapter, you will have a good understanding of Kubernetes security challenges. You will gain practical knowledge of how to harden Kubernetes against various potential attacks, establishing defense in depth, and will even be able to safely run a multi-tenant cluster while providing different users full isolation as well as full control over their part of the cluster.

Understanding Kubernetes security challenges

Kubernetes is a very flexible system that manages very low-level resources in a generic way. Kubernetes itself can be deployed on many operating systems and hardware or virtual-machine solutions, on-premises, or in the cloud. Kubernetes runs workloads implemented by runtimes it interacts with through a well-defined runtime interface, but without understanding how they are implemented. Kubernetes manipulates critical resources such as networking, DNS, and resource allocation on behalf of or in service of applications it knows nothing about. This means that Kubernetes is faced with the difficult task of providing good security mechanisms and capabilities in a way that application developers and cluster administrators can utilize, while protecting itself, the developers, and the administrators from common mistakes.

In this section, we will discuss security challenges in several layers or components of a Kubernetes cluster: nodes, network, images, pods, and containers. Defense in depth is an important security concept that requires systems to protect themselves at each level, both to mitigate attacks that penetrate other layers and to limit the scope and damage of a breach. Recognizing the challenges in each layer is the first step toward defense in depth.

Node challenges

The nodes are the hosts of the runtime engines. If an attacker gets access to a node, this is a serious threat. It can control at least the host itself and all the workloads running on it. But it gets worse. The node has a kubelet running that talks to the API server. A sophisticated attacker can replace the kubelet with a modified version and effectively evade detection by communicating normally with the Kubernetes API server, yet running its own workloads instead of the scheduled workloads, collecting information about the overall cluster, and disrupting the API server and the rest of the cluster by sending malicious messages. The node will have access to shared resources and to secrets that may allow it to infiltrate even deeper. A node breach is very serious, both because of the possible damage and the difficulty of detecting it after the fact.

Nodes can be compromised at the physical level too. This is more relevant on bare-metal machines where you can tell which hardware is assigned to the Kubernetes cluster.

Another attack vector is resource drain. Imagine that your nodes become part of a bot network that, unrelated to your Kubernetes cluster, just runs its own workloads like cryptocurrency mining and drains CPU and memory. The danger here is that your cluster will choke and run out of resources to run your workloads or alternatively, your infrastructure may scale automatically and allocate more resources.

Another problem is the installation of debugging and troubleshooting tools or modifying the configuration outside of an automated deployment. Those are typically untested and, if left behind and active, can lead to at least degraded performance, but can also cause more sinister problems. At the least, it increases the attack surface.

Where security is concerned, it's a numbers game. You want to understand the attack surface of the system and where you're vulnerable. Let's list all the node challenges:

- An attacker takes control of the host
- An attacker replaces the kubelet
- An attacker takes control of a node that runs master components (such as the API server, scheduler, or controller manager)
- An attacker gets physical access to a node
- An attacker drains resources unrelated to the Kubernetes cluster
- Self-inflicted damage occurs through the installation of debugging and troubleshooting tools or a configuration change

Network challenges

Any non-trivial Kubernetes cluster spans at least one network. There are many challenges related to networking. You need to understand how your system components are connected at a very fine level. Which components are supposed to talk to each other? What network protocols do they use? What ports? What data do they exchange? How is your cluster connected to the outside world?

There is a complex chain of exposing ports and capabilities or services:

- Container to host
- Host to host within the internal network
- Host to the world

Using overlay networks (which will be discussed more in *Chapter 10, Exploring Advanced Networking*) can help with defense in depth where, even if an attacker gains access to a container, they are sandboxed and can't escape to the underlay network's infrastructure.

Discovering components is a big challenge too. There are several options here, such as DNS, dedicated discovery services, and load balancers. Each comes with a set of pros and cons that take careful planning and insight to get right for your situation.

Making sure two containers can find each other and exchange information is not trivial.

You need to decide which resources and endpoints should be publicly accessible. Then you need to come up with a proper way to authenticate users, services, and authorize them to operate on resources. Often you may want to control access between internal services too.

Sensitive data must be encrypted on the way into and out of the cluster and sometimes at rest, too. That means key management and safe key exchange, which is one of the most difficult problems to solve in security.

If your cluster shares networking infrastructure with other Kubernetes clusters or non-Kubernetes processes then you have to be diligent about isolation and separation.

The ingredients are network policies, firewall rules, and **software-defined networking (SDN)**. The recipe is often customized. This is especially challenging with on-premises and bare-metal clusters. Let's recap:

- Come up with a connectivity plan
- Choose components, protocols, and ports
- Figure out dynamic discovery
- Public versus private access
- Authentication and authorization (including between internal services)
- Design firewall rules
- Decide on a network policy
- Key management and exchange

There is a constant tension between making it easy for containers, users, and services to find and talk to each other at the network level versus locking down access and preventing attacks through the network or attacks on the network itself.

Many of these challenges are not Kubernetes-specific. However, the fact that Kubernetes is a generic platform that manages key infrastructure and deals with low-level networking makes it necessary to think about dynamic and flexible solutions that can integrate system-specific requirements into Kubernetes.

Image challenges

Kubernetes runs containers that comply with one of its runtime engines. It has no idea what these containers are doing (except collecting metrics). You can put certain limits on containers via quotas. You can also limit their access to other parts of the network via network policies. But, in the end, containers do need access to host resources, other hosts in the network, distributed storage, and external services. The image determines the behavior of a container. There are two categories of problems with images:

- Malicious images
- Vulnerable images

Malicious images are images that contain code or configuration that was designed by an attacker to do some harm, collect information, or just take advantage of your infrastructure for their purposes (for example, crypto mining). Malicious code can be injected into your image preparation pipeline, including any image repositories you use. Alternatively, you may install third-party images that were compromised themselves and now contain malicious code.

Vulnerable images are images you designed (or third-party images you install) that just happen to contain some vulnerability that allows an attacker to take control of the running container or cause some other harm, including injecting their own code later.

It's hard to tell which category is worse. At the extreme, they are equivalent because they allow seizing total control of the container. The other defenses that are in place (remember defense in depth?) and the restrictions you put on the container will determine how much damage it can do. Minimizing the danger of bad images is very challenging. Fast-moving companies utilizing microservices may generate many images daily. Verifying an image is not an easy task either. Consider, for example, how Docker images are made of layers.

The base images that contain the operating system may become vulnerable any time a new vulnerability is discovered. Moreover, if you rely on base images prepared by someone else (very common) then malicious code may find its way into those base images, which you have no control over and you trust implicitly.

When a vulnerability in a third-party dependency is discovered, ideally there is already a fixed version and you should patch it as soon as possible.

We can summarize the image challenges that developers are likely to face as follows:

- Kubernetes doesn't know what images are doing
- Kubernetes must provide access to sensitive resources for the designated function
- It's difficult to protect the image preparation and delivery pipeline (including image repositories)
- The speed of development and deployment of new images conflict with the careful review of changes
- Base images that contain the OS or other common dependencies can easily get out of date and become vulnerable
- Base images are often not under your control and might be more prone to the injection of malicious code

Integrating a static image analyzer like CoreOS Clair or the Anchore Engine into your CI/CD pipeline can help a lot. In addition, minimizing the blast radius by limiting the resource access of containers only to what they need to perform their job can reduce the impact on your system if a container gets compromised. You must also be diligent about patching known vulnerabilities.

Configuration and deployment challenges

Kubernetes clusters are administered remotely. Various manifests and policies determine the state of the cluster at each point in time. If an attacker gets access to a machine with administrative control over the cluster, they can wreak havoc, such as collecting information, injecting bad images, weakening security, and tampering with logs. As usual, bugs and mistakes can be just as harmful; by neglecting important security measures, you leave the cluster open for attack. It is very common these days for employees with administrative access to the cluster to work remotely from home or from a coffee shop and have their laptops with them, where you are just one `kubectl` command from opening the floodgates.

Let's reiterate the challenges:

- Kubernetes is administered remotely
- An attacker with remote administrative access can gain complete control over the cluster

- Configuration and deployment is typically more difficult to test than code
- Remote or out-of-office employees risk extended exposure, allowing an attacker to gain access to their laptops or phones with administrative access

There are some best practices to minimize this risk, such as a layer of indirection in the form of a jump box, requiring a VPN connection, and using multi-factor authentication and one-time passwords.

Pod and container challenges

In Kubernetes, pods are the unit of work and contain one or more containers. The pod is a grouping and deployment construct. But often, containers that are deployed together in the same pod interact through direct mechanisms. The containers all share the same localhost network and often share mounted volumes from the host. This easy integration between containers in the same pod can result in exposing parts of the host to all the containers. This might allow one bad container (either malicious or just vulnerable) to open the way for an escalated attack on other containers in the pod, later taking over the node itself and the entire cluster. Master add-ons are often colocated with master components and present that kind of danger, especially because many of them are experimental. The same goes for daemon sets that run pods on every node. The practice of sidecar containers where additional containers are deployed in a pod along with your application container is very popular, especially with service meshes. This increases that risk because the sidecar containers are often outside your control, and if compromised, can provide access to your infrastructure.

Multi-container pod challenges include the following:

- The same pod containers share the localhost network
- The same pod containers sometimes share a mounted volume on the host filesystem
- Bad containers might poison other containers in the pod
- Bad containers have an easier time attacking the node if colocated with another container that accesses crucial node resources
- Experimental add-ons that are colocated with master components might be experimental and less secure
- Service meshes introduce a sidecar container that might become an attack vector

Consider carefully the interaction between containers running in the same pod. You should realize that a bad container might try to compromise its sibling containers in the same pod as its first attack.

Organizational, cultural, and process challenges

Security is often held in contrast to productivity. This is a normal trade-off and nothing to worry about. Traditionally, when developers and operations were separate, this conflict was managed at an organizational level. Developers pushed for more productivity and treated security requirements as the cost of doing business. Operations controlled the production environment and were responsible for access and security procedures. The DevOps movement brought down the wall between developers and operations. Now, speed of development often takes a front-row seat. Concepts such as continuous deployment deploying multiple times a day without human intervention were unheard of in most organizations. Kubernetes was designed for this new world of cloud-native applications. But, it was developed based on Google's experience. Google had a lot of time and skilled experts to develop the proper processes and tooling to balance rapid deployments with security. For smaller organizations, this balancing act might be very challenging and security could be weakened by focusing too much on productivity.

The challenges facing organizations that adopt Kubernetes are as follows:

- Developers that control the operation of Kubernetes might be less security-oriented
- The speed of development might be considered more important than security
- Continuous deployment might make it difficult to detect certain security problems before they reach production
- Smaller organizations might not have the knowledge and expertise to manage security properly in Kubernetes clusters

There are no easy answers here. You should be deliberate in striking the right balance between security and agility. I recommend having a dedicated security team (or at least one person focused on security) participate in all planning meetings and advocate for security. It's important to bake security into your system from the get-go.

In this section, we reviewed the many challenges you face when you try to build a secure Kubernetes cluster. Most of these challenges are not specific to Kubernetes, but using Kubernetes means there is a large part of your system that is generic and is unaware of what the system is doing.

This can pose problems when trying to lock down a system. The challenges are spread across different levels:

- Node challenges
- Network challenges
- Image challenges
- Configuration and deployment challenges
- Pod and container challenges
- Organizational and process challenges

In the next section, we will look at the facilities Kubernetes provides to address some of those challenges. Many of the challenges require solutions at the larger system scope. It is important to realize that just utilizing all of Kubernetes' security features is not enough.

Hardening Kubernetes

The previous section cataloged and listed the variety of security challenges facing developers and administrators deploying and maintaining Kubernetes clusters. In this section, we will hone in on the design aspects, mechanisms, and features offered by Kubernetes to address some of the challenges. You can get to a pretty good state of security by judicious use of capabilities such as service accounts, network policies, authentication, authorization, admission control, AppArmor, and secrets.

Remember that a Kubernetes cluster is one part of a bigger system that includes other software systems, people, and processes. Kubernetes can't solve all problems. You should always keep in mind general security principles, such as defense in depth, a need-to-know basis, and the principle of least privilege. In addition, log everything you think may be useful in the event of an attack and have alerts for early detection when the system deviates from its state. It may be just a bug or it may be an attack. Either way, you want to know about it and respond.

Understanding service accounts in Kubernetes

Kubernetes has regular users that are managed outside the cluster for humans connecting to the cluster (for example, via the `kubectl` command), and it has service accounts.

Regular user accounts are global and can access multiple namespaces in the cluster. Service accounts are constrained to one namespace. This is important. It ensures namespace isolation, because whenever the API server receives a request from a pod, its credentials will apply only to its own namespace.

Kubernetes manages service accounts on behalf of the pods. Whenever Kubernetes instantiates a pod, it assigns the pod a service account. The service account identifies all the pod processes when they interact with the API server. Each service account has a set of credentials mounted in a secret volume. Each namespace has a default service account called default. When you create a pod, it is automatically assigned the default service account unless you specify a different service account.

You can create additional service accounts. Create a file called `custom-service-account.yaml` with the following content:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: custom-service-account
```

Now type the following:

```
$ kubectl create -f custom-service-account.yaml
serviceaccount/custom-service-account created
```

Here is the service account listed alongside the default service account:

```
$ kubectl get serviceAccounts
```

NAME	SECRETS	AGE
custom-service-account	1	39s
default	1	18d

Note that a secret was created automatically for your new service account.

To get more detail, type the following:

```
$ kubectl get serviceAccounts/custom-service-account -o yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  creationTimestamp: "2020-06-01T01:24:24Z"
  name: custom-service-account
  namespace: default
  resourceVersion: "654316"
  selfLink: /api/v1/namespaces/default/serviceaccounts/custom-service-account
```

```
uid: 69393e47-c3b2-11e9-bb43-0242ac130002
secrets:
- name: custom-service-account-token-kdwhs
```

You can see the secret itself, which includes a `ca.crt` file and a token, by typing the following:

```
$ kubectl get secret custom-service-account-token-kdwhs -o yaml
```

How does Kubernetes manage service accounts?

The API server has a dedicated component called the service account admission controller. It is responsible for checking, at pod creation time, if the API server has a custom service account and, if it does, that the custom service account exists. If there is no service account specified, then it assigns the default service account.

It also ensures the pod has `ImagePullSecrets`, which are necessary when images need to be pulled from a remote image registry. If the pod spec doesn't have any secrets, it uses the service account's `ImagePullSecrets`.

Finally, it adds a volume with an API token for API access and a `volumeSource` mounted at `/var/run/secrets/kubernetes.io/serviceaccount`.

The API token is created and added to the secret by another component called the **Token Controller** whenever a service account is created. The Token Controller also monitors secrets and adds or removes tokens wherever secrets are added to or removed from a service account.

The service account controller ensures the default service account exists for every namespace.

Accessing the API server

Accessing the API server requires a chain of steps that include authentication, authorization, and admission control. At each stage, the request may be rejected. Each stage consists of multiple plugins that are chained together.

The following diagram illustrates this:

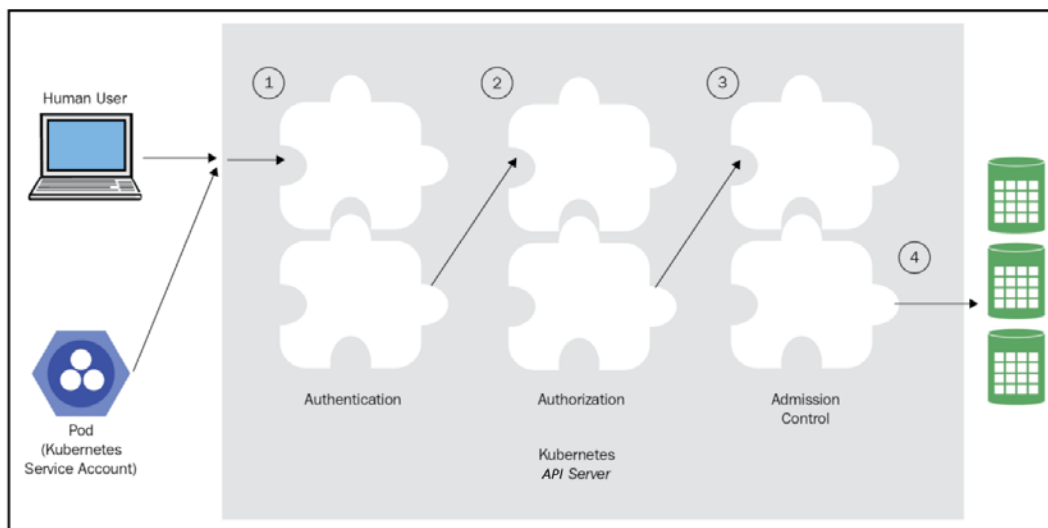


Figure 4.1: Accessing the API server

Authenticating users

When you first create the cluster, some keys and certificates are created for you to authenticate against the cluster. Kubectl uses them to authenticate itself to the API server and vice versa over TLS (an encrypted HTTPS connection). You can view your configuration using this command:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://localhost:6443
    name: default
contexts:
- context:
    cluster: default
    user: default
    name: default
current-context: default
kind: Config
preferences: {}
users:
```

```
- name: default
  user:
    password: DATA+OMITTED
    username: admin
```

This is the configuration for a k3d cluster. It may look different for other types of clusters.

Note that if multiple users need to access the cluster, the creator should provide the necessary client certificates and keys to the other users in a secure manner.

This is just establishing basic trust with the Kubernetes API server itself. You're not authenticated yet. Various authentication modules may look at the request and check for various additional client certificates, passwords, bearer tokens, and JWT tokens (for service accounts). Most requests require an authenticated user (either a regular user or a service account), although there are some anonymous requests too. If a request fails to authenticate with all the authenticators it will be rejected with a 401 HTTP status code (unauthorized, which is a bit of a misnomer).

The cluster administrator determines what authentication strategies to use by providing various command-line arguments to the API server:

- `--client-ca-file=` (for x509 client certificates specified in a file)
- `--token-auth-file=` (for bearer tokens specified in a file)
- `--basic-auth-file=` (for user/password pairs specified in a file)
- `--enable-bootstrap-token-auth` (for bootstrap tokens used by kubeadm)

Service accounts use an automatically loaded authentication plugin. The administrator may provide two optional flags:

- `--service-account-key-file=` (A PEM-encoded key for signing bearer tokens. If unspecified, the API server's TLS private key will be used.)
- `--service-account-lookup` (If enabled, tokens that are deleted from the API will be revoked.)

There are several other methods, such as OpenID Connect, webhooks, Keystone (the OpenStack identity service), and an authenticating proxy. The main theme is that the authentication stage is extensible and can support any authentication mechanism.

The various authentication plugins will examine the request and, based on the provided credentials, will associate the following attributes:

- **username** (a user-friendly name)
- **uid** (a unique identifier and more consistent than the username)

- **groups** (a set of group names the user belongs to)
- **extra fields** (these map string keys to string values)

In Kubernetes 1.11, `kubectl` gained the ability to use credential plugins to receive an opaque token from a provider such as an organizational LDAP server. These credentials are sent by `kubectl` to the API server that typically uses a webhook token authenticator to authenticate the credentials and accept the request.

The authenticators have no knowledge whatsoever of what a particular user is allowed to do. They just map a set of credentials to a set of identities. The authenticators run in an unspecified order; the first authenticator to accept the passed credentials will associate an identity with the incoming request and the authentication is considered successful. If all authenticators reject the credentials then authentication failed.

Impersonation

It is possible for users to impersonate different users (with proper authorization). For example, an admin may want to troubleshoot some issue as a different user with fewer privileges. This requires passing impersonation headers to the API request. The headers are as follows:

- **Impersonate-User:** The username to act as.
- **Impersonate-Group:** A group name to act as. Can be provided multiple times to set multiple groups. Optional. Requires `Impersonate-User`.
- **Impersonate-Extra-(extra name):** A dynamic header used to associate extra fields with the user. Optional. Requires `Impersonate-User`.

With `kubectl`, you pass `--as` and `--as-group` parameters.

Authorizing requests

Once a user is authenticated, authorization commences. Kubernetes has generic authorization semantics. A set of authorization modules receives the request, which includes information such as the authenticated username and the request's verb (list, get, watch, create, and so on). Unlike authentication, all authorization plugins will get a shot at any request. If a single authorization plugin rejects the request or no plugin had an opinion then it will be rejected with a 403 HTTP status code (forbidden). A request will continue only if at least one plugin accepts it and no other plugin rejected it.

The cluster administrator determines what authorization plugins to use by specifying the `--authorization-mode` command-line flag, which is a comma-separated list of plugin names.

The following modes are supported:

- `--authorization-mode=AlwaysDeny` rejects all requests. Use if you don't need authorization.
- `--authorization-mode=AlwaysAllow` allows all requests. Use if you don't need authorization. This is useful during testing.
- `--authorization-mode=ABAC` allows for a simple, local file-based, user-configured authorization policy. ABAC stands for Attribute-Based Access Control.
- `--authorization-mode=RBAC` is a role-based mechanism where authorization policies are stored and driven by the Kubernetes API. RBAC stands for Role-Based Access Control.
- `--authorization-mode=Node` is a special mode designed to authorize API requests made by kubelets.
- `--authorization-mode=Webhook` allows for authorization to be driven by a remote service using REST.

You can add your own custom authorization plugin by implementing the following straightforward Go interface:

```
type Authorizer interface {  
  
    Authorize(a Attributes) (authorized bool, reason string, err error)  
  
}
```

The `Attributes` input argument is also an interface that provides all the information you need to make an authorization decision:

```
type Attributes interface {  
    GetUser() user.Info  
    GetVerb() string  
    IsReadOnly() bool  
    GetNamespace() string  
    GetResource() string  
    GetSubresource() string  
    GetName() string  
    GetAPIGroup() string  
    GetAPIVersion() string  
    IsResourceRequest() bool  
    GetPath() string  
}
```

You can find the source code at <https://github.com/kubernetes/apiserver/blob/master/pkg/authorization/authorizer/interfaces.go>.

Using the `kubectl can-i` command, you check what actions you can perform and even impersonate other users:

```
$ kubectl auth can-i create deployments
```

```
Yes
```

```
$ kubectl auth can-i create deployments --as jack
```

```
no
```

Using admission control plugins

OK. The request was authenticated and authorized, but there is one more step before it can be executed. The request must go through a gauntlet of admission-control plugins. Similar to the authorizers, if a single admission controller rejects a request, it is denied.

Admission controllers are a neat concept. The idea is that there may be global cluster concerns that could be grounds for rejecting a request. Without admission controllers, all authorizers would have to be aware of these concerns and reject the request. But, with admission controllers, this logic can be performed once. In addition, an admission controller may modify the request. Admission controllers run in either validating mode or mutating mode. As usual, the cluster administrator decides which admission control plugins run by providing a command-line argument called `admission-control`. The value is a comma-separated and ordered list of plugins. Here is the list of recommended plugins for Kubernetes ≥ 1.9 (the order matters):

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,PersistentVolumeLabel,DefaultStorageClass,MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota,DefaultTolerationSeconds
```

Let's look at some of the available plugins (more are added all the time):

- `DefaultStorageClass`: Adds a default storage class to requests for the creation of a `PersistentVolumeClaim` that doesn't specify a storage class.
- `DefaultTolerationSeconds`: Sets the default toleration of pods for taints (if not set already): `notready:NoExecute` and `notreachable:NoExecute`.
- `EventRateLimit`: Limits flooding of the API server with events (new in Kubernetes 1.9).

- **ExtendedResourceToleration**: Combine taints on nodes with special resources such as GPUs and **Field Programmable Gate Array (FPGA)** with toleration on pods that request those resources. The end result is that the node with the extra resources will be dedicated for pods with the proper toleration.
- **ImagePolicyWebhook**: This complicated plugin connects to an external backend to decide whether a request should be rejected based on the image.
- **LimitPodHardAntiAffinity**: Denies any pod that defines the **AntiAffinity** topology key other than `kubernetes.io/hostname` in `requiredDuringSchedulingRequiredDuringExecution`.
- **LimitRanger**: Rejects requests that violate resource limits.
- **MutatingAdmissionWebhook**: Calls registered mutating webhooks that are able to modify their target object. Note that there is no guarantee that the change will be effective due to potential changes by other mutating webhooks.
- **NamespaceAutoProvision**: Creates the namespace in the request if it doesn't exist already.
- **NamespaceLifecycle**: Rejects object creation requests in namespaces that are in the process of being terminated or don't exist.
- **PodSecurityPolicy**: Rejects a request if the request security context doesn't conform to pod security policies.
- **ResourceQuota**: Rejects requests that violate the namespace's resource quota.
- **ServiceAccount**: Automation for service accounts.
- **ValidatingAdmissionWebhook**: This admission controller calls any validating webhooks that match the request. Matching webhooks are called in parallel; if any of them rejects the request, the request fails.

As you can see, the admission control plugins have very diverse functionality. They support namespace-wide policies and enforce validity of requests mostly from the resource management and security points of view. This frees up the authorization plugins to focus on valid operations. **ImagePolicyWebHook** is the gateway to validating images, which is a big challenge. **MutatingAdmissionWebhook** and **ValidatingAdmissionWebhook** are the gateways to dynamic admission control, where you can deploy your own admission controller without compiling it into Kubernetes. Dynamic admission control is suitable for tasks like semantic validation of resources (do all pods have the standard set of labels?).

The division of responsibility for validating an incoming request through the separate stages of authentication, authorization, and admission, each with its own plugins, makes a complicated process much more manageable to understand and use.

The mutating admission controllers provide a lot of flexibility and the ability to automatically enforce certain policies without burdening the users (for example, creating a namespace automatically if it doesn't exist).

Securing pods

Pod security is a major concern, since Kubernetes schedules the pods and lets them run. There are several independent mechanisms for securing pods and containers. Together these mechanisms support defense in depth, where, even if an attacker (or a mistake) bypasses one mechanism, it will get blocked by another.

Using a private image repository

This approach gives you a lot of confidence that your cluster will only pull images that you have previously vetted, and you can manage upgrades better. You can configure your `HOME/.docker/config.json` on each node. But, on many cloud providers, you can't do this because nodes are provisioned automatically for you.

ImagePullSecrets

This approach is recommended for clusters on cloud providers. The idea is that the credentials for the registry will be provided by the pod, so it doesn't matter what node it is scheduled to run on. This circumvents the problem with `.dockercfg` at the node level.

First, you need to create a secret object for the credentials:

```
$ kubectl create secret the-registry-secret
--docker-server=<docker registry server>
--docker-username=<username>
--docker-password=<password>
--docker-email=<email>
secret 'docker-registry-secret' created.
```

You can create secrets for multiple registries (or multiple users for the same registry) if needed. The kubelet will combine all `ImagePullSecrets`.

But, since pods can access secrets only in their own namespace, you must create a secret on each namespace where you want the pod to run.

Once the secret is defined, you can add it to the pod spec and run some pods on your cluster. The pod will use the credentials from the secret to pull images from the target image registry:

```
apiVersion: v1
kind: Pod
metadata:
  name: cool-pod
  namespace: the-namespace
spec:
  containers:
    - name: cool-container
      image: cool/app:v1
  imagePullSecrets:
    - name: the-registry-secret
```

Specifying a security context

A security context is a set of operating-system-level security settings such as UID, gid, capabilities, and SELinux role. These settings are applied at the container level as a container security context. You can specify a pod security context that will apply to all the containers in the pod. The pod security context can also apply its security settings (in particular, `fsGroup` and `seLinuxOptions`) to volumes.

Here is a sample pod security context:

```
apiVersion: v1
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    ...
  securityContext:
    fsGroup: 1234
    supplementalGroups: [5678]
    seLinuxOptions:
      level: 's0:c123,c456'
```

The container security context is applied to each container and it overrides the pod security context. It is embedded in the containers section of the pod manifest. Container context settings can't be applied to volumes, which remain at the pod level.

Here is a sample container security context:

```
apiVersion: v1
```

```
kind: Pod
metadata:
  name: hello-world
spec:
  containers:
    - name: hello-world-container
      # The container definition
      # ...
      securityContext:
        privileged: true
        seLinuxOptions:
          level: 's0:c123,c456'
```

Protecting your cluster with AppArmor

AppArmor is a Linux kernel security module. With AppArmor, you can restrict a process running in a container to a limited set of resources such as network access, Linux capabilities, and file permissions. You configure AppArmor through profiles.

Requirements

AppArmor support was added as beta in Kubernetes 1.4. It is not available for every operating system, so you must choose a supported OS distribution in order to take advantage of it. Ubuntu and SUSE Linux support AppArmor and enable it by default. Other distributions have optional support. To check if AppArmor is enabled, type the following:

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

If the result is Y then it's enabled.

The profile must be loaded into the kernel. Check the following file:

```
/sys/kernel/security/apparmor/profiles
```

Also, only the Docker runtime supports AppArmor at this time.

Securing a pod with AppArmor

Since AppArmor is still in beta, you specify the metadata as annotations and not as bonafide fields. When it gets out of beta, this will change.

To apply a profile to a container, add the following annotation:

```
container.apparmor.security.beta.kubernetes.io/:
```

The profile reference can be either the default profile, runtime/default, or a profile file on the host/localhost.

Here is a sample profile that prevents writing to files:

```
#include <tunables/global>
profile k8s-apparmor-example-deny-write flags=(attach\_disconnected) {
    #include <abstractions/base>
    file,
    # Deny all file writes.
    deny /\*\* w,
}
```

AppArmor is not a Kubernetes resource, so the format is not the YAML or JSON you're familiar with.

To verify the profile was attached correctly, check the attributes of process 1:

```
kubect1 exec <pod-name> cat /proc/1/attr/current
```

Pods can be scheduled on any node in the cluster by default. This means the profile should be loaded into every node. This is a classic use case for DaemonSet.

Writing AppArmor profiles

Writing profiles for AppArmor by hand is not trivial. There are some tools that can help: aa-genprof and aa-logprof can generate a profile for you and assist in fine-tuning it by running your application with AppArmor in complain mode. The tools keep track of your application's activity and AppArmor warnings, and create a corresponding profile. This approach works, but it feels clunky.

My favorite tool is bane (<https://github.com/jessfraz/bane>), which generates AppArmor profiles from a simpler profile language based on the TOML syntax. Bane profiles are very readable and easy to grasp. Here is a snippet from a bane profile:

```
Name = 'nginx-sample'
[Filesystem]
# read only paths for the container
ReadOnlyPaths = [
    '/bin/\*\*',
    '/boot/\*\*',
```

```
    '/dev/\*\*',
  ]
  # paths where you want to log on write
  LogOnWritePaths = [
    '/\*\*'
  ]

  # allowed capabilities
  [Capabilities]
  Allow = [
    'chown',
    'setuid',
  ]
  [Network]
  Raw = false
  Packet = false
  Protocols = [
    'tcp',
    'udp',
    'icmp'
  ]
]
```

The generated AppArmor profile is pretty gnarly.

Pod security policies

Pod security policies (PSPs) are available as beta since Kubernetes 1.4. It must be enabled, and you must also enable the PSP admission control to use them. A PSP is defined at the cluster level and defines the security context for pods. There are a couple of differences between using a PSP and directly specifying a security context in the pod manifest, as we did earlier:

- Apply the same policy to multiple pods or containers
- Let the administrator control pod creation so users don't create pods with inappropriate security contexts
- Dynamically generate a different security context for a pod via the admission controller

PSPs really scale the concept of security contexts. Typically, you'll have a relatively small number of security policies compared to the number of pods (or rather, pod templates). This means that many pod templates and containers will have the same security policy. Without PSP, you have to manage it individually for each pod manifest.

Here is a sample PSP that allows everything:

```
kind: PodSecurityPolicy
apiVersion: extensions/v1beta1policy/v1beta1
metadata:
  name: permissive
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: RunAsAny
  volumes:
    - "*"

```

As you can see it is much more human-readable than AppArmor, and is available on every OS and runtime.

Authorizing pod security policies via RBAC

This is the recommended way to enable the use of policies. Let's create a ClusterRole (Role works too) to grant access to use the target policies. It should look like the following:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: <role name>
rules:
- apiGroups: ['extensionspolicy']
  resources: ['podsecuritypolicies']
  verbs: ['use']
  resourceNames:
    - <list of policies to authorize>

```

Then, we need to bind the cluster role to the authorized users:

```
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:

```

```
  name: <binding name>
roleRef:
  kind: ClusterRole
  name: <role name>
  apiGroup: rbac.authorization.k8s.io
subjects:
- < list of authorized service accounts >
```

Here is a specific service account:

```
- kind: ServiceAccount
  name: <authorized service account name>
  namespace: <authorized pod namespace>
```

You can also authorize specific users, but it's not recommended:

```
- kind: User
  apiGroup: rbac.authorization.k8s.io
  name: <authorized user name>
```

If using a role binding instead of cluster role binding, then it will apply only to pods in the same namespace as the binding. This can be paired with system groups to grant access to all pods run in the namespace:

```
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:serviceaccounts
```

Or equivalently, granting access to all authenticated users in a namespace is done as follows:

```
- kind: Group
  apiGroup: rbac.authorization.k8s.io
  name: system:authenticated
```

Managing network policies

Node, pod, and container security is imperative, but it's not enough. Network segmentation is critical to design secure Kubernetes clusters that allow multi-tenancy, as well as to minimize the impact of security breaches. Defense in depth mandates that you compartmentalize parts of the system that don't need to talk to each other, while also carefully managing the direction, protocols, and ports of traffic.

Network policies allow the fine-grained control and proper network segmentation of your cluster. At the core, a network policy is a set of firewall rules applied to a set of namespaces and pods selected by labels. This is very flexible because labels can define virtual network segments and be managed as a Kubernetes resource.

This is a huge improvement over trying to segment your network using traditional approaches like IP address ranges and subnet masks, where you often run out of IP addresses or allocate too many just in case.

Choosing a supported networking solution

Some networking backends (network plugins) don't support network policies. For example, the popular Flannel can't be used to apply policies. This is critical. You will be able to define network policies even if your network plugin doesn't support them. Your policies will simply have no effect, giving you a false sense of security.

Here is a list of network plugins that support network policies (both ingress and egress):

- Calico
- WeaveNet
- Canal
- Cilium
- Kube-Router
- Romana
- Contiv

If you run your cluster on a managed Kubernetes service then the choice has already been made for you.

We will explore the ins and outs of network plugins in *Chapter 10, Exploring Advanced Networking*. Here we focus on network policies.

Defining a network policy

You define a network policy using a standard YAML manifest.

Here is a sample policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
```



```
name: the-network-policy
namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
    - from:
      - namespaceSelector:
          matchLabels:
            project: cool-project
      - podSelector:
          matchLabels:
            role: frontend
  ports:
    - protocol: tcp
      port: 6379
```

The spec part has two important parts, the podSelector and the ingress. The podSelector governs which pods this network policy applies to. The ingress governs which namespaces and pods can access these pods and which protocols and ports they can use.

In the preceding sample network policy, the pod selector specified the target for the network policy to be all the pods that are labeled `role: db`. The ingress section has a `from` sub-section with a namespace selector and a pod selector. All the namespaces in the cluster that are labeled `project: cool-project`, and within these namespaces, all the pods that are labeled `role: frontend` can access the target pods labeled `role: db`. The ports section defines a list of pairs (protocol and port) that further restrict what protocols and ports are allowed. In this case, the protocol is `tcp` and the port is `6379` (the standard Redis port).

Note that the network policy is cluster-wide, so pods from multiple namespaces in the cluster can access the target namespace. The current namespace is always included, so even if it doesn't have the `project: cool` label, pods with `role: frontend` can still have access.

It's important to realize that the network policy operates in a whitelist fashion. By default, all access is forbidden, and the network policy can open certain protocols and ports to certain pods that match the labels. However, the whitelist nature of the network policy applies only to pods that are selected for at least one network policy. If a pod is not selected it will allow all access. Always make sure all your pods are covered by a network policy.

Another implication of the whitelist nature is that, if multiple network policies exist, then the unified effect of all the rules applies. If one policy gives access to port 1234 and another gives access to port 5678 for the same set of pods, then a pod may be accessed through either 1234 or 5678.

To use network policies responsibly, consider starting with a deny-all network policy:

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

Then, start adding network policies to allow ingress to specific pods explicitly. Note that you must apply the deny-all policy for each namespace:

```
$ kubectl -n <namespace> create -f deny-all-network-policy.yaml
```

Limiting egress to external networks

Kubernetes 1.8 added egress network policy support, so you can control outbound traffic too. Here is an example that prevents access to the external IP 1.2.3.4. The order: 999 ensures the policy is applied before other policies:

```
apiVersion: v1
kind: policy
metadata:
  name: default-deny-egress
spec:
  order: 999
  egress:
    - action: deny
      destination:
        net: 1.2.3.4
      source: {}
```

Cross-namespace policies

If you divide your cluster into multiple namespaces, it can come in handy sometimes if pods can communicate across namespaces. You can specify the `ingress.namespaceSelector` field in your network policy to enable access from multiple namespaces. This is useful, for example, if you have production and staging namespaces and you periodically populate your staging environments with snapshots of your production data.

Using secrets

Secrets are paramount in secure systems. They can be credentials such as usernames and passwords, access tokens, API keys, certificates, or crypto keys. Secrets are typically small. If you have large amounts of data you want to protect, you should encrypt it and keep the encryption/decryption keys as secrets.

Storing secrets in Kubernetes

Kubernetes used to store secrets in etcd as plaintext by default. This means that direct access to etcd should be limited and carefully guarded. Starting with Kubernetes 1.7, you can now encrypt your secrets at rest (when they're stored by etcd).

Secrets are managed at the namespace level. Pods can mount secrets either as files via secret volumes or as environment variables. From a security standpoint, this means that any user or service that can create a pod in a namespace can have access to any secret managed for that namespace. If you want to limit access to a secret, put it in a namespace accessible to a limited set of users or services.

When a secret is mounted into a container, it is never written to disk. It is stored in tmpfs. When the kubelet communicates with the API server, it normally uses TLS, so the secret is protected in transit.

Configuring encryption at rest

You need to pass this argument when you start the API server:

```
--encryption-provider-config
```

Here is a sample encryption config:

```
apiVersion: apiserver.config.k8s.io/v1
kind: EncryptionConfiguration
resources:
```

```

- resources:
  - secrets
providers:
- identity: {}
- aesgcm:
  keys:
    - name: key1
      secret: c2VjcmV0IGlzIHNIY3VyZQ==
    - name: key2
      secret: dGhpcyBpcyBwYXNzd29yZA==
- aescbc:
  keys:
    - name: key1
      secret: c2VjcmV0IGlzIHNIY3VyZQ==
    - name: key2
      secret: dGhpcyBpcyBwYXNzd29yZA==
- secretbox:
  keys:
    - name: key1
      secret: YWJjZGVmZ2hpamtsbW5vcHFyc3R1dnd4eXoxMjM0NTY=

```

Creating secrets

Secrets must be created before you try to create a pod that requires them. The secret must exist; otherwise, the pod creation will fail.

You can create secrets with the following command: `kubectl create secret`.

Here I create a generic secret called hush-hush, which contains two keys, a username and password:

```

$ kubectl create secret generic hush-hush --from-literal=username=tobias
--from-literal=password=cutoffs
secret/hush-hush created

```

The resulting secret is opaque:

```

$ kubectl describe secrets/hush-hush
Name:          hush-hush
Namespace:    default
Labels:        <none>
Annotations:   <none>

Type: Opaque

```

Data

====

password: 7 bytes

username: 6 bytes

You can create secrets from files using `--from-file` instead of `--from-literal`, and you can also create secrets manually if you encode the secret value as base64.

Key names inside a secret must follow the rules for DNS sub-domains (without the leading dot).

Decoding secrets

To get the content of a secret you can use `kubectl get secret`:

```
$ kubectl get secrets/hush-hush -o yaml
apiVersion: v1
data:
  password: Y3V0b2Zmcw==
  username: dG9iaWFz
kind: Secret
metadata:
  creationTimestamp: "2020-06-01T06:57:07Z"
  name: hush-hush
  namespace: default
  resourceVersion: "56655"
  selfLink: /api/v1/namespaces/default/secrets/hush-hush
  uid: 8d50c767-c705-11e9-ae89-0242ac120002
type: Opaque
```

The values are base64-encoded. You need to decode them yourself:

```
$ echo 'Y3V0b2Zmcw==' | base64 --decode
cutoffs
```

Using secrets in a container

Containers can access secrets as files by mounting volumes from the pod. Another approach is to access the secrets as environment variables. Finally, a container (given that its service account has the permission) can access the Kubernetes API directly or use `kubectl get secret`.

To use a secret mounted as a volume, the pod manifest should declare the volume and it should be mounted in the container's spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-secret
spec:
  containers:
  - name: container-with-secret
    image: g1g1/py-kube:0.2
    command: ["/bin/bash", "-c", "while true ; do sleep 10 ; done"]
    volumeMounts:
    - name: secret-volume
      mountPath: "/mnt/hush-hush"
      readOnly: true
  volumes:
  - name: secret-volume
    secret:
      secretName: hush-hush
```

The volume name (secret-volume) binds the pod volume to the mount in the container. Multiple containers can mount the same volume. When this pod is running, the username and password are available as files under /etc/hush-hush:

```
$ kubectl create -f pod-with-secret.yaml
```

```
$ kubectl exec pod-with-secret -- cat /mnt/hush-hush/username
tobias
```

```
$ kubectl exec pod-with-secret -- cat /mnt/hush-hush/password
cutoffs
```

Running a multi-user cluster

In this section, we will look briefly at the option to use a single cluster to host systems for multiple users or multiple user communities (which is also known as multi-tenancy). The idea is that those users are totally isolated and may not even be aware that they share the cluster with other users. Each user community will have its own resources, and there will be no communication between them (except maybe through public endpoints). The Kubernetes namespace concept is the ultimate expression of this idea.

The case for a multi-user cluster

Why should you run a single cluster for multiple isolated users or deployments? Isn't it simpler to just have a dedicated cluster for each user? There are two main reasons: cost and operational complexity. If you have many relatively small deployments and you want to create a dedicated cluster for each one, then you'll have a separate master node and possibly a three-node etcd cluster for each one. That can add up. Operational complexity is very important too. Managing tens, hundreds, or thousands of independent clusters is no picnic. Every upgrade and every patch needs to be applied to each cluster. Operations might fail and you'll have to manage a fleet of clusters where some of them are in a slightly different state than the others. Meta-operations across all clusters may be more difficult. You'll have to aggregate and write your tools to perform operations and collect data from all clusters.

Let's look at some use cases and requirements for multiple isolated communities or deployments:

- A platform or service provider for software-as-a-service
- Managing separate testing, staging, and production environments
- Delegating responsibility to community/deployment admins
- Enforcing resource quotas and limits on each community
- Users see only resources in their community

Using namespaces for safe multi-tenancy

Kubernetes namespaces are the perfect answer to safe multi-tenant clusters. This is not a surprise, as this was one of the design goals of namespaces.

You can easily create namespaces in addition to the built-in `kube-system` and `default`. Here is a YAML file that will create a new namespace called `custom-namespace`. All it has is a metadata item called `name`. It doesn't get any simpler:

```
apiVersion: v1
kind: Namespace
metadata:
  name: custom-namespace
```

Let's create the namespace:

```
$ kubectl create -f custom-namespace.yaml
namespace/custom-namespace created
```

```
$ kubectl get namespaces
NAME                STATUS    AGE
custom-namespace    Active    36s
default             Active    26h
kube-node-lease     Active    26h
kube-public         Active    26h
kube-system         Active    26h
```

We can see the default namespace, our new custom-namespace, and a few system namespaces prefixed with kube-.

The status field can be Active or Terminating. When you delete a namespace, it will move into the Terminating state. When the namespace is in this state, you will not be able to create new resources in this namespace. This simplifies the clean-up of namespace resources and ensures the namespace is really deleted. Without it, the replication controllers might create new pods when existing pods are deleted.

To work with a namespace, you add the `--namespace` (or `-n` for short) argument to `kubectl` commands. Here is how to run a pod in interactive mode in the custom-namespace namespace:

```
$ kubectl run trouble -it -n custom-namespace --image=g1g1/py-kube:0.2
--generator=run-pod/v1 bash
If you don't see a command prompt, try pressing enter.
root@trouble:/#
```

Listing pods in the custom-namespace returns only the pod we just launched:

```
$ kubectl get pods --namespace=custom-namespace
NAME      READY   STATUS    RESTARTS   AGE
trouble   1/1     Running   0           113s
```

Listing pods without the namespace returns the pods in the default namespace:

```
$ kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
pod-with-secret  1/1     Running   0           11h
```

Avoiding namespace pitfalls

Namespaces are great, but they can add some friction. When you use just the default namespace, you can simply omit the namespace. When using multiple namespaces, you must qualify everything with the namespace. This can add some burden, but doesn't present any danger.

However, if some users (for example, cluster administrators) can access multiple namespaces, then you're open to accidentally modifying or querying the wrong namespace. The best way to avoid this situation is to hermetically seal the namespace and require different users and credentials for each namespace, just like you should use a user account for most operations on your machine or remote machines and use root via sudo only when you have too.

In addition, you should use tools that help make it clear what namespace you're operating on (for example, shell prompt if working from the command line or listing the namespace prominently in a web interface). One of the most popular tools is kubens (available along with kubectl), available at <https://github.com/ahmetb/kubectx>.

Make sure that users that can operate on a dedicated namespace don't have access to the default namespace. Otherwise, every time they forget to specify a namespace, they'll operate quietly on the default namespace.

Summary

In this chapter, we covered the many security challenges facing developers and administrators building systems and deploying applications on Kubernetes clusters. But we also explored the many security features and the flexible plugin-based security model that provides many ways to limit, control, and manage containers, pods, and nodes. Kubernetes already provides versatile solutions to most security challenges, and it will only get better as capabilities such as AppArmor and various plugins move from alpha/beta status to general availability. Finally, we considered how to use namespaces to support multi-user communities or deployments in the same Kubernetes cluster.

In the next chapter, we will look in detail into many Kubernetes resources and concepts, and how to use them and combine them effectively. The Kubernetes object model is built on top of a solid foundation of a small number of generic concepts such as resources, manifests, and metadata. This empowers an extensible, yet surprisingly consistent, object model to expose a very diverse set of capabilities for developers and administrators.

References

- <https://www.stackrox.com/post/2019/04/setting-up-kubernetes-network-policies-a-detailed-guide/>
- <https://github.com/ahmetb/kubernetes-network-policy-recipes>
- <https://jeremievallee.com/2018/05/28/kubernetes-rbac-namespace-user.html>

5

Using Kubernetes Resources in Practice

In this chapter, we will design a fictional massive-scale platform that will challenge Kubernetes' capabilities and scalability. The Hue platform is all about creating an omniscient and omnipotent digital assistant. Hue is a digital extension of you. Hue will help you do anything, find anything, and, in many cases will do a lot on your behalf. It will obviously need to store a lot of information, integrate with many external services, respond to notifications and events, and be smart about interacting with you.

We will take the opportunity in this chapter to get to know `kubectl` and related tools a little better and explore in detail familiar resources we've seen before, such as pods, as well as new resources such as jobs. We will explore advanced scheduling and resource management. At the end of this chapter, you will have a clear picture of how impressive Kubernetes is and how it can be used as the foundation for hugely complex systems.

Designing the Hue platform

In this section, we will set the stage and define the scope of the amazing Hue platform. Hue is not Big Brother, Hue is Little Brother! Hue will do whatever you allow it to do. Hue will be able to do a lot, which might concern some people, but you get to pick how much or how little Hue can help you with. Get ready for a wild ride!

Defining the scope of Hue

Hue will manage your digital persona. It will know you better than you know yourself. Here is a list of some of the services Hue can manage and help you with:

- Search and content aggregation
- Medical – electronic health records, DNA sequencing
- Smart home
- Finance – banking, savings, retirement, investing
- Office
- Social
- Travel
- Wellbeing
- Family

Smart reminders and notifications

Let's think of the possibilities. Hue will know you, but also know your friends, the aggregate of other users across all domains. Hue will update its models in real time. It will not be confused by stale data. It will act on your behalf, present relevant information, and learn your preferences continuously. It can recommend new shows or books that you may like, make restaurant reservations based on your schedule and your family or friends, and control your house automation.

Security, identity, and privacy

Hue is your proxy online. The ramifications of someone stealing your Hue identity, or even just eavesdropping on your Hue interactions, are devastating. Potential users may even be reluctant to trust the Hue organization with their identity. Let's devise a non-trust system where users have the power to pull the plug on Hue at any time. Here are a few ideas:

- Strong identity via a dedicated device with multi-factor authorization, including multiple biometric factors
- Frequently rotating credentials
- Quick service pause and identity verification of all external services (will require original proof of identity for each provider)
- The Hue backend will interact with all external services via short-lived tokens

- Architecting Hue as a collection of loosely coupled microservices with strong compartmentalization
- GDPR compliance
- End-to-end encryption
- Avoid owning critical data (let external providers manage it)

Hue's architecture will need to support enormous variation and flexibility. It will also need to be very extensible where existing capabilities and external services are constantly upgraded, and new capabilities and external services are integrated into the platform. That level of scale calls for microservices, where each capability or service is totally independent of other services except for well-defined interfaces via standard and/or discoverable APIs.

Hue components

Before embarking on our microservice journey, let's review the types of component we need to construct for Hue.

User profile

The user profile is a major component, with lots of sub-components. It is the essence of the user, their preferences, their history across every area, and everything that Hue knows about them. The benefit you can get from Hue is affected strongly by the richness of the profile. But the more information is managed by the profile, the more damage you can suffer if the data (or part of it) is compromised.

A big piece of managing the user profile is the reports and insights that Hue will provide to the user. Hue will employ sophisticated machine learning to better understand the user and their interactions with other users and external service providers.

User graph

The user graph component models networks of interactions between users across multiple domains. Each user participates in multiple networks: social networks such as Facebook, Instagram, and Twitter; professional networks; hobby networks; and volunteer communities. Some of these networks are ad hoc and Hue will be able to structure them to benefit users. Hue can take advantage of the rich profiles it has of user connections to improve interactions even without exposing private information.

Identity

Identity management is critical, as mentioned previously, so it merits a separate component. A user may prefer to manage multiple mutually exclusive profiles with separate identities. For example, maybe users are not comfortable with mixing their health profile with their social profile at the risk of inadvertently exposing personal health information to their friends. While Hue can find useful connections for you, you may prefer to trade off capabilities for more privacy.

Authorizer

The authorizer is a critical component where the user explicitly authorizes Hue to perform certain actions or collect various data on its behalf. This involves access to physical devices, accounts of external services, and levels of initiative.

External service

Hue is an aggregator of external services. It is not designed to replace your bank, your health provider, or your social network. It will keep a lot of metadata about your activities, but the content will remain with your external services. Each external service will require a dedicated component to interact with the external service API and policies. When no API is available, Hue emulates the user by automating the browser or native apps.

Generic sensor

A big part of Hue's value proposition is to act on the user's behalf. In order to do that effectively, Hue needs to be aware of various events. For example, if Hue reserved a vacation for you but it senses that a cheaper flight is available, it can either automatically change your flight or ask you for confirmation. There is an infinite number of things to sense. To reign in sensing, a generic sensor is needed. The generic sensor will be extensible, but exposes a generic interface that the other parts of Hue can utilize uniformly even as more and more sensors are added.

Generic actuator

This is the counterpart of the generic sensor. Hue needs to perform actions on your behalf; for example, reserving a flight or a doctor's appointment. To do that, Hue needs a generic actuator that can be extended to support particular functions but can interact with other components, such as the identity manager and the authorizer, in a uniform fashion.

User learner

This is the brain of Hue. It will constantly monitor all your interactions (that you authorize) and update its model of you and other users in your networks. This will allow Hue to become more and more useful over time, predict what you need and what will interest you, provide better choices, surface more relevant information at the right time, and avoid being annoying and overbearing.

Hue microservices

The complexity of each of the components is enormous. Some of the components, such as the external service, the generic sensor, and the generic actuator, will need to operate across hundreds, thousands, or more external services that constantly change outside the control of Hue. Even the user learner needs to learn the user's preferences across many areas and domains. Microservices address this need by allowing Hue to evolve gradually and grow more isolated capabilities without collapsing under its own complexity. Each microservice interacts with generic Hue infrastructure services through standard interfaces and, optionally, with a few other services through well-defined and versioned interfaces. The surface area of each microservice is manageable and the orchestration between microservices is based on standard best practices.

Plugins

Plugins are the key to extending Hue without a proliferation of interfaces. The thing about plugins is that often, you need plugin chains that cross multiple abstraction layers. For example, if you want to add a new integration for Hue with YouTube, then you can collect a lot of YouTube-specific information – your channels, favorite videos, recommendations, and videos you have watched. To display this information to users and allow them to act on it, you need plugins across multiple components and eventually in the user interface as well. Smart design will help by aggregating categories of actions such as recommendations, selections, and delayed notifications to many different services.

The great thing about plugins is that they can be developed by anyone. Initially, the Hue development team will have to develop the plugins, but as Hue becomes more popular, external services will want to integrate with Hue and build Hue plugins to enable their service.

That will lead, of course, to a whole ecosystem of plugin registration, approval, and curation.

Data stores

Hue will need several types of data stores, and multiple instances of each type, to manage its data and metadata:

- Relational database
- Graph database
- Time-series database
- In-memory caching
- Blob storage

Due to the scope of Hue, each one of these databases will have to be clustered, scalable, and distributed.

In addition, Hue will use local storage on edge devices.

Stateless microservices

The microservices should be mostly stateless. This will allow specific instances to be started and killed quickly and migrated across the infrastructure as necessary. The state will be managed by the stores and accessed by the microservices with short-lived access tokens. Hue will store frequently accessed data in easily hydrated fast caches when appropriate.

Serverless functions

A big part of Hue's functionality per user will involve relatively short interactions with external services or other Hue services. For those activities it may not be necessary to run a full-fledged persistent microservice that needs to be scaled and managed. A more appropriate solution may be to use a serverless function that is more lightweight.

Queue-based interactions

All these microservices need to talk to each other. Users will ask Hue to perform tasks on their behalf. External services will notify Hue of various events. Queues coupled with stateless microservices provide the perfect solution. Multiple instances of each microservice will listen to various queues and respond when relevant events or requests are popped from the queue. Serverless functions may be triggered as a result of particular events too. This arrangement is very robust and easy to scale. Every component can be redundant and highly available. While each component is fallible, the system is very fault-tolerant.

A queue can be used for asynchronous RPC or request-response style interactions too, where the calling instance provides a private queue name and the response is posted to the private queue.

That said, sometimes direct service-to-service interaction (or serverless function-to-service interaction) though a well-defined interface makes more sense and simplifies the architecture.

Planning workflows

Hue often needs to support workflows. A typical workflow will take a high-level task, such as making a dentist appointment. It will extract the user's dentist's details and schedule, match it with the user's schedule, choose between multiple options, potentially confirm with the user, make the appointment, and set up a reminder. We can classify workflows into fully automatic and human workflows where humans are involved. Then there are workflows that involve spending money and might require an additional level of approval.

Automatic workflows

Automatic workflows don't require human intervention. Hue has full authority to execute all the steps from start to finish. The more autonomy the user allocates to Hue, the more effective it will be. The user will be able to view and audit all workflows, past and present.

Human workflows

Human workflows require interaction with a human. Most often it will be the user that needs to make a choice from multiple options or approve an action. But it may involve a person on another service. For example, to make an appointment with a dentist, Hue may have to get a list of available times from the secretary. In the future, Hue will be able to handle conversation with humans and possibly automate some of these workflows too.

Budget-aware workflows

Some workflows, such as paying bills or purchasing a gift, require spending money. While, in theory, Hue can be granted unlimited access to the user's bank account, most users will probably be more comfortable with setting budgets for different workflows or just making spending a human-approved activity. Potentially, the user can grant Hue access to a dedicated account or set of accounts and, based on reminders and reports, allocate more or fewer funds to Hue as needed.

Using Kubernetes to build the Hue platform

In this section, we will look at various Kubernetes resources and how they can help us build Hue. First, we'll get to know the versatile `kubectl` a little better, then we will look at how to run long-running processes in Kubernetes, exposing services internally and externally, using namespaces to limit access, launching ad hoc jobs, and mixing in non-cluster components. Obviously, Hue is a huge project, so we will demonstrate the ideas on a local cluster and not actually build a real Hue Kubernetes cluster. Consider it primarily a thought experiment. If you wish to explore building a real microservice-based distributed system on Kubernetes, check out *Hands-On Microservices with Kubernetes*.

Using `kubectl` effectively

`kubectl` is your Swiss Army knife. It can do pretty much anything around a cluster. Under the hood, `kubectl` connects to your cluster via the API. It reads your `~/.kube/config` file, which contains information necessary to connect to your cluster or clusters. The commands are divided into multiple categories:

- **Generic commands:** Deal with resources in a generic way: create, get, delete, run, apply, patch, replace, and so on
- **Cluster management commands:** Deal with nodes and the cluster at large: cluster-info, certificate, drain, and so on
- **Troubleshooting commands:** Describe, logs, attach, exec, and so on
- **Deployment commands:** Deal with deployment and scaling: rollout, scale, auto-scale, and so on
- **Settings commands:** Deal with labels and annotations: label, annotate, and so on
- **Misc commands:** Help, config, and version
- **Customization commands:** Integrate the kustomize.io capabilities into `kubectl`

You can view the configuration with Kubernetes' `config view` command.

Here is the configuration for my local k3s cluster:

```
$ k config view
apiVersion: v1
clusters:
```

```

- cluster:
  certificate-authority-data: DATA+OMITTED
  server: https://localhost:6443
  name: default
contexts:
- context:
  cluster: default
  user: default
  name: default
current-context: default
kind: Config
preferences: {}
users:
- name: default
  user:
    password: 6ce7b64ff48ac13f06af428d92b3d4bf
    username: admin

```

Understanding kubectl resource configuration files

Many kubectl operations, such as create, require a complicated hierarchical structure (since the API requires this structure). kubectl uses YAML or JSON configuration files. YAML is more concise and human-readable. Here is a YAML configuration file for creating a pod:

```

apiVersion: v1
kind: Pod
metadata:
  name: ""
  labels:
    name: ""
  namespace: ""
  annotations: []
  generateName: ""
spec:
  ...

```

ApiVersion

The very important Kubernetes API keeps evolving and can support different versions of the same resource via different versions of the API.

Kind

Kind tells Kubernetes what type of resource it is dealing with; in this case, pod. This is always required.

Metadata

A lot of information that describes the pod and where it operates:

- **Name:** Identifies the pod uniquely within its namespace
- **Labels:** Multiple labels can be applied
- **Namespace:** The namespace the pod belongs to
- **Annotations:** A list of annotations available for query

Spec

Spec is a pod template that contains all the information necessary to launch a pod. It can be quite elaborate, so we'll explore it in multiple parts:

```
spec:
  containers: [
    ...
  ],
  "restartPolicy": "",
  "volumes": []
```

Container spec

The pod spec's containers section is a list of container specs. Each container spec has the following structure:

```
name: "",
image: "",
command: [""],
args: [""],
env:
  - name: "",
```

```

    value: ""

    imagePullPolicy: "",
    ports:
      - containerPort: 0,
        name: "",
        protocol: ""
    resources:
      cpu: ""
      memory: ""

```

Each container has an image, a command that, if specified, replaces the Docker image command. It also has arguments and environment variables. Then, there are of course the image pull policy, ports, and resource limits. We covered those in earlier chapters.

Deploying long-running microservices in pods

Long-running microservices should run in pods and be stateless. Let's look at how to create pods for one of Hue's microservices. Later, we will raise the level of abstraction and use a deployment.

Creating pods

Let's start with a regular pod configuration file for creating a Hue learner internal service. This service doesn't need to be exposed as a public service and it will listen to a queue for notifications and store its insights in some persistent storage.

We need a simple container that will run in the pod. Here is possibly the simplest Docker file ever, which will simulate the Hue learner:

```

FROM busybox
CMD ash -c "echo 'Started...'; while true ; do sleep 10 ; done"

```

It uses the busybox base image, prints to standard output Started..., and then goes into an infinite loop, which is, by all accounts, long-running.

I have built two Docker images tagged as `g1g1/hue-learn:0.3` and `g1g1/hue-learn:v0.4` and pushed them to the Docker Hub registry (g1g1 is my username):

```

$ docker build . -t g1g1/hue-learn:0.3
$ docker build . -t g1g1/hue-learn:0.4

```

```
$ docker push g1g1/hue-learn:0.3
$ docker push g1g1/hue-learn:0.4
```

Now these images are available to be pulled into containers inside of Hue's pods.

We'll use YAML here because it's more concise and human-readable. Here are the boilerplate and metadata labels:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
  labels:
    app: hue
    service: learner
    runtime-environment: production
    tier: internal-service
```

The reason I use an annotation for the version and not a label is that labels are used to identify the set of pods in the deployment. Modifying labels is not allowed.

Next comes the important `containers` spec, which defines for each container the mandatory name and image:

```
spec:
  containers:
  - name: hue-learner
    image: g1g1/hue-learn:0.3
```

The `resources` section tells Kubernetes the resource requirements of the container, which allows for more efficient and compact scheduling and allocations. Here, the container requests 200 milli-cpu units (0.2 core) and 256 MiB (2 to the power of 28 bytes):

```
resources:
  requests:
    cpu: 200m
    memory: 256Mi
```

The `environment` section allows the cluster administrator to provide environment variables that will be available to the container. Here it tells it to discover the queue and the store via `dns`. In a testing environment, it may use a different discovery method:

```
env:
- name: DISCOVER_QUEUE
  value: dns
- name: DISCOVER_STORE
  value: dns
```

Decorating pods with labels

Labeling pods wisely is key for flexible operations. It lets you evolve your cluster live, organize your microservices into groups you can operate on uniformly, and drill down on the fly to observe different subsets.

For example, our Hue learner pod has the following labels:

- **runtime-environment:** production
- **tier:** internal-service

The runtime-environment label allows performing global operations on all pods that belong to a certain environment. The tier label can be used to query all pods that belong to a particular tier. These are just examples; your imagination is the limit here.

Here is how to list the labels with the get pods command:

```
$ kubectl get po -n kube-system --show-labels
```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
coredns-b7464766c-s4z28	1/1	Running	3	15d	k8s-app=kube-
dns,pod-template-hash=b7464766c					
svclb-traefik-688zv	2/2	Running	6	15d	
app=svclb-traefik,controller-revision-hash=66fd644d6,pod-template-generation=1,svccontroller.k3s.cattle.io/svcname=traefik					
svclb-traefik-hfk8t	2/2	Running	6	15d	
app=svclb-traefik,controller-revision-hash=66fd644d6,pod-template-generation=1,svccontroller.k3s.cattle.io/svcname=traefik					
svclb-traefik-kp9wh	2/2	Running	6	15d	
app=svclb-traefik,controller-revision-hash=66fd644d6,pod-template-generation=1,svccontroller.k3s.cattle.io/svcname=traefik					
svclb-traefik-sgmbg	2/2	Running	6	15d	
app=svclb-traefik,controller-revision-hash=66fd644d6,pod-template-generation=1,svccontroller.k3s.cattle.io/svcname=traefik					
traefik-56688c4464-c4sfq	1/1	Running	3	15d	
app=traefik,chart=traefik-1.64.0,heritage=Tiller,pod-template-hash=56688c4464,release=traefik					

Now, if you want to filter and list only the pods of the `traefik` app type:

```
$ kubectl get po -n kube-system -l app=traefik
```

NAME	READY	STATUS	RESTARTS	AGE
traefik-56688c4464-c4sfq	1/1	Running	3	15d

Deploying long-running processes with deployments

In a large-scale system, pods should never be just created and let loose. If a pod dies unexpectedly for whatever reason, you want another one to replace it to maintain overall capacity. You can create replication controllers or replica sets yourself, but that leaves the door open to mistakes, as well as the possibility of partial failure. It makes much more sense to specify how many replicas you want when you launch your pods in a declarative manner. This is what Kubernetes deployments are for.

Let's deploy three instances of our Hue learner microservice with a Kubernetes deployment resource. Note that deployment objects became stable at Kubernetes 1.9:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-learn
  labels:
    app: hue
    service: learn
spec:
  replicas: 3
  selector:
    matchLabels:
      app: hue
      service: learn
  template:
    metadata:
      labels:
        app: hue
    spec:
      <same spec as in the pod template>
```

The pod spec is identical to the spec section from the pod configuration file previously.

Let's create the deployment and check its status:

```
$ kubectl create -f hue-learn-deployment.yaml
deployment.apps/hue-learn created
```

```
$ kubectl get deployment hue-learn
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
hue-learn	3/3	3	3	32s

```
$ kubectl get pods -l app=hue
```

NAME	READY	STATUS	RESTARTS	AGE
hue-learn-558f5c45cd-fbvpj	1/1	Running	0	81s
hue-learn-558f5c45cd-s6vkk	1/1	Running	0	81s
hue-learn-558f5c45cd-tdlpq	1/1	Running	0	81s

You can get a lot more information about the deployment using the `kubectl describe` command.

Updating a deployment

The Hue platform is a large and ever-evolving system. You need to upgrade constantly. Deployments can be updated to roll out updates in a painless manner. You change the pod template to trigger a rolling update fully managed by Kubernetes.

Currently, all the pods are running with version 0.3:

```
$ kubectl get pods -o jsonpath='{.items[*].spec.containers[0].image}'
g1g1/hue-learn:0.3
g1g1/hue-learn:0.3
g1g1/hue-learn:0.3
```

Let's update the deployment to upgrade to version 0.4. Modify the image version in the deployment file. Don't modify labels; it will cause an error. Save it to `hue-learn-deployment-0.4.yaml`. Then we can use the `apply` command to upgrade the version and verify that the pods now run 0.4:

```
$ kubectl apply -f hue-learn-deployment-0.4.yaml
deployment "hue-learn" updated
```


Note that it can take several minutes to see the following output due to the nature of the rolling update operation:

```
$ kubectl get pods -o jsonpath='{.items[*].spec.containers[0].image}'
g1g1/hue-learn:0.4
g1g1/hue-learn:0.4
g1g1/hue-learn:0.4
```

Note that new pods are created and the original 0.3 pods are terminated in a rolling update manner:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE	IP
hue-learn-558f5c45cd-fbvpj	1/1	Terminating	0	8m7s	
10.42.3.15 k3d-k3s-default-server					
hue-learn-558f5c45cd-s6vkk	0/1	Terminating	0	8m7s	
10.42.0.7 k3d-k3s-default-worker-0					
hue-learn-558f5c45cd-tdlpq	0/1	Terminating	0	8m7s	
10.42.2.15 k3d-k3s-default-worker-2					
hue-learn-5c9bb545d9-lggk7	1/1	Running	0	38s	
10.42.2.16 k3d-k3s-default-worker-2					
hue-learn-5c9bb545d9-pwflv	1/1	Running	0	31s	
10.42.1.10 k3d-k3s-default-worker-1					
hue-learn-5c9bb545d9-q25hl	1/1	Running	0	35s	
10.42.0.8 k3d-k3s-default-worker-0					

Separating internal and external services

Internal services are services that are accessed directly only by other services or jobs in the cluster (or administrators that log in and run ad hoc tools). In some cases, internal services are not accessed at all, and just perform their function and store their results in a persistent store that other services access in a decoupled way.

But some services need to be exposed to users or external programs. Let's look at a fake Hue service that manages a list of reminders for a user. It doesn't really do much – just returns a fixed list of reminders – but we'll use it to illustrate how to expose services. I already pushed a hue-reminders image to Docker Hub:

```
docker push g1g1/hue-reminders:3.0
```

Deploying an internal service

Here is the deployment, which is very similar to the Hue-learner deployment, except that I dropped the annotations, env, and resources sections, kept just one or two labels to save space, and added a ports section to the container. That's crucial because a service must expose a port through which other services can access it:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hue
      service: reminders
  template:
    metadata:
      name: hue-reminders
    labels:
      app: hue
      service: reminders
    spec:
      containers:
        - name: hue-reminders
          image: g1g1/hue-reminders:3.0
          ports:
            - containerPort: 8080
```

When we run the deployment, two hue-reminders pods are added to the cluster:

```
$ kubectl create -f hue-reminders-deployment.yaml
deployment.apps/hue-reminders created
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hue-learn-5c9bb545d9-lggk7	1/1	Running	0	22m
hue-learn-5c9bb545d9-pwflv	1/1	Running	0	22m
hue-learn-5c9bb545d9-q25h1	1/1	Running	0	22m
hue-reminders-9b7d65d86-4kf5t	1/1	Running	0	7s
hue-reminders-9b7d65d86-tch4w	1/1	Running	0	7s

OK. The pods are running. In theory, other services can look up or be configured with their internal IP address and just access them directly because they are all in the same network address space. But this doesn't scale. Every time a reminder's pod dies and is replaced by a new one, or when we just scale up the number of pods, all the services that access these pods must know about it. Kubernetes services solve this issue by providing a single stable access point to all the pods. Here is the service definition:

```
apiVersion: v1
kind: Service
metadata:
  name: hue-reminders
  labels:
    app: hue
    service: reminders
spec:
  ports:
    - port: 8080
      protocol: TCP
  selector:
    app: hue
    service: reminders
```

The service has a selector that determines the backing pods by their matching labels. It also exposes a port, which other services will use to access it (it doesn't have to be the same port as the container's port).

The protocol field can be one of TCP, UDP, and since Kubernetes 1.12 also SCTP (if the feature gate is enabled).

Creating the Hue-reminders service

Let's create the service and explore it a little bit:

```
$ kubectl create -f hue-reminders-service.yaml
service/hue-reminders created
```

```
$ kubectl describe svc hue-reminders
Name:                hue-reminders
Namespace:           default
Labels:              app=hue
                   service=reminders
Annotations:         <none>
```

```

Selector:      app=hue,service=reminders
Type:          ClusterIP
IP:            10.43.166.58
Port:          <unset> 8080/TCP
TargetPort:    8080/TCP
Endpoints:     10.42.1.12:8080,10.42.2.17:8080
Session Affinity: None
Events:        <none>

```

The service is up and running. Other pods can find it through environment variables or DNS. The environment variables for all services are set at pod creation time. That means that if a pod is already running when you create your service, you'll have to kill it and let Kubernetes recreate it with the environment variables (you create your pods via a deployment, right?):

```

$ kubectl exec hue-learn-5c9bb545d9-w8hrr -- printenv | grep HUE_REMINDERS_
SERVICE
HUE_REMINDERS_SERVICE_HOST=10.43.166.58
HUE_REMINDERS_SERVICE_PORT=8080

```

But using DNS is much simpler. Your service DNS name is:

```

<service name>.<namespace>.svc.cluster.local

$ kubectl exec hue-learn-5c9bb545d9-w8hrr -- nslookup hue-reminders.
default.svc.cluster.local
Server:      10.43.0.10
Address 1:  10.43.0.10 kube-dns.kube-system.svc.cluster.local

Name:        hue-reminders.default.svc.cluster.local
Address 1:  10.43.247.147 hue-reminders.default.svc.cluster.local

```

Now, all the services in the default namespace can access the hue-reminders service though its service endpoint and port 8080:

```

$ kubectl exec hue-learn-5c9bb545d9-w8hrr -- wget -q -O - hue-reminders.
default.svc.cluster.local:8080
Dentist appointment at 3pm
Dinner at 7pm

```

Yes, at the moment hue-reminders always returns the same two reminders:

```

Dentist appointment at 3pm
Dinner at 7pm

```

Exposing a service externally

The service is accessible inside the cluster. If you want to expose it to the world, Kubernetes provides three ways to do it:

- Configure NodePort for direct access
- Configure a cloud load balancer if you run it in a cloud environment
- Configure your own load balancer if you run on bare metal

Before you configure a service for external access, you should make sure it is secure. The Kubernetes documentation has a good example that covers all the gory details here:

<https://github.com/kubernetes/examples/blob/master/staging/https-nginx/README.md>.

We've already covered the principles in *Chapter 4, Securing Kubernetes*.

Here is the spec section of the hue-reminders service when exposed to the world through NodePort:

```
spec:
  type: NodePort
  ports:
  - port: 8080
    targetPort: 8080
    protocol: TCP
    name: http

  - port: 443
    protocol: TCP
    name: https
  selector:
    app: hue-reminders
```

The main downside of exposing services though NodePort is that the port numbers are shared across all services and you must coordinate them globally across your entire cluster to avoid conflicts.

But there are other reasons that you may want to avoid exposing a Kubernetes service directly, and you may prefer to use an Ingress resource in front of the service.

Ingress

Ingress is a Kubernetes configuration object that lets you expose a service to the outside world and takes care of a lot of details. It can do the following:

- Provide an externally visible URL to your service
- Load balance traffic
- Terminate SSL
- Provide name-based virtual hosting

To use Ingress, you must have an Ingress controller running in your cluster. Note that Ingress was introduced in Kubernetes 1.1, but it is still in beta and has many limitations. If you're running your cluster on GKE, you're probably OK. Otherwise, proceed with caution. One of the current limitations of the Ingress controller is that it isn't built for scale. As such, it is not a good option for the Hue platform yet. We'll cover the Ingress controller in greater detail in *Chapter 10, Exploring Advanced Networking*.

Here is what an Ingress resource looks like:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - http:
        paths:
          - path: /testpath
            backend:
              serviceName: test
              servicePort: 80
```

Note the annotation, which hints that it is an Ingress object that works with the Nginx Ingress controller. There are many other Ingress controllers and they typically use annotations to encode information they need that is not captured by the Ingress object itself and its rules.

Other Ingress controllers include:

- Traefik
- Gloo
- Contour
- AWS ALB Ingress controller
- HAProxy Ingress
- Voyager

Advanced scheduling

One of the strongest suits of Kubernetes is its powerful yet flexible scheduler. The job of the scheduler, put simply, is to choose nodes to run newly created pods. In theory the scheduler could even move existing pods around between nodes, but in practice it doesn't do that at the moment and instead leaves this functionality for other components.

By default, the scheduler follows several guiding principles, including:

- Split pods from the same replica set or stateful set across nodes
- Schedule pods to nodes that have enough resources to satisfy the pod requests
- Balance out the overall resource utilization of nodes

This is pretty good default behavior, but sometimes you may want better control over specific pod placement. Kubernetes 1.6 introduced several advanced scheduling options that give you fine-grained control over which pods are scheduled or not scheduled on which nodes as well as which pods are to be scheduled together or separately.

Let's review these mechanisms in the context of Hue.

Node selector

The node selector is pretty simple. A pod can specify which nodes it wants to be scheduled on in its spec. For example, the `trouble-shooter` pod has a `nodeSelector` that specifies the `kubernetes.io/hostname` label of the `worker-2` node:

```
apiVersion: v1
kind: Pod
metadata:
```

```

name: trouble-shooter
labels:
  role: trouble-shooter
spec:
  nodeSelector:
    kubernetes.io/hostname: k3d-k3s-default-worker-2
  containers:
  - name: trouble-shooter
    image: g1g1/py-kube:0.2
    command: ["bash"]
    args: ["-c", "echo started...; while true ; do sleep 1 ; done"]

```

When creating this pod it is indeed scheduled to the worker-2 node:

```
$ k apply -f trouble-shooter.yaml
pod/trouble-shooter created
```

```
$ k get po trouble-shooter -o jsonpath='{.spec.nodeName}'
k3d-k3s-default-worker-2
```

Taints and tolerations

You can taint a node in order to prevent pods from being scheduled on this node. This can be useful, for example, if you don't want pods to be scheduled on your master nodes. Tolerations allow pods to declare that they can "tolerate" a specific node taint and then these pods can be scheduled on the tainted node. A node can have multiple taints and a pod can have multiple tolerations. A taint is a triplet: key, value, effect. The key and value are used to identify the taint. The effect is one of:

- NoSchedule (no pods will be scheduled to the node unless it tolerates the taint)
- PreferNoSchedule (soft version of NoSchedule; the scheduler will attempt not to schedule pods that don't tolerate the taint)
- NoExecute (no new pods will be scheduled, but also existing pods that don't tolerate the taint will be evicted)

Currently, there is a hue-learn pod that runs on the master node (k3d-k3s-default-server):

```
$ kubectl get po -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
hue-learn-5c9bb545d9-dk4c4	1/1	Running	0	7h40m	

10.42.3.17	k3d-k3s-default-server	hue-learn-5c9bb545d9-sqx28	1/1	Running	0	7h40m
10.42.2.18	k3d-k3s-default-worker-2	hue-learn-5c9bb545d9-w8hrr	1/1	Running	0	7h40m
10.42.0.11	k3d-k3s-default-worker-0	hue-reminders-6f9f54d8f-hwjwd	1/1	Running	0	3h51m
10.42.0.13	k3d-k3s-default-worker-0	hue-reminders-6f9f54d8f-p4z8z	1/1	Running	0	3h51m
10.42.1.14	k3d-k3s-default-worker-1					

Let's taint our master node:

```
$ k taint nodes k3d-k3s-default-server master=true:NoExecute
node/k3d-k3s-default-server tainted
```

We can now review the taint:

```
$ k get nodes k3d-k3s-default-server -o jsonpath='{.spec.taints[0]}'
map[effect:NoExecute key:master value:true]
```

Yeah, it worked! there are now no pods scheduled on the master node. The pod on the master was terminated and a new pod (hue-learn-5c9bb545d9-nn4xk) is now running on worker-1:

```
$ kubectl get po -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
hue-learn-5c9bb545d9-nn4xk	1/1	Running	0	3m46s	
10.42.1.15 k3d-k3s-default-worker-1	1/1	Running	0	9h	
10.42.2.18 k3d-k3s-default-worker-2	1/1	Running	0	9h	
10.42.0.11 k3d-k3s-default-worker-0	1/1	Running	0	6h	
10.42.0.13 k3d-k3s-default-worker-0	1/1	Running	0	6h	
10.42.1.14 k3d-k3s-default-worker-1	1/1	Running	0	16m	
trouble-shooter	1/1	Running	0		
10.42.2.20 k3d-k3s-default-worker-2					

To allow pods to tolerate the taint, add a toleration to their spec, such as:

```
tolerations:
- key: "master"
  operator: "Equal"
  value: "true"
  effect: "NoSchedule"
```

Node affinity and anti-affinity

Node affinity is a more sophisticated form of `nodeSelector`. It has three main advantages:

- Rich selection criteria (`nodeSelector` is just AND of exact matches on the labels)
- Rules can be soft
- You can achieve anti-affinity using operators such as `NotIn` and `DoesNotExist`

Note that if you specify both `nodeSelector` and `nodeAffinity` then the pod will be scheduled only to a node that satisfies both requirements.

For example, if we add the following section to our trouble-shooter pod it will not be able to run on any node because it conflicts with `nodeSelector`:

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: NotIn
          values:
            - k3d-k3s-default-worker-2
```

Pod affinity and anti-affinity

Pod affinity and anti-affinity provide yet another avenue for managing where your workloads run. All the methods we discussed so far – node selectors, taints/tolerations, node affinity/anti-affinity – were about assigning pods to nodes. But pod affinity is about the relationships between different pods. Pod affinity has several other concepts associated with it: namespacing (since pods are namespaced), topology zone (node, rack, cloud provider zone, cloud provider region), and weight (for preferred scheduling). A simple example is if you want hue-reminders to always be scheduled with a trouble-shooter pod. Let's see how to define it in the pod template spec of the hue-reminders deployment:

```
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
```

```
- key: role
  operator: In
  values:
  - trouble-shooter
topologyKey: failure-domain.beta.kubernetes.io/zone
```

Then after re-deploying hue-reminders, all the hue-reminders pods are scheduled to run on worker-2 next to the trouble-shooter pod:

```
$ k get po -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP
hue-learn-5c9bb545d9-nn4xk	1/1	Running	0	156m	
10.42.1.15 k3d-k3s-default-worker-1					
hue-learn-5c9bb545d9-sqx28	1/1	Running	0	12h	
10.42.2.18 k3d-k3s-default-worker-2					
hue-learn-5c9bb545d9-w8hrr	1/1	Running	0	12h	
10.42.0.11 k3d-k3s-default-worker-0					
hue-reminders-5cb9b845d8-kck5d	1/1	Running	0	14s	
10.42.2.24 k3d-k3s-default-worker-2					
hue-reminders-5cb9b845d8-kpvx5	1/1	Running	0	14s	
10.42.2.23 k3d-k3s-default-worker-2					
trouble-shooter	1/1	Running	0	14m	
10.42.2.21 k3d-k3s-default-worker-2					

Using namespaces to limit access

The Hue project is moving along nicely, and we have a few hundred microservices and about 100 developers and DevOps engineers working on it. Groups of related microservices emerge, and you notice that many of these groups are pretty autonomous. They are completely oblivious to the other groups. Also, there are some sensitive areas such as health and finance that you want to control access to more effectively. Enter namespaces.

Let's create a new service, Hue-finance, and put it in a new namespace called restricted.

Here is the YAML file for the new restricted namespace:

```
kind: Namespace
apiVersion: v1
metadata:
  name: restricted
  labels:
    name: restricted
```

We can create it as usual:

```
$ kubectl create -f restricted-namespace.yaml
namespace "restricted" created
```

Once the namespace has been created, we can configure a context for the namespace. This will allow restricting access just to this namespace to specific users:

```
$ kubectl config set-context restricted --namespace=restricted
--cluster=default --user=default
Context "restricted" set.
```

```
$ kubectl config use-context restricted
Switched to context "restricted".
```

Let's check our cluster configuration:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://localhost:6443
    name: default
contexts:
- context:
    cluster: default
    user: default
    name: default
- context:
    cluster: default
    namespace: restricted
    user: default
    name: restricted
current-context: restricted
kind: Config
preferences: {}
users:
- name: default
  user:
    password: <REDACTED>
    username: admin
```

As you can see, there are two contexts now and the current context is restricted. If we wanted to, we could even create dedicated users with their own credentials that are allowed to operate in the restricted namespace. This is not necessary in this case since we are the cluster admin.

Now, in this empty namespace, we can create our hue-finance service, and it will be on its own:

```
$ kubectl create -f hue-finance-deployment.yaml
deployment.apps/hue-learn created
```

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
hue-finance-7d4b84cc8d-gcjnz	1/1	Running	0	6s
hue-finance-7d4b84cc8d-tqvr9	1/1	Running	0	6s
hue-finance-7d4b84cc8d-zthdr	1/1	Running	0	6s

You don't have to switch contexts. You can also use the `--namespace=<namespace>` and `--all-namespaces` command-line switches, but when operating for a while in the same non-default namespace it's nice to set the context to that namespace.

Using kustomization for hierarchical cluster structures

This is not a typo. Kubectl recently incorporated the functionality of kustomize (<https://kustomize.io/>). It is a way to configure Kubernetes without templates. There was a lot of drama about the way the kustomize functionality was integrated into kubectl itself, since there are other options and it was an open question if kubectl should be that opinionated. But that's all in the past. The bottom line is that kubectl `apply -k` unlocks a treasure trove of configuration options. Let's understand what problem it helps us to solve and take advantage of it to help us manage Hue.

Understanding the basics of kustomize

Kustomize was created as a response to template-heavy approaches like Helm to configure and customize Kubernetes clusters. It is designed around the principle of declarative application management. It takes a valid Kubernetes YAML manifest (base) and specializes it or extends it by overlaying additional YAML patches (overlays). Overlays depend on their bases. All files are valid YAML files. There are no placeholders.

A `kustomization.yaml` file controls the process. Any directory that contains a `kustomization.yaml` file is called a root. For example:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: staging
commonLabels:
  environment: staging
bases:
- ../base
patchesStrategicMerge:
- hue-learn-patch.yaml

resources:
- namespace.yaml
```

kustomize can work well in a GitOps environment where different kustomizations live in a Git repo and changes to the bases, overlays, or `kustomization.yaml` files trigger a deployment.

One of the best use cases for kustomize is organizing your system into multiple namespaces such as staging and production. Let's restructure the Hue platform deployment manifests.

Configuring the directory structure

First, we need a base directory that will contain the commonalities of all the manifests. Then we will have an `overlays` directory that contains `staging` and `production` sub-directories:

\$ tree

```
.
├── base
│   ├── hue-learn.yaml
│   └── kustomization.yaml
└── overlays
    ├── production
    │   ├── kustomization.yaml
    │   └── namespace.yaml
    └── staging
        ├── hue-learn-patch.yaml
        ├── kustomization.yaml
        └── namespace.yaml
```

The `hue-learn.yaml` file in the base directory is just an example. There may be many files there. Let's review it quickly:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
  labels:
    tier: internal-service
spec:
  containers:
  - name: hue-learner
    image: g1g1/hue-learn:0.3
    resources:
      requests:
        cpu: 200m
        memory: 256Mi
    env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
```

It is very similar to the manifest we created earlier, but it doesn't have the `app: hue` label. It is not necessary because the label is provided by the `kustomization.yaml` file as a common label for all the listed resources:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
commonLabels:
  app: hue

resources:
- hue-learn.yaml
```

Applying kustomizations

We can observe the results by running the `kubectl kustomize` command on the base directory. You can see that the common label `app: hue` was added:

```
$ kubectl kustomize base
```

```
apiVersion: v1
```

```

kind: Pod
metadata:
  labels:
    app: hue
    tier: internal-service
    name: hue-learner
spec:
  containers:
  - env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
    image: g1g1/hue-learn:0.3
    name: hue-learner
    resources:
      requests:
        cpu: 200m
        memory: 256Mi

```

In order to actually deploy the kustomization, we can run `kubectl -k apply`. But the base is not supposed to be deployed on its own. Let's dive into the `overlays/staging` directory and examine it.

The `namespace.yaml` file just creates the staging namespace. It will also benefit from all the kustomizations, as we'll soon see:

```

apiVersion: v1
kind: Namespace
metadata:
  name: staging

```

The `kustomization.yaml` file adds the common label `environment: staging`. It depends on the base directory and adds the `namespace.yaml` file to the resources list (which already includes the `hue-learn.yaml` from the base directory):

```

apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: staging
commonLabels:
  environment: staging
bases:
- ../../base

```



```
patchesStrategicMerge:
  - hue-learn-patch.yaml

resources:
  - namespace.yaml
```

But that's not all. The most interesting part of kustomization is patching.

Patching

Patches add or replace parts of manifests. They never remove existing resources or parts of resources. The `hue-learn-patch.yaml` updates the image from `g1g1/hue-learn:0.3` to `g1g1/hue-learn:0.4`:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-learner
spec:
  containers:
    - name: hue-learner
      image: g1g1/hue-learn:0.4
```

This is a strategic merge. Kustomize supports another type of patch called `JsonPatches6902`. It is based on RFC 6902. It is often more concise than a strategic merge. Note that since JSON is a subset of YAML, we can use YAML syntax for JSON 6902 patches. Here is the same patch of changing the image version:

```
- op: replace
  path: /spec/containers/0/image
  value: g1g1/hue-learn:0.4
```

Kustomizing the entire staging namespace

Here is what kustomize generates when it is running on the `overlays/staging` directory:

```
$ kubextl kustomize overlays/staging
```

```
apiVersion: v1
kind: Namespace
metadata:
```

```

  labels:
    environment: staging
  name: staging
---
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: hue
    environment: staging
    tier: internal-service
  name: hue-learner
  namespace: staging
spec:
  containers:
  - env:
    - name: DISCOVER_QUEUE
      value: dns
    - name: DISCOVER_STORE
      value: dns
    image: g1g1/hue-learn:0.4
    name: hue-learner
    resources:
      requests:
        cpu: 200m
        memory: 256Mi

```

Note that the namespace didn't inherit the `app: hue` label from the base, but only the `environment: staging` label from its local customization file. The `hue-learn` pod on the other hand got all labels as well the namespace designation.

It's time to deploy it to the cluster:

```

$ kubectl apply -k overlays/staging/
namespace/staging created
pod/hue-learner created

```

Now, we can review the pod in the newly created staging namespace:

```

$ kubectl get po -n staging

```

NAME	READY	STATUS	RESTARTS	AGE
hue-learner	1/1	Running	0	8s

Launching jobs

Hue has evolved and has a lot of long-running processes deployed as microservices, but it also has a lot of tasks that run, accomplish some goal, and exit. Kubernetes supports this functionality via the Job resource. A Kubernetes job manages one or more pods and ensures that they run until they are successful. If one of the pods managed by the job fails or is deleted, then the job will run a new pod until it succeeds.

There are also many serverless or function-as-a-service solutions for Kubernetes, but they are built on top of native Kubernetes. We will dedicate a whole chapter to serverless computing.

Here is a job that runs a Python process to compute the factorial of 5 (hint: it's 120):

```
apiVersion: batch/v1
kind: Job
metadata:
  name: factorial5
spec:
  template:
    metadata:
      name: factorial5
    spec:
      containers:
      - name: factorial5
        image: g1g1/py-kube:0.2
        command: ["python",
                  "-c",
                  "import math; print(math.factorial(5))"]
      restartPolicy: Never
```

Note that the restartPolicy must be either Never or OnFailure. The default value – Always – is invalid because a job shouldn't restart after a successful completion.

Let's start the job and check its status:

```
$ kubectl create -f factorial-job.yaml
job.batch/factorial5 created
```

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
factorial5	1/1	2s	2m53s

The pods of completed tasks are displayed with a status of Completed:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
factorial5-tf9qb	0/1	Completed	0	26m
hue-learn-5c9bb545d9-nn4xk	1/1	Running	3	2d11h
hue-learn-5c9bb545d9-sqx28	1/1	Running	3	2d21h
hue-learn-5c9bb545d9-w8hrr	1/1	Running	3	2d21h
hue-reminders-5cb9b845d8-kck5d	1/1	Running	3	2d8h
hue-reminders-5cb9b845d8-kpvx5	1/1	Running	3	2d8h
trouble-shooter	1/1	Running	3	2d9h

The factorial5 pod has a status of Completed. Let's check out its output in the logs:

```
$ kubectl logs factorial5-tf9qb
120
```

Running jobs in parallel

You can also run a job with parallelism. There are two fields in the spec, called completions and parallelism. completions is set to 1 by default. If you want more than one successful completion, then increase this value. parallelism determines how many pods to launch. A job will not launch more pods than needed for successful completion, even if the parallelism value is greater.

Let's run another job that just sleeps for 20 seconds until it has three successful completions. We'll use a parallelism factor of six, but only three pods will be launched:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: sleep20
spec:
  completions: 3
  parallelism: 6
  template:
    metadata:
      name: sleep20
    spec:
      containers:
      - name: sleep20
        image: g1g1/py-kube:0.2
```

```
command: ["python",  
          "-c",  
          "import time; print('started...');  
          time.sleep(20); print('done.')"]  
restartPolicy: Never
```

We can now see that all jobs completed, and the pods are not ready because they already did the job:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
sleep20-2mb7g	0/1	Completed	0	17m
sleep20-74pwh	0/1	Completed	0	15m
sleep20-txgpz	0/1	Completed	0	15m

Cleaning up completed jobs

When a job completes, it sticks around – and its pods, too. This is by design, so you can look at logs or connect to pods and explore. But normally, when a job has completed successfully, it is not needed anymore. It's your responsibility to clean up completed jobs and their pods. The easiest way is to simply delete the job object, which will delete all the pods too:

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
factorial5	1/1	2s	6h59m
sleep20	3/3	3m7s	5h54m

```
$ kubectl delete job factorial5  
job.batch "factorial5" deleted
```

```
$ kubectl delete job sleep20  
job.batch "sleep20" deleted
```

Scheduling cron jobs

Kubernetes cron jobs are jobs that run for a specified time, once or repeatedly. They behave as regular Unix cron jobs specified in the `/etc/crontab` file.

In Kubernetes 1.4 they were known as a `ScheduledJob`. But in Kubernetes 1.5, the name was changed to `CronJob`. Starting with Kubernetes 1.8 the `CronJob` resource is enabled by default in the API server and there is no need to pass a `--runtime-config` flag anymore, but it's still in beta. Here is the configuration to launch a cron job every minute to remind you to stretch. In the schedule, you may replace the `*` with `?`:

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: cron-demo
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        metadata:
          labels:
            name: cron-demo
        spec:
          containers:
            - name: cron-demo
              image: g1g1/py-kube:0.2
              args:
                - python
                - -c
                - from datetime import datetime; print('[{}] CronJob demo
here...'.format(datetime.now()))
          restartPolicy: OnFailure
```

In the pod spec, under the job template, I added a label name: `cron-demo`. The reason is that cron jobs and their pods are assigned names with a random prefix by Kubernetes. The label allows you to easily discover all the pods of a particular cron job.

See the following command lines:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
cron-demo-1568250120-7jrqs	0/1	Completed	0	3m
cron-demo-1568250180-sw5qq	0/1	Completed	0	2m
cron-demo-1568250240-mmfmz	0/1	Completed	0	1m

Note that each invocation of a cron job launches a new job object with a new pod:

```
$ kubectl get jobs
```

NAME	COMPLETIONS	DURATION	AGE
cron-demo-1568244780	1/1	2s	1m
cron-demo-1568250060	1/1	3s	88s
cron-demo-1568250120	1/1	3s	38s

As usual, you can check the output of the pod of a completed cron job using the logs command:

```
$ kubectl logs cron-demo-1568250240-mmfmz  
[2020-06-01 01:04:03.245204] CronJob demo here...
```

When you delete a cron job it stops scheduling new jobs, then deletes all the existing job objects and all the pods it created.

You can use the designated label (name=cron-demo in this case) to locate all the job objects launched by the cron job. You can also suspend a cron job so it doesn't create more jobs without deleting completed jobs and pods. You can also manage previous jobs by setting in the spec history limits: `.spec.successfulJobsHistoryLimit` and `.spec.failedJobsHistoryLimit`.

Mixing non-cluster components

Most real-time system components in the Kubernetes cluster will communicate with out-of-cluster components. Those could be completely external third-party services accessible through some API, but can also be internal services running in the same local network that, for various reasons, are not part of the Kubernetes cluster.

There are two categories here: inside the cluster network and outside the cluster network. Why is the distinction important?

Outside-the-cluster-network components

These components have no direct access to the cluster. They can only access it through APIs, externally visible URLs, and exposed services. These components are treated just like any external user. Often, cluster components will just use external services, which pose no security issue. For example, in a previous job we had a Kubernetes cluster that reported exceptions to a third-party service (<https://sentry.io/welcome/>). It was one-way communication from the Kubernetes cluster to the third-party service.

Inside-the-cluster-network components

These are components that run inside the network but are not managed by Kubernetes. There are many reasons to run such components. They could be legacy applications that have not been kubernetesized yet, or some distributed data store that is not easy to run inside Kubernetes. The reason to run these components inside the network is for performance, and to have isolation from the outside world so traffic between these components and pods can be more secure. Being part of the same network ensures low latency, and the reduced need for authentication is both convenient and can avoid authentication overhead.

Managing the Hue platform with Kubernetes

In this section, we will look at how Kubernetes can help operate a huge platform such as Hue. Kubernetes itself provides a lot of capabilities to orchestrate pods and manage quotas and limits, detecting and recovering from certain types of generic failures (hardware malfunctions, process crashes, unreachable services). But in a complicated system such as Hue, pods and services may be up and running but in an invalid state or waiting for other dependencies in order to perform their duties. This is tricky because if a service or pod is not ready yet but is already receiving requests, then you need to manage it somehow: fail (puts responsibility on the caller), retry (how many times? For how long? How often?), and queue for later (who will manage this queue?).

It is often better if the system at large can be aware of the readiness state of different components, or if components are visible only when they are truly ready. Kubernetes doesn't know Hue, but it provides several mechanisms such as liveness probes, readiness probes, and init containers to support application-specific management of your cluster.

Using liveness probes to ensure your containers are alive

Kubelets watch over your containers. If a container process crashes, the kubelet will take care of it based on the restart policy. But this is not always enough. Your process may not crash but instead run into an infinite loop or a deadlock. The restart policy might not be nuanced enough. With a liveness probe, you get to decide when a container is considered alive. Here is a pod template for the Hue music service. It has a `livenessProbe` section, which uses the `httpGet` probe. An HTTP probe requires a scheme (`http` or `https`, default to `http`; a host, default to `PodIp`; a path; and a port).

The probe is considered successful if the HTTP status is between 200 and 399. Your container may need some time to initialize, so you can specify an `initialDelayInSeconds`. The kubelet will not hit the liveness check during this period:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    app: music
    service: music
  name: hue-music
spec:
  containers:
    - name: hue-music
      image: busybox
      livenessProbe:
        httpGet:
          path: /pulse
          port: 8888
          httpHeaders:
            - name: X-Custom-Header
              value: ItsAlive
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

If a liveness probe fails for any container, then the pod's restart policy comes into effect. Make sure your restart policy is not `Never`, because that will make the probe useless.

There are two other types of probe:

- `TcpSocket` – Just checks that a port is open
- `Exec` – Runs a command that returns 0 for success

Using readiness probes to manage dependencies

Readiness probes are used for different purposes. Your container may be up and running, but it may depend on other services that are unavailable at the moment. For example, `hue-music` may depend on access to a data service that contains your listening history. Without access, it is unable to perform its duties.

In this case, other services or external clients should not send requests to the Hue music service, but there is no need to restart it. Readiness probes address this use case. When a readiness probe fails for a container, the container's pod will be removed from any service endpoint it is registered with. This ensures that requests don't flood services that can't process them. Note that you can also use readiness probes to temporarily remove pods that are overbooked until they drain some internal queue.

Here is a sample readiness probe. I use the exec probe here to execute a custom command. If the command exits a non-zero exit code, the container will be torn down:

```
readinessProbe:
  exec:
    command:
      - /usr/local/bin/checker
      - --full-check
      - --data-service=hue-multimedia-service
  initialDelaySeconds: 60
  timeoutSeconds: 5
```

It is fine to have both a readiness probe and a liveness probe on the same container as they serve different purposes.

Employing init containers for orderly pod bring-up

Liveness and readiness probes are great. They recognize that, at startup, there may be a period where the container is not ready yet but shouldn't be considered failed. To accommodate that, there is the `initialDelayInSeconds` setting, which describes the time for which containers will not be considered failed. But what if this initial delay is potentially very long? Maybe, in most cases, a container is ready after a couple of seconds and ready to process requests, but because the initial delay is set to five minutes just in case, we waste a lot of time where the container is idle. If the container is part of a high-traffic service, then many instances can all sit idle for five minutes after each upgrade and pretty much make the service unavailable.

Init containers address this problem. A pod may have a set of init containers that run to completion before other containers are started. An init container can take care of all the non-deterministic initialization and let application containers with their readiness probe have minimal delay.

Init containers came out of beta in Kubernetes 1.6. You specify them in the pod spec as the `initContainers` field, which is very similar to the `containers` field. Here is an example:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-fitness
spec:
  containers:
  - name: hue-fitness
    image: busybox
  initContainers:
  - name: install
    image: busybox
```

Pod readiness and readiness gates

Pod readiness was introduced in Kubernetes 1.11 and became stable in Kubernetes 1.14. While readiness probes allow you to determine at the pod level if a pod's ready to serve requests, the overall infrastructure that supports delivering traffic to the pod might not be ready yet. For example, the service, network policy, and load balancer might take some extra time. This can be a problem especially during rolling deployments where Kubernetes might terminate the old pods before the new pods are really ready, which will cause degradation in service capacity and even cause a service outage in extreme cases (if all old pods were terminated and no new pod is fully ready).

This is the problem that the `Pod ready++` proposal addresses. The idea is to extend the concept of pod readiness to check additional conditions in addition to making sure all the containers are ready. This is done by adding a new field to the `PodSpec` called `readinessGates`. You can specify a set of conditions that must be satisfied for the pod to be considered ready. In the following example, the pod is not ready because the `"www.example.com/feature-1"` condition has a status of `False`:

```
Kind: Pod
...
spec:
  readinessGates:
  - conditionType: www.example.com/feature-1
status:
  conditions:
  - type: Ready # this is a builtin PodCondition
```

```

    status: "False"
    lastProbeTime: null
    lastTransitionTime: 2018-01-01T00:00:00Z
  - type: "www.example.com/feature-1"    # an extra PodCondition
    status: "False"
    lastProbeTime: null
    lastTransitionTime: 2018-01-01T00:00:00Z
  containerStatuses:
  - containerID: docker://abcd...
    ready: true
  ...

```

Sharing with DaemonSet pods

DaemonSet pods are pods that are deployed automatically, one per node (or a designated subset of the nodes). They are typically used for keeping an eye on nodes and ensuring they are operational. This is a very important function, which we will cover in *Chapter 13, Monitoring Kubernetes Clusters*. But they can be used for much more. The nature of the default Kubernetes scheduler is that it schedules pods based on resource availability and requests. If you have lots of pods that don't require a lot of resources, similarly many pods will be scheduled on the same node. Let's consider a pod that performs a small task and then, every second, sends a summary of all its activities to a remote service. Now, imagine that, on average, 50 of these pods are scheduled on the same node. This means that, every second, 50 pods make 50 network requests with very little data. How about we cut it down by 50× to just a single network request? With a DaemonSet pod, all the other 50 pods can communicate with it instead of talking directly to the remote service. The DaemonSet pod will collect all the data from the 50 pods and, once a second, will report it in aggregate to the remote service. Of course, that requires the remote service API to support aggregate reporting. The nice thing is that the pods themselves don't have to be modified; they will just be configured to talk to the DaemonSet pod on localhost instead of the remote service. The DaemonSet pod serves as an aggregating proxy.

The interesting part about this configuration file is that the `hostNetwork`, `hostPID`, and `hostIPC` options are set to `true`. This enables the pods to communicate efficiently with the proxy, utilizing the fact they are running on the same physical host:

```

apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: hue-collect-proxy
  labels:
    tier: stats

```

```
  app: hue-collect-proxy
spec:
  selector:
    matchLabels:
      tier: stats
      app: hue-collect-proxy
  template:
    metadata:
      labels:
        tier: stats
        app: hue-collect-proxy
    spec:
      hostPID: true
      hostIPC: true
      hostNetwork: true
      containers:
        - name: hue-collect-proxy
          image: busybox
```

Evolving the Hue platform with Kubernetes

In this section, we'll discuss other ways to extend the Hue platform and service additional markets and communities. The question is always this: what Kubernetes features and capabilities can we use to address new challenges or requirements?

Utilizing Hue in an enterprise

An enterprise often can't run in the cloud, either due to security and compliance reasons or for performance reasons because the system has to work with data and legacy systems that are not cost-effective to move to the cloud. Either way, Hue for enterprise must support on-premises clusters and/or bare-metal clusters.

While Kubernetes is most often deployed on the cloud, and even has a special cloud-provider interface, it doesn't depend on the cloud and can be deployed anywhere. It does require more expertise, but enterprise organizations that already run systems on their own data centers have that expertise.

Advancing science with Hue

Hue is so great at integrating information from multiple sources that it would be a boon for the scientific community. Consider how Hue can help multi-disciplinary collaboration between scientists from different disciplines.

A network of scientific communities might require deployment across multiple geographically distributed clusters. Enter cluster federation. Kubernetes has this use case in mind and has developed support for it. We will discuss it at length in a later chapter.

Educating the kids of the future with Hue

Hue can be utilized for education and provide many services to online education systems. But privacy concerns may prevent deploying Hue for kids as a single, centralized system. One possibility is to have a single cluster, with namespaces for different schools. Another deployment option is that each school or county has its own Hue Kubernetes cluster. In the second case, Hue for education must be extremely easy to operate to cater for schools without a lot of technical expertise. Kubernetes can help a lot by providing self-healing and auto-scaling features and capabilities for Hue, to be as close to zero-administration as possible.

Summary

In this chapter, we designed and planned the development, deployment, and management of the Hue platform – an imaginary omniscient and omnipotent service – built on microservice architecture. We used Kubernetes as the underlying orchestration platform, of course, and delved into many of its concepts and resources. In particular, we focused on deploying pods for long-running services as opposed to jobs for launching short-term or cron jobs, explored internal services versus external services, and also used namespaces to segment a Kubernetes cluster. Then we looked at the management of a large system such as Hue with liveness and readiness probes, init containers, and DaemonSets.

You should now feel comfortable architecting web-scale systems composed of microservices and understand how to deploy and manage them in a Kubernetes cluster.

In the next chapter, we will look into the super-important area of storage. Data is king but often the least flexible element of the system. Kubernetes provides a storage model and many options for integrating with various storage solutions.

References

- <https://blog.jetstack.io/blog/kustomize-cert-manager/>
- <https://skryvets.com/blog/2019/05/15/kubernetes-kustomize-json-patches-6902>

6

Managing Storage

In this chapter, we'll look at how Kubernetes manages storage. Storage is very different from compute, but at a high level they are both resources. Kubernetes as a generic platform takes the approach of abstracting storage behind a programming model and a set of plugins for storage providers. First, we'll go into detail about the conceptual storage model and how storage is made available to containers in the cluster. Then, we'll cover the common cloud platform storage providers, such as **Amazon Web Services (AWS)**, **Google Compute Engine (GCE)**, and Azure. Then we'll look at a prominent open source storage provider, GlusterFS from Red Hat, which provides a distributed filesystem. We'll also look into an alternative solution – Flocker – that manages your data in containers as part of the Kubernetes cluster. Finally, we'll see how Kubernetes supports the integration of existing enterprise storage solutions.

At the end of this chapter, you'll have a solid understanding of how storage is represented in Kubernetes, the various storage options in each deployment environment (local testing, public cloud, and enterprise), and how to choose the best option for your use case.

You should try the code samples in this chapter on minikube or another cluster that supports storage adequately. The KinD cluster has some problems related to labeling nodes, which is necessary for some storage solutions.

Persistent volumes walkthrough

In this section, we will understand the Kubernetes storage conceptual model and see how to map persistent storage into containers so they can read and write. Let's start by understanding the problem of storage. Containers and pods are ephemeral.

Anything a container writes to its own filesystem gets wiped out when the container dies. Containers can also mount directories from their host node and read or write to them. These will survive container restarts, but the nodes themselves are not immortal. Also, if the pod itself is rescheduled to a different node, the container will not have access to the old node host's filesystem.

There are other problems, such as ownership for mounted hosted directories when the container dies. Just imagine a bunch of containers writing important data to various data directories on their host and then going away, leaving all that data all over the nodes with no direct way to tell what container wrote what data. You can try to record this information, but where would you record it? It's pretty clear that for a large-scale system, you need persistent storage accessible from any node to reliably manage the data.

Volumes

The basic Kubernetes storage abstraction is the volume. Containers mount volumes that bind to their pod and they access the storage wherever it may be as if it's in their local filesystem. This is nothing new, and it is great, because as a developer who writes applications that need access to data, you don't have to worry about where and how the data is stored.

Using emptyDir for intra-pod communication

It is very simple to share data between containers in the same pod using a shared volume. Container 1 and container 2 simply mount the same volume and can communicate by reading and writing to this shared space. The most basic volume is the `emptyDir`. An `emptyDir` volume is an empty directory on the host. Note that it is not persistent because when the pod is removed from the node, the contents are erased. If a container just crashes, the pod will stick around and you can access it later. Another very interesting option is to use a RAM disk, by specifying the medium as `Memory`. Now, your containers communicate through shared memory, which is much faster, but more volatile of course. If the node is restarted, the `emptyDir`'s volume contents are lost.

Here is a pod configuration file that has two containers that mount the same volume, called `shared-volume`. The containers mount it in different paths, but when the `hue-global-listener` container is writing a file to `/notifications`, the `hue-job-scheduler` will see that file under `/incoming`:

```
apiVersion: v1
kind: Pod
metadata:
```

```

  name: hue-scheduler
spec:
  containers:
  - image: g1g1/hue-global-listener:1.0
    name: hue-global-listener
    volumeMounts:
    - mountPath: /notifications
      name: shared-volume
  - image: g1g1/hue-job-scheduler:1,0
    name: hue-job-scheduler
    volumeMounts:
    - mountPath: /incoming
      name: shared-volume
  volumes:
  - name: shared-volume
    emptyDir: {}

```

To use the shared memory option, we just need to add `medium: Memory` to the `emptyDir` section:

```

volumes:
- name: shared-volume
  emptyDir:
    medium: Memory

```

To verify it worked, let's create the pod and then write a file using one container and read it using the other container:

```

$ kubectl create -f hue-scheduler.yaml
pod/hue-scheduler created

```

Note that the pod has two containers:

```

$ kubectl get pod hue-scheduler -o json | jq .spec.containers
[
  {
    "image": "g1g1/hue-global-listener:1.0",
    "name": "hue-global-listener",
    "volumeMounts": [
      {
        "mountPath": "/notifications",
        "name": "shared-volume"
      },
      ...
    ]
  },
  ...
]

```

```
    ]
    ...
  },
  {
    "image": "g1g1/hue-job-scheduler:1.0",
    "name": "hue-job-scheduler",
    "volumeMounts": [
      {
        "mountPath": "/incoming",
        "name": "shared-volume"
      },
      ...
    ]
  }
  ...
}
```

Now, we can create a file in the `/notifications` directory of the `hue-global-listener` container and list it in the `/incoming` directory of the `hue-job-scheduler` container:

```
$ kubectl exec -it hue-scheduler -c hue-global-listener -- touch /
notifications/1.txt
$ kubectl exec -it hue-scheduler -c hue-job-scheduler -- ls /incoming
1.txt
```

Using HostPath for intra-node communication

Sometimes, you want your pods to get access to some host information (for example, the Docker daemon) or you want pods on the same node to communicate with each other. This is useful if the pods know they are on the same host. Since Kubernetes schedules pods based on available resources, pods usually don't know what other pods they share the node with. There are several cases where a pod can rely on other pods being scheduled with it on the same node:

- In a single-node cluster, all pods obviously share the same node
- DaemonSet pods always share a node with any other pod that matches their selector
- Pods are always scheduled together due to node or pod affinity

For example, in *Chapter 5, Using Kubernetes Resources in Practice*, we discussed a DaemonSet pod that serves as an aggregating proxy to other pods. Another way to implement this behavior is for the pods to simply write their data to a mounted volume that is bound to a host directory and the DaemonSet pod can directly read it and act on it.

Before you decide to use the HostPath volume, make sure you understand the limitations:

- The behavior of pods with the same configuration might be different if they are data-driven and the files on their host are different
- It can violate resource-based scheduling (coming soon to Kubernetes) because Kubernetes can't monitor HostPath resources
- The containers that access host directories must have a security context with privileged set to true or, on the host side, you need to change the permissions to allow writing

Here is a configuration file that mounts the /coupons directory into the hue-coupon-hunter container, which is mapped to the host's /etc/hue/data/coupons directory:

```
apiVersion: v1
kind: Pod
metadata:
  name: hue-coupon-hunter
spec:
  containers:
  - image: busybox    name: hue-coupon-hunter
    volumeMounts:
    - mountPath: /coupons
      name: coupons-volume
  volumes:
  - name: coupons-volume
    host-path:
      path: /etc/hue/data/coupons
```

Since the pod doesn't have a privileged security context, it will not be able to write to the host directory. Let's change the container spec to enable it by adding a security context:

```
- image: the_g1g1/hue-coupon-hunter
  name: hue-coupon-hunter
  volumeMounts:
  - mountPath: /coupons
```

```

name: coupons-volume
securityContext:
  privileged: true

```

In the following diagram, you can see that each container has its own local storage area inaccessible to other containers or pods, and the host's /data directory is mounted as a volume into both container 1 and container 2:

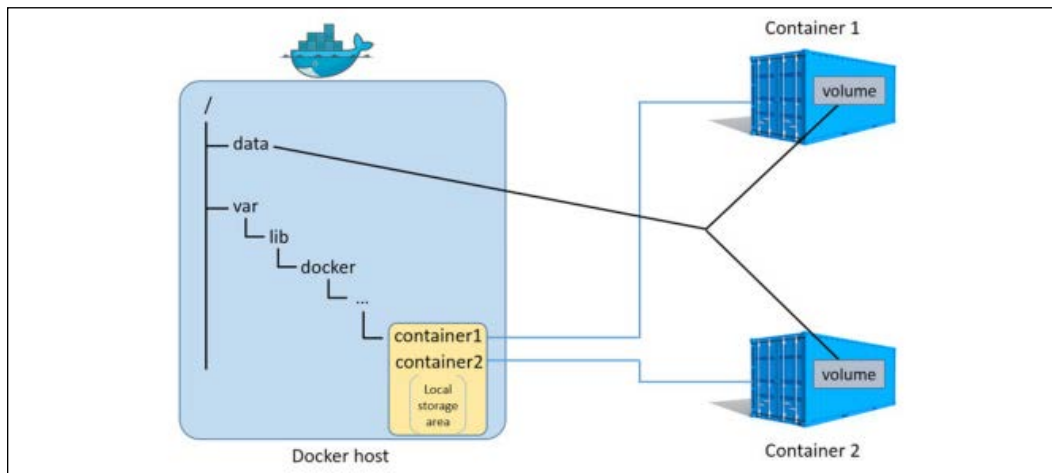


Figure 6.1: Container directories

Using local volumes for durable node storage

Local volumes are similar to HostPath, but they persist across pod restarts and node restarts. In that sense, they are considered persistent volumes. They were added in Kubernetes 1.7. As of Kubernetes 1.14, they are considered stable. The purpose of local volumes is to support StatefulSets where specific pods need to be scheduled on nodes that contain specific storage volumes. Local volumes have node affinity annotations that simplify the binding of pods to the storage they need to access.

We need to define a storage class for using local volumes. We will cover storage classes in depth later in this chapter. In one sentence, storage classes use a provisioner to allocate storage to pods. Let's define the storage class in a file called `local-storage-class.yaml` and create it:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: local-storage
provisioner: kubernetes.io/no-provisioner
volumeBindingMode: WaitForFirstConsumer

```

```
$ kubectl create -f local-storage-class.yaml
storageclass.storage.k8s.io/local-storage created
```

Now, we can create a persistent volume using the storage class that will persist even after the pod that's using it is terminated:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
  labels:
    release: stable
    capacity: 100Gi
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/disk-1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - minikube
```

Provisioning persistent volumes

While emptyDir volumes can be mounted and used by containers, they are not persistent and don't require any special provisioning because they use existing storage on the node. HostPath volumes persist on the original node, but if a pod is restarted on a different node, it can't access the HostPath volume from its previous node. Local volumes are real persistent volumes that use storage provisioned ahead of time by administrators or dynamic provisioning via storage classes. They persist on the node and can survive pod restarts and rescheduling and even node restarts. Some persistent volumes use external storage (not a disk physically attached to the node) provisioned ahead of time by administrators.

In cloud environments, the provisioning may be very streamlined but it is still required, and as a Kubernetes cluster administrator you have to at least make sure your storage quota is adequate and monitor usage versus your quota diligently.

Remember that persistent volumes are resources that the Kubernetes cluster is using, similar to nodes. As such, they are not managed by the Kubernetes API server.

You can provision resources statically or dynamically.

Provisioning persistent volumes statically

Static provisioning is straightforward. The cluster administrator creates persistent volumes backed up by some storage media ahead of time, and these persistent volumes can be claimed by containers.

Provisioning persistent volumes dynamically

Dynamic provisioning may happen when a persistent volume claim doesn't match any of the statically provisioned persistent volumes. If the claim specified a storage class and the administrator configured that class for dynamic provisioning, then a persistent volume may be provisioned on the fly. We will see examples later when we discuss persistent volume claims and storage classes.

Provisioning persistent volumes externally

One of the recent trends is to move storage provisioners out of Kubernetes core into volume plugins (also known as *out-of-tree*). External provisioners work just like in-tree dynamic provisioners but can be deployed and updated independently. More and more in-tree storage provisioners migrate out-of-tree. Check out this Kubernetes incubator project:

<https://github.com/kubernetes-incubator/external-storage>

Creating persistent volumes

Here is the configuration file for an NFS persistent volume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-777
spec:
  capacity:
    storage: 100Gi
```

```

volumeMode: Filesystem
accessModes:
  - ReadWriteMany
  - ReadOnlyMany
persistentVolumeReclaimPolicy: Recycle
storageClassName: slow
mountOptions:
  - hard
  - nfsvers=4.1
nfs:
  path: /tmp
  server: 172.17.0.2

```

A persistent volume has a spec and metadata that possibly includes a storage class name. Let's focus on the spec here. There are six sections: capacity, volume mode, access modes, reclaim policy, storage class, and the volume type (nfs in the example).

Capacity

Each volume has a designated amount of storage. Storage claims may be satisfied by persistent volumes that have at least that amount of storage. In the example, the persistent volume has a capacity of 100 gibibytes (a single **gibibyte (GiB)** is 2 to the power of 30 bytes). It is important when allocating static persistent volumes to understand the storage request patterns. For example, if you provision 20 persistent volumes with 100 GiB capacity and a container claims a persistent volume with 150 GiB, then this claim will not be satisfied even though there is enough capacity overall in the cluster:

```

capacity:
  storage: 100Gi

```

Volume mode

The optional volume mode was added in Kubernetes 1.9 as an alpha feature (moved to beta in Kubernetes 1.13) for static provisioning. It lets you specify whether you want a filesystem ("Filesystem") or raw storage ("Block"). If you don't specify volume mode, then the default is "Filesystem", just like it was pre-1.9.

Access modes

There are three access modes:

- **ReadOnlyMany**: Can be mounted read-only by many nodes

- `ReadWriteOnce`: Can be mounted as read-write by a single node
- `ReadWriteMany`: Can be mounted as read-write by many nodes

The storage is mounted to nodes, so even with `ReadWriteOnce`, multiple containers on the same node can mount the volume and write to it. If that causes a problem, you need to handle it through some other mechanism (for example, claim the volume only in `DaemonSet` pods that you know will have just one per node).

Different storage providers support some subset of these modes. When you provision a persistent volume, you can specify which modes it will support. For example, NFS supports all modes, but in the example, only these modes were enabled:

```
accessModes:  
- ReadWriteMany  
- ReadOnlyMany
```

Reclaim policy

The reclaim policy determines what happens when a persistent volume claim is deleted. There are three different policies:

- **Retain**: The volume will need to be reclaimed manually
- **Delete**: The associated storage asset, such as AWS EBS, GCE PD, Azure disk, or OpenStack Cinder volume, is deleted
- **Recycle**: Delete content only (`rm -rf /volume/*`)

The Retain and Delete policies mean the persistent volume is not available anymore for future claims. The Recycle policy allows the volume to be claimed again.

Currently, only NFS and `HostPath` support recycling. AWS EBS, GCE PD, Azure disks, and Cinder volumes support deletion. Dynamically provisioned volumes are always deleted.

Storage class

You can specify a storage class using the optional `storageClassName` field of the spec. If you do, then only persistent volume claims that specify the same storage class can be bound to the persistent volume. If you don't specify a storage class, then only persistent volume claims that don't specify a storage class can be bound to it:

```
storageClassName: slow
```

Volume type

The volume type is specified by name in the spec. There is no `volumeType` section. In the preceding example, NFS is the volume type:

```
nfs:
  path: /tmp
  server: 172.17.0.8
```

Each volume type may have its own set of parameters. In this case, it's a path and server.

We will go over various volume types later.

Mount options

Some persistent volume types have additional mount options you can specify. The mount options are not validated. If you provide an invalid mount option, the volume provisioning will fail. For example, NFS supports additional mount options:

```
mountOptions:
- hard
- nfsvers=4.1
```

Making persistent volume claims

When containers want access to some persistent storage, they make a claim (or rather, the developer and cluster administrator coordinate on necessary storage resources to claim). The claim will match some storage (such as a volume).

Let's create a local volume. First, we need to create a backing directory:

```
$ mk ssh "sudo mkdir -p /mnt/disks/disk-1"
```

Now, we can create a local volume backed by the `/mnt/disks/disk1` directory:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: local-pv
  labels:
    release: stable
    capacity: 100Gi
```

```
spec:
  capacity:
    storage: 100Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Delete
  storageClassName: local-storage
  local:
    path: /mnt/disks/disk-1
  nodeAffinity:
    required:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - minikube
```

```
$ kubectl create -f local-volume.yaml
persistentvolume/local-pv created
```

Here is a sample claim that matches the persistent volume we just created:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: local-storage-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 80Gi
  storageClassName: local-storage
  selector:
    matchLabels:
      release: "stable"
    matchExpressions:
      - {key: capacity, operator: In, values: [80Gi, 100Gi]}
```

Let's create the claim and then explain what the different pieces do:

```
$ kubectl create -f local-persistent-volume-claim.yaml  
persistentvolumeclaim/local-storage-claim created
```

The name `storage-claim` will be important later when mounting the claim into a container.

The access mode in the spec is `ReadWriteOnce`, which means if the claim is satisfied, no other claim with the `ReadWriteOnce` access mode can be satisfied, but claims for `ReadOnlyMany` can still be satisfied.

The resources section requests 80 GiB. This can be satisfied by our persistent volume, which has a capacity of 100 Gi. But, this is a little bit of a waste because 20 Gi will not be used by definition.

The storage class name is `normal`. As mentioned earlier it must match the class name of the persistent volume. However, with PVC there is a difference between an empty class name (`""`) and no class name at all. The former (empty class name) matches persistent volumes with no storage class name. The latter (no class name) will be able to bind to persistent volumes only if the `DefaultStorageClass` admission plugin is turned off or if it's on and the default storage class is used.

The selector section allows you to filter available volumes further. For example, here the volume must match the label `release:stable` and also have a label with either `capacity:80Gi` or `capacity:100Gi`. Imagine that we have several other volumes provisioned with capacities of 200 Gi and 500 Gi. We don't want to claim a 500 Gi volume when we only need 80 Gi.

Kubernetes always tries to match the smallest volume that can satisfy a claim, but if there are no 80 Gi or 100 Gi volumes then the labels will prevent assigning a 200 Gi or 500 Gi volume and use dynamic provisioning instead.

It's important to realize that claims don't mention volumes by name. You can't claim a specific volume. The matching is done by Kubernetes based on storage class, capacity, and labels.

Finally, persistent volume claims belong to a namespace. Binding a persistent volume to a claim is exclusive. That means that a persistent volume will be bound to a namespace. Even if the access mode is `ReadOnlyMany` or `ReadWriteMany`, all the pods that mount the persistent volume claim must be from that claim's namespace.

Mounting claims as volumes

OK. We have provisioned a volume and claimed it. It's time to use the claimed storage in a container. This turns out to be pretty simple. First, the persistent volume claim must be used as a volume in the pod and then the containers in the pod can mount it, just like any other volume. Here is a pod configuration file that specifies the persistent volume claim we created earlier (bound to the local persistent volume we provisioned):

```
kind: Pod
apiVersion: v1
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: g1g1/py-kube:0.2
      volumeMounts:
        - mountPath: "/mnt/data"
          name: persistent-volume
  volumes:
    - name: persistent-volume
      persistentVolumeClaim:
        claimName: local-storage-claim
```

The key is in the `persistentVolumeClaim` section under `volumes`. The claim name (storage-claim here) uniquely identifies the specific claim within the current namespace and makes it available as a volume named `persistent-volume` here. Then, the container can refer to it by its name and mount it to `/mnt/data`.

Before we create the pod it's important to note that the persistent volume claim didn't actually claim any storage yet and wasn't bound to our local volume. The claim is pending until some container actually attempts to mount a volume using the claim:

```
$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES
STORAGECLASS	AGE			
local-storage-claim	Pending			local-
storage	12s			

Now, the claim will be bound when creating the pod:

```
$ kubectl create -f pod-with-local-claim.yaml
```

pod/the-pod created

```
$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES
STORAGECLASS	AGE			
local-storage-claim	Bound	local-pv	100Gi	RWO
storage 20m				local-

Raw block volumes

Kubernetes 1.9 added this capability as an alpha feature. As of Kubernetes 1.16, it is in beta.

Raw block volumes provide direct access to the underlying storage, which is not mediated via a filesystem abstraction. This is very useful for applications that require high performance from storage, such as databases, or when consistent I/O performance and low latency are needed. **Fiber Channel (FC)**, iSCSI, and a local SSD are all suitable for use as raw block storage. As of Kubernetes 1.16, the following storage providers support raw block volumes:

- AWS Elastic Block Store
- AzureDisk
- FC
- GCEPersistentDisk
- iSCSI
- Local volume
- RBD (Ceph Block Device)
- Vsphere volume (alpha)

Here is how to define a raw block volume using an FC provider:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: block-pv
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
```

```
persistentVolumeReclaimPolicy: Retain
fc:
  targetWWNs: ["50060e801049cfd1"]
  lun: 0
  readOnly: false
```

A matching **Persistent Volume Claim (PVC)** *must* specify `volumeMode: Block` as well. Here is what it looks like:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: block-pvc
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Block
  resources:
    requests:
      storage: 10Gi
```

Pods consume raw block volumes as devices under `/dev` and *not* as mounted filesystems. Containers can then access this device and read/write to it. In practice, this means that I/O requests to block storage go directly to the underlying block storage and don't pass through the filesystem drivers. This is faster in theory, but in practice, it can actually decrease performance if your applications benefit from filesystem buffering.

Here is a pod with a container that binds the `block-pvc` with the raw block storage as a device named `/dev/xvda`:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-block-volume
spec:
  containers:
    - name: fc-container
      image: fedora:26
      command: ["/bin/sh", "-c"]
      args: ["tail -f /dev/null"]
      volumeDevices:
        - name: data
```

```
        devicePath: /dev/xvda
volumes:
  - name: data
    persistentVolumeClaim:
      claimName: block-pvc
```

Storage classes

Storage classes let an administrator configure your cluster with custom persistent storage (as long as there is a proper plugin to support it). A storage class has a name in the metadata (it must be specified in the annotation to claim), a provisioner, and parameters.

We declared a storage class for local storage earlier. Here is a sample storage class that uses AWS EBS as a provisioner (so, it works only on AWS):

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

You may create multiple storage classes for the same provisioner with different parameters. Each provisioner has its own parameters.

The currently supported provisioners are as follows:

- AWS Elastic Block Store
- AzureFile
- AzureDisk
- CephFS
- Cinder
- FC
- FlexVolume
- Flocker
- GcePersistentDisk

- GlusterFS
- iSCSI
- Quobyte
- NFS
- RBD
- Vsphere volume
- Portworx volume
- ScaleIO
- StorageOS
- Local

This list doesn't contain provisioners for other volume types, such as gitRepo or secret, that are not backed by your typical network storage. Those volume types don't require a storage class. Utilizing volume types intelligently is a major part of architecting and managing your cluster.

Default storage class

The cluster administrator can also assign a default storage class. When a default storage class is assigned and the DefaultStorageClass admission plugin is turned on, then claims with no storage class will be dynamically provisioned using the default storage class. If the default storage class is not defined or the admission plugin is not turned on, then claims with no storage class can only match volumes with no storage class.

Demonstrating persistent volume storage end to end

To illustrate all the concepts, let's do a mini demonstration where we create a HostPath volume, claim it, mount it, and have containers write to it.

Let's start by creating a hostPath volume using the dir storage class. Save the following in `dir-persistent-volume.yaml`:

```
kind: PersistentVolume
apiVersion: v1
metadata:
  name: dir-pv
spec:
```

```

storageClassName: dir
capacity:
  storage: 1Gi
accessModes:
  - ReadWriteMany
hostPath:
  path: "/tmp/data"

```

Then, let's create it:

```

$ kubectl create -f dir-persistent-volume.yaml
persistentvolume/dir-pv created

```

To check out the available volumes, you can use the resource type `persistentvolumes` or `pvc` for short:

```

$ kubectl get pv

```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
dir-pv	1Gi	RWX	Retain	Available	dir

The capacity is 1 GiB as requested. The reclaim policy is `Retain` because host path volumes are retained (not destroyed). The status is `Available` because the volume has not been claimed yet. The access mode is specified as `RWX`, which means `ReadWriteMany`. All access modes have a shorthand version:

- **RWO**: `ReadWriteOnce`
- **ROX**: `ReadOnlyMany`
- **RWX**: `ReadWriteMany`

We have a persistent volume. Let's create a claim. Save the following to `dir-persistent-volume-claim.yaml`:

```

kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: dir-pvc
spec:
  storageClassName: dir  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi

```

Then, run the following command:

```
$ kubectl create -f dir-persistent-volume-claim.yaml
persistentvolumeclaim/dir-pvc created
```

Let's check the claim and the volume:

```
$ kubectl get pvc
NAME      STATUS   VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS
dir-pvc   Bound    dir-pv   1Gi        RWX             dir

$ kubectl get pv
NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
STORAGECLASS
dir-pv    1Gi        RWX             Retain            Bound    default/dir-
pvc      dir
```

As you can see, the claim and the volume are bound to each other and reference each other. The reason the binding works is that the same storage class is used by the volume and the claim. But, what happens if they don't match? Let's remove the storage class from the persistent volume claim and see what happens. Save the following persistent volume claim to `some-persistent-volume-claim.yaml`:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: some-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 1Gi
```

Then, create it:

```
$ kubectl create -f some-persistent-volume-claim.yaml
persistentvolumeclaim/some-pvc created
```

OK. It was created. Let's check it out:

```
$ kubectl get pvc
NAME      STATUS   VOLUME   CAPACITY
ACCESS MODES   STORAGECLASS
dir-pvc   Bound    dir-pv   1Gi
```

```

RWX          dir
some-pvc     Bound      pvc-276fdd9d-b787-4c3e-a94b-e886edaa1039  1Gi
RWX          standard

```

Very interesting. The some-pvc claim was bound to a new volume using the standard storage class. This is an example of dynamic provisioning, where a new persistent volume was created on the fly to satisfy the some-pvc claim that didn't match any existing volume.

Here is the dynamic volume. It is also a HostPath volume created under /tmp/hostpath-provisioner:

```

$ kubectl get pv pvc-276fdd9d-b787-4c3e-a94b-e886edaa1039 -o yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  annotations:
    hostPathProvisionerIdentity: 7f22c7da-dc16-11e9-a3e9-080027a42754
    pv.kubernetes.io/provisioned-by: k8s.io/minikube-hostpath
  creationTimestamp: "2020-06-08T23:11:36Z"
  finalizers:
    - kubernetes.io/pv-protection
  name: pvc-276fdd9d-b787-4c3e-a94b-e886edaa1039
  resourceVersion: "193570"
  selfLink: /api/v1/persistentvolumes/pvc-276fdd9d-b787-4c3e-a94b-e886edaa1039
  uid: e1f6579f-8ddb-401f-be44-f52742c91cfa
spec:
  accessModes:
    - ReadWriteMany
  capacity:
    storage: 1Gi
  claimRef:
    apiVersion: v1
    kind: PersistentVolumeClaim
    name: some-pvc
    namespace: default
    resourceVersion: "193563"
    uid: 276fdd9d-b787-4c3e-a94b-e886edaa1039
  hostPath:
    path: /tmp/hostpath-provisioner/pvc-276fdd9d-b787-4c3e-a94b-e886edaa1039
    type: ""

```

```
persistentVolumeReclaimPolicy: Delete
storageClassName: standard
volumeMode: Filesystem
status:
  phase: Bound
```

The final step is to create a pod and assign the claim as a volume. Save the following to `shell-pod.yaml`:

```
kind: Pod
apiVersion: v1
metadata:
  name: just-a-shell
  labels:
    name: just-a-shell
spec:
  containers:
    - name: a-shell
      image: g1g1/py-kube:0.2
      command: ["/bin/bash", "-c", "while true ; do sleep 1 ; done"]
      volumeMounts:
        - mountPath: "/data"
          name: pv
    - name: another-shell
      image: g1g1/py-kube:0.2
      command: ["/bin/bash", "-c", "while true ; do sleep 1 ; done"]
      volumeMounts:
        - mountPath: "/another-data"
          name: pv
  volumes:
    - name: pv
      persistentVolumeClaim:
        claimName: dir-pvc
```

This pod has two containers that use the `g1g1/py-kube:0.2` image and both run a shell command that just sleeps in an infinite loop. The idea is that the containers will keep running, so we can connect to them later and check their filesystem. The pod mounts our persistent volume claim with a volume name of `pv`. Note that the volume specification is done at the pod level just once and multiple containers can mount it into different directories.

We can even create a new file, `yo.txt`, in one of the containers and see that it's available to the other container or to the node itself:

```
$ docker exec -it 341f7ab2b4cc bash
root@just-a-shell:/# echo yo > /another-data/yo.txt
root@just-a-shell:/#
```

Let's verify directly from `kubectl` that `yo.txt` is available to both containers:

```
$ kubectl exec -it just-a-shell -c a-shell -- cat /data/yo.txt
yo

$ kubectl exec -it just-a-shell -c another-shell -- cat /another-data/
yo.txt
yo
```

Yes. Everything works as expected and both containers share the same storage.

Public cloud storage volume types – GCE, AWS, and Azure

In this section, we'll look at some of the common volume types available in the leading public cloud platforms. Managing storage at scale is a difficult task that eventually involves physical resources, similar to nodes. If you choose to run your Kubernetes cluster on a public cloud platform, you can let your cloud provider deal with all these challenges and focus on your system. But it's important to understand the various options, constraints, and limitations of each volume type.

Many of the volume types we will go over are handled by in-tree plugins (part of core Kubernetes), but are in the process of migrating to out-of-tree CSI plugins. We will cover CSI later.

Amazon EBS

AWS provides **Elastic Block Store (EBS)** as persistent storage for EC2 instances. An AWS Kubernetes cluster can use AWS EBS as persistent storage with the following limitations:

- The pods must run on AWS EC2 instances as nodes
- Pods can only access EBS volumes provisioned in their availability zone
- An EBS volume can be mounted on a single EC2 instance.

Those are severe limitations. The restriction for a single availability zone, while great for performance, eliminates the ability to share storage at scale or across a geographically distributed system without custom replication and synchronization. The limit of a single EBS volume to a single EC2 instance means even within the same availability zone, pods can't share storage (even for reading) unless you make sure they run on the same node.

With all the disclaimers out of the way, let's see how to mount an EBS volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - image: some-container
    name: some-container
    volumeMounts:
    - mountPath: /ebs
      name: some-volume
  volumes:
  - name: some-volume
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

You must create the EBS volume in AWS and then you just mount it into the pod. There is no need for a claim or storage class because you mount the volume directly by ID. The `awsElasticBlockStore` volume type is known to Kubernetes.

Amazon EFS

AWS recently released a new service called **Elastic File System (EFS)**. This is really a managed NFS service. It's using NFS 4.1 protocol and it has many benefits over EBS:

- Multiple EC2 instances can access the same files across multiple availability zones (but within the same region)
- Capacity is automatically scaled up and down based on actual usage
- You pay only for what you use

- You can connect on-premises servers to EFS over VPN
- EFS runs off SSD drives that are automatically replicated across availability zones

That said, EFS is more expensive than EBS even when you consider the automatic replication to multiple availability zones (assuming you fully utilize your EBS volumes). It uses an external provisioner and it is not trivial to deploy. Follow the instructions here:

<https://github.com/kubernetes-incubator/external-storage/tree/master/aws/efs>

From Kubernetes' point of view, AWS EFS is just an NFS volume. You provision it as such:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: efs-share
spec:
  capacity:
    storage: 200Gi
  accessModes:
    - ReadWriteMany
  nfs:
    server: eu-west-1b.fs-64HJku4i.efs.eu-west-1.amazonaws.com
    path: /
```

Once everything is set up, you've defined your storage class, and the persistent volume exists, you can create a claim and mount it into as many pods as you like in ReadWriteMany mode. Here is the persistent claim:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: efs
  annotations:
    volume.beta.kubernetes.io/storage-class: "aws-efs"
spec:
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 1Mi
```

Here is a pod that consumes it:

```
kind: Pod
apiVersion: v1
metadata:
  name: test-pod
spec:
  containers:
  - name: test-pod
    image: gcr.io/google/_containers/busybox:1.24
    command:
      - "/bin/sh"
    args:
      - "-c"
      - "touch /mnt/SUCCESS && exit 0 || exit 1"
    volumeMounts:
      - name: efs-pvc
        mountPath: "/mnt"
  restartPolicy: "Never"
  volumes:
  - name: efs-pvc
    persistentVolumeClaim:
      claimName: efs
```

GCE persistent disk

The `gcePersistentDisk` volume type is very similar to `awsElasticBlockStore`. You must provision the disk ahead of time. It can only be used by GCE instances in the same project and zone. But the same volume can be used as read-only on multiple instances. This means it supports `ReadWriteOnce` and `ReadOnlyMany`. You can use a GCE persistent disk to share data as read-only between multiple pods in the same zone.

The pod that's using a persistent disk in `ReadWriteOnce` mode must be controlled by a replication controller, a replica set, or a deployment with a replica count of 0 or 1. Trying to scale beyond 1 will fail for obvious reasons:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
```

```
- image: some-image
  name: some-container
  volumeMounts:
    - mountPath: /pd
      name: some-volume
  volumes:
    - name: some-volume
      gcePersistentDisk:
        pdName: <persistent disk name>
        fsType: ext4
```

The GCE persistent disk supports a regional disk option since Kubernetes 1.10 (in beta). Regional persistent disks automatically sync between two zones. The key to using them in Kubernetes is to specify a special label for failure domain that specifies the two zones:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: test-volume
  labels:
    failure-domain.beta.kubernetes.io/zone: us-central1-a__us-
    central1-b
spec:
  capacity:
    storage: 400Gi
  accessModes:
    - ReadWriteOnce
  gcePersistentDisk:
    pdName: data-disk
    fsType: ext4
```

Azure data disk

The Azure data disk is a virtual hard disk stored in Azure Storage. It's similar in capabilities to AWS EBS. Here is a sample pod configuration file:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
```

```

- image: some-container
  name: some-container
  volumeMounts:
    - name: some-volume
      mountPath: /azure
volumes:
  - name: some-volume
    azureDisk:
      diskName: test.vhd
      diskURI: https://someaccount.blob.microsoft.net/vhds/test.vhd

```

In addition to the mandatory `diskName` and `diskURI` parameters, it also has a few optional parameters:

- `kind`: Either Shared (multiple disks per storage account), Dedicated (a single blob disk per storage account), or Managed (an Azure-managed data disk). The default is Shared.
- `cachingMode`: The disk caching mode. This must be either `None`, `ReadOnly`, or `ReadWrite`. The default is `None`.
- `fsType`: The filesystem type set to mount. The default is `ext4`.
- `readOnly`: Sets whether the filesystem is to be used as `readOnly`. The default is `false`.

Azure data disks are limited to 1,023 GB. Each Azure VM can have up to 16 data disks. You can attach an Azure data disk to a single Azure VM.

Azure Files

In addition to the data disk, Azure has also a shared filesystem similar to AWS EFS. However, Azure Files uses the SMB/CIFS protocol (it supports SMB 2.1 and SMB 3.0). It is based on the Azure storage platform and has the same availability, durability, scalability, and geo-redundancy capabilities as Azure Blob, Table, or Queue storage.

In order to use Azure Files, you need to install on each client VM the `cifs-utils` package. You also need to create a secret, which is a required parameter:

```

apiVersion: v1
kind: Secret
metadata:
  name: azure-file-secret
type: Opaque

```

```
data:
  azurestorageaccountname: <base64 encoded account name>
  azurestorageaccountkey: <base64 encoded account key>
```

Here is a pod that uses Azure Files:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
    - image: some-container
      name: some-container
      volumeMounts:
        - name: some-volume
          mountPath: /azure
  volumes:
    - name: some-volume
      azureFile:
        secretName: azure-file-secret
        shareName: azure-share
        readOnly: false
```

Azure Files supports sharing within the same region as well as connecting on-premises clients.

GlusterFS and Ceph volumes in Kubernetes

GlusterFS and Ceph are two distributed persistent storage systems. GlusterFS is, at its core, a network filesystem. Ceph is, at its core, an object store. Both expose block, object, and filesystem interfaces. Both use the xfs filesystem under the hood to store the data and metadata as xattr attributes. There are several reasons why you may want to use GlusterFs or Ceph as persistent volumes in your Kubernetes cluster:

- You may have a lot of data and applications that access the data in GlusterFS or Ceph
- You have operational expertise managing GlusterFS or Ceph
- You run in the cloud, but the limitations of the cloud platform's persistent storage are a non-starter

Using GlusterFS

GlusterFS is intentionally simple, exposing the underlying directories as they are and leaving it to clients (or middleware) to handle high availability, replication, and distribution. Gluster organizes the data into logical volumes, which encompass multiple nodes (machines) that contain bricks, which store files. Files are allocated to bricks according to the **Distributed Hash Table (DHT)**. If files are renamed or the GlusterFS cluster is expanded or rebalanced, files may be moved between bricks. The following diagram shows the GlusterFS building blocks:

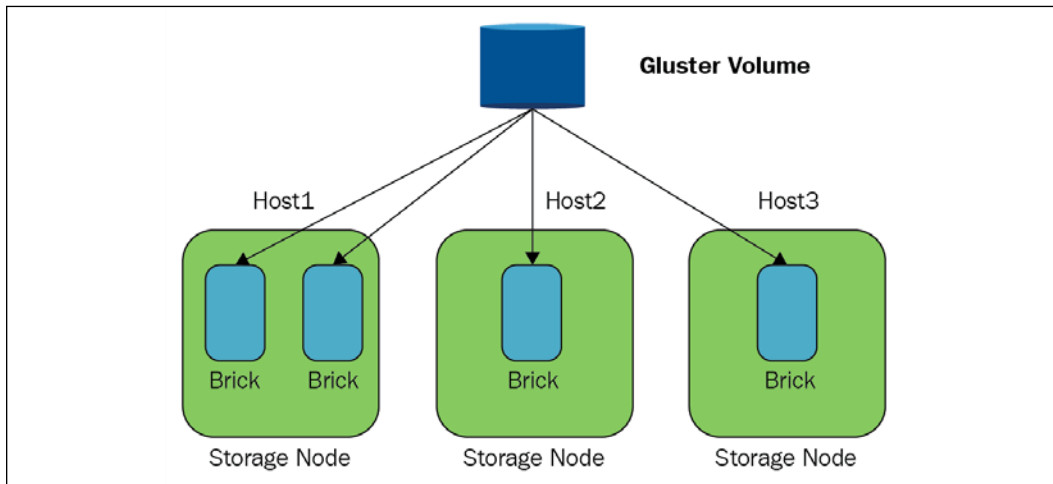


Figure 6.2: Gluster FS building blocks

To use a GlusterFS cluster as persistent storage for Kubernetes (assuming you have an up-and-running GlusterFS cluster), you need to follow several steps. In particular, the GlusterFS nodes are managed by the plugin as a Kubernetes service (although as an application developer it doesn't concern you).

Creating endpoints

Here is an example of an endpoints resource that you can create as a normal Kubernetes resource using `kubectl create`:

```
{
  "kind": "Endpoints",
  "apiVersion": "v1",
  "metadata": {
    "name": "glusterfs-cluster"
  },
  "subsets": [
```

```
{
  "addresses": [
    {
      "ip": "10.240.106.152"
    }
  ],
  "ports": [
    {
      "port": 1
    }
  ]
},
{
  "addresses": [
    {
      "ip": "10.240.79.157"
    }
  ],
  "ports": [
    {
      "port": 1
    }
  ]
}
]
```

Adding a GlusterFS Kubernetes service

To make the endpoints persistent, you use a Kubernetes service with no selector to indicate the endpoints are managed manually:

```
{
  "kind": "Service",
  "apiVersion": "v1",
  "metadata": {
    "name": "glusterfs-cluster"
  },
  "spec": {
    "ports": [
      {"port": 1}
    ]
  }
}
```

Creating pods

Finally, in the pod spec's volumes section, provide the following information:

```
"volumes": [
  {
    "name": "glusterfsvol",
    "glusterfs": {
      "endpoints": "glusterfs-cluster",
      "path": "kube\_vol",
      "readOnly": true
    }
  }
]
```

The containers can then mount `glusterfsvol` by name.

The endpoints tell the GlusterFS volume plugin how to find the storage nodes of the GlusterFS cluster.

Using Ceph

Ceph's object store can be accessed using multiple interfaces. Kubernetes supports the **Rados Block Device (RBD)** (block) and **CEPHFS** (filesystem) interfaces. Unlike GlusterFS, Ceph does a lot of work automatically. It does distribution, replication, and self-healing all on its own. The following diagram shows how RADOS – the underlying object store – can be accessed in multiple ways:

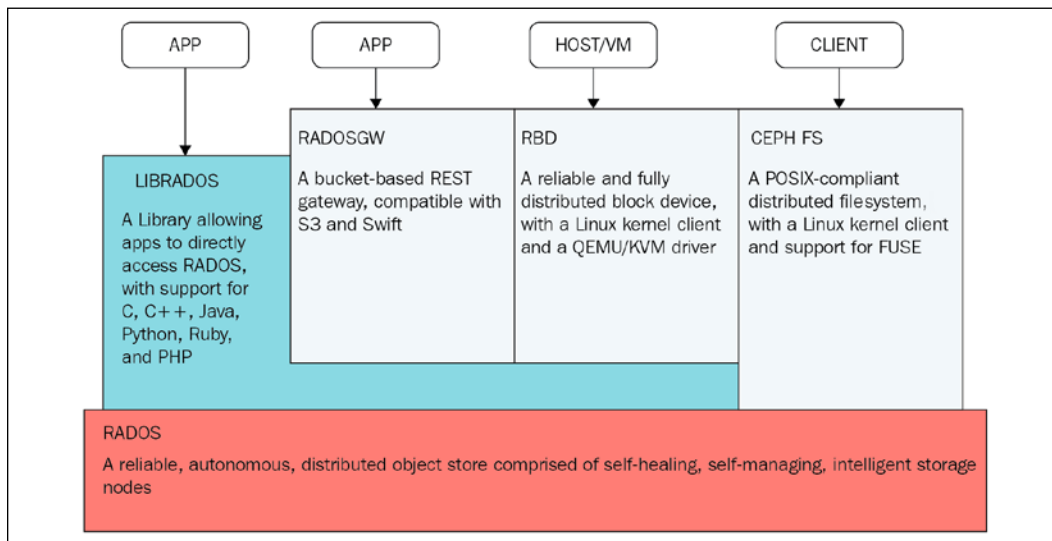


Figure 6.3: Ways to access RADOS

Kubernetes supports Ceph via the Rados Block Device (RBD) interface.

Connecting to Ceph using RBD

You must install `ceph-common` on each node of the Kubernetes cluster. Once you have your Ceph cluster up and running, you need to provide some information required by the Ceph RBD volume plugin in the pod configuration file:

- **monitors:** The Ceph monitors.
- **pool:** The name of the RADOS pool. If not provided, the default RBD pool is used.
- **image:** The image name that RBD has created.
- **user:** The RADOS username. If not provided, the default admin is used.
- **keyring:** The path to the keyring file. If not provided, the default `/etc/ceph/keyring` is used.

- `secretName`: The name of the authentication secrets. If provided, `secretName` overrides `keyring`. See the following paragraph for more information about how to create a secret.
- `fsType`: The filesystem type (`ext4`, `xfs`, and so on) that is formatted on the device.
- `readOnly`: Whether the filesystem is used as `readOnly`.

If the Ceph authentication secret is used, you need to create a secret object:

```
apiVersion: v1
kind: Secret
metadata:
  name: ceph-secret
type: "kubernetes.io/rbd"
data:
  key: QVFCMTZWVZvRjVtRXhBQTVrQ1FzN2JCajhWVUxSdzI2Qzg0SEE9PQ==
```

The secret type is `kubernetes.io/rbd`.

Here is a sample pod that uses Ceph through RBD with a secret:

```
apiVersion: v1
kind: Pod
metadata:
  name: rbd2
spec:
  containers:
    - image: kubernetes/pause
      name: rbd-rw
      volumeMounts:
        - name: rbdpd
          mountPath: /mnt/rbd
  volumes:
    - name: rbdpd
      rbd:
        monitors:
          - '10.16.154.78:6789'
          - '10.16.154.82:6789'
          - '10.16.154.83:6789'
        pool: kube
        image: foo
        fsType: ext4
        readOnly: true
```

```
user: admin
secretRef:
  name: ceph-secret
```

Ceph RBD supports the `ReadWriteOnce` and `ReadOnlyMany` access modes.

Connecting to Ceph using CephFS

If your Ceph cluster is already configured with CephFS, then you can assign it very easily to pods. Also, CephFS supports `ReadWriteMany` access modes.

The configuration is similar to Ceph RBD, except you don't have a pool, image, or filesystem type. The secret can be a reference to a Kubernetes secret object (preferred) or a secret file:

```
apiVersion: v1
kind: Pod
metadata:
  name: cephfs2
spec:
  containers:
    - name: cephfs-rw
      image: kubernetes/pause
      volumeMounts:
        - mountPath: "/mnt/cephfs"
          name: cephfs
  volumes:
    - name: cephfs
      cephfs:
        monitors:
          - 10.16.154.78:6789
          - 10.16.154.82:6789
          - 10.16.154.83:6789
        user: admin
        secretRef:
          name: ceph-secret
        readOnly: true
```

You can also provide a path as a parameter in the CephFS system. The default is `/`.

The in-tree RBD provisioner has an out-of-tree copy now in the external-storage Kubernetes incubator project.

Flocker as a clustered container data volume manager

So far, we have discussed storage solutions that stored the data outside the Kubernetes cluster (except for `emptyDir` and `HostPath`, which are not persistent). Flocker is a little different. It is Docker-aware. It was designed to let Docker data volumes transfer with their container when the container is moved between nodes. You may want to use the Flocker volume plugin if you're migrating a Docker-based system that uses a different orchestration platform, such as Docker Compose or Mesos, to Kubernetes and you use Flocker for orchestrating storage. Personally, I feel that there is a lot of duplication between what Flocker does and what Kubernetes does to abstract storage.

Flocker has a control service and agents on each node. Its architecture is very similar to Kubernetes with its API server and the kubelet running on each node. The Flocker control service exposes a REST API and manages the configuration of the state across the cluster. The agents are responsible for ensuring that the state of their node matches the current configuration. For example, if a dataset needs to be on node X, then the Flocker agent on node X will create it.

The following diagram showcases the Flocker architecture:

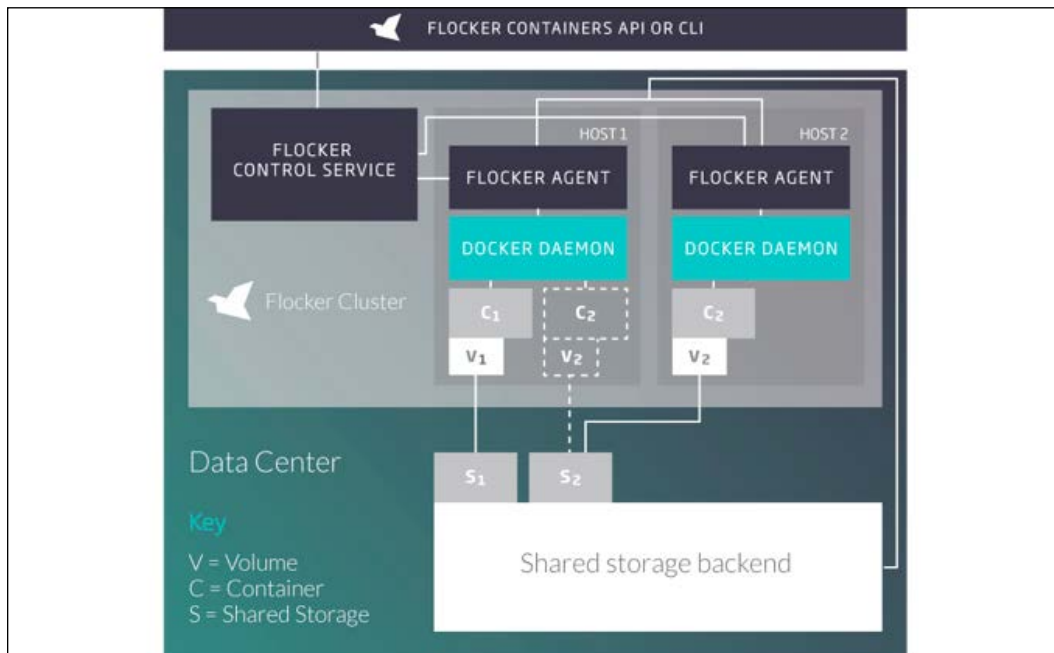


Figure 6.4: The Flocker architecture

In order to use Flocker as persistent volumes in Kubernetes, you first must have a properly configured Flocker cluster. Flocker can work with many backing stores (again, very similar to Kubernetes persistent volumes).

Then you need to create Flocker datasets, and at that point you're ready to hook it up as a persistent volume. After all your hard work, this part is easy and you just need to specify the Flocker dataset name:

```
apiVersion: v1
kind: Pod
metadata:
  name: flocker-web
spec:
  containers:
    - name: web
      image: nginx
      ports:
        - name: web
          containerPort: 80
      volumeMounts:
        # name must match the volume name below
        - name: www-root
          mountPath: "/usr/share/nginx/html"
  volumes:
    - name: www-root
      flocker:
        datasetName: my-flocker-vol
```

Integrating enterprise storage into Kubernetes

If you have an existing **Storage Area Network (SAN)** exposed over the iSCSI interface, Kubernetes has a volume plugin for you. It follows the same model as other shared persistent storage plugins we've seen earlier. It supports the following features:

- Connect to one portal
- Mount a device directly or via multipathd
- Format and partition any new device
- Authenticate via CHAP

You must configure the iSCSI initiator, but you don't have to provide any initiator information. All you need to provide is the following:

- The IP address of the iSCSI target and port (if not the default 3260)
- The target's iqn (an iSCSI-qualified name) – typically the reversed domain name
- LUN – the logical unit number
- The filesystem type
- A read-only Boolean flag

The iSCSI plugin supports `ReadWriteOnce` and `ReadOnlyMany`. Note that you can't partition your device at this time. Here is the volume spec:

```
volumes:
- name: iscsi-volume
  iscsi:
    targetPortal: 10.0.2.34:3260
    iqn: iqn.2001-04.com.example:storage.kube.sys1.xyz
    lun: 0
    fsType: ext4
    readOnly: true
```

Rook – the new kid on the block

Rook is an open source cloud-native storage orchestrator. It is currently an incubating CNCF project. It provides a consistent experience on top of multiple storage solutions including Ceph, edgeFS, Cassandra, Minio, NFS, CockroachDB, and YugabyteDB (although only Ceph and EdgeFS support is considered stable). Here are the features Rook provides:

- Scheduling
- Life cycle management
- Resource management
- Monitoring

Rook takes advantage of modern Kubernetes' best practices like CRDs and operators. Once you install the Rook operator, you can create a Ceph cluster using a Rook CRD as follows:

```
apiVersion: ceph.rook.io/v1
kind: CephCluster
```

```
metadata:
  name: rook-ceph
  namespace: rook-ceph
spec:
  cephVersion:
    # For the latest ceph images, see https://hub.docker.com/r/ceph/ceph/tags
    image: ceph/ceph:v14.2.4-20190917
    dataDirHostPath: /var/lib/rook
  mon:
    count: 3
  dashboard:
    enabled: true
  storage:
    useAllNodes: true
    useAllDevices: false
    # Important: Directories should only be used in pre-production
    # environments
    directories:
      - path: /var/lib/rook
```

Note that the Rook framework itself is still considered alpha software. It is definitely a project to watch even if you decide not to use it right away.

Projecting volumes

It's possible to project multiple volumes into a single directory, so they appear as a single volume. The supported volume types are Kubernetes-managed: secret, downwardAPI, and configMap. This is useful if you want to mount multiple sources of configuration into a pod. Instead of having to create a separate volume for each source, you can bundle all of them into a single projected volume. Here is an example:

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  containers:
    - name: the-container
      image: busybox
      volumeMounts:
```

```

- name: all-in-one
  mountPath: "/projected-volume"
  readOnly: true
volumes:
- name: all-in-one
  projected:
    sources:
    - secret:
        name: the-secret
        items:
        - key: username
          path: the-group/the-user
    - downwardAPI:
        items:
        - path: "labels"
          fieldRef:
            fieldPath: metadata.labels
        - path: "cpu\_limit"
          resourceFieldRef:
            containerName: the-container
            resource: limits.cpu
    - configMap:
        name: the-configmap
        items:
        - key: config
          path: the-group/the-config

```

Using out-of-tree volume plugins with FlexVolume

FlexVolume became generally available in Kubernetes 1.8. It allows you to consume out-of-tree storage through a uniform API. Storage providers write a driver that you install on all nodes. The FlexVolume plugin can dynamically discover existing drivers. Here is an example of using a FlexVolume to bind to an external NFS volume:

```

apiVersion: v1
kind: Pod
metadata:
  name: nginx-nfs
  namespace: default

```



```
spec:
  containers:
  - name: nginx-nfs
    image: nginx
    volumeMounts:
    - name: test
      mountPath: /data
    ports:
    - containerPort: 80
  volumes:
  - name: test
    flexVolume:
      driver: "k8s/nfs"
      fsType: "nfs"
      options:
        server: "172.16.0.25"
        share: "dws\_nas\_scratch"
```

However, at this point I highly recommend you avoid using the FlexVolume plugin and utilize CSI plugins instead.

The Container Storage Interface

The **Container Storage Interface (CSI)** is an initiative to standardize the interaction between container orchestrators and storage providers. It is driven by Kubernetes, Docker, Mesos, and Cloud Foundry. The idea is that storage providers implement just one CSI driver and container orchestrators need to support only the CSI. It is the equivalent of CNI for storage. There are several advantages over the FlexVolume approach:

- CSI is an industry-wide standard
- New capabilities are made available for CSI plugins only (such as volume snapshots and volume cloning)
- FlexVolume plugins require access to the node and master root filesystem to deploy drivers
- FlexVolume's storage driver often requires many external dependencies
- FlexVolume's EXEC-style interface is clunky

A CSI volume plugin was added in Kubernetes 1.9 as an alpha feature and is generally available since Kubernetes 1.13. FlexVolume will remain for backward compatibility, at least for a while.

Here is a diagram that demonstrates how CSI works within Kubernetes:

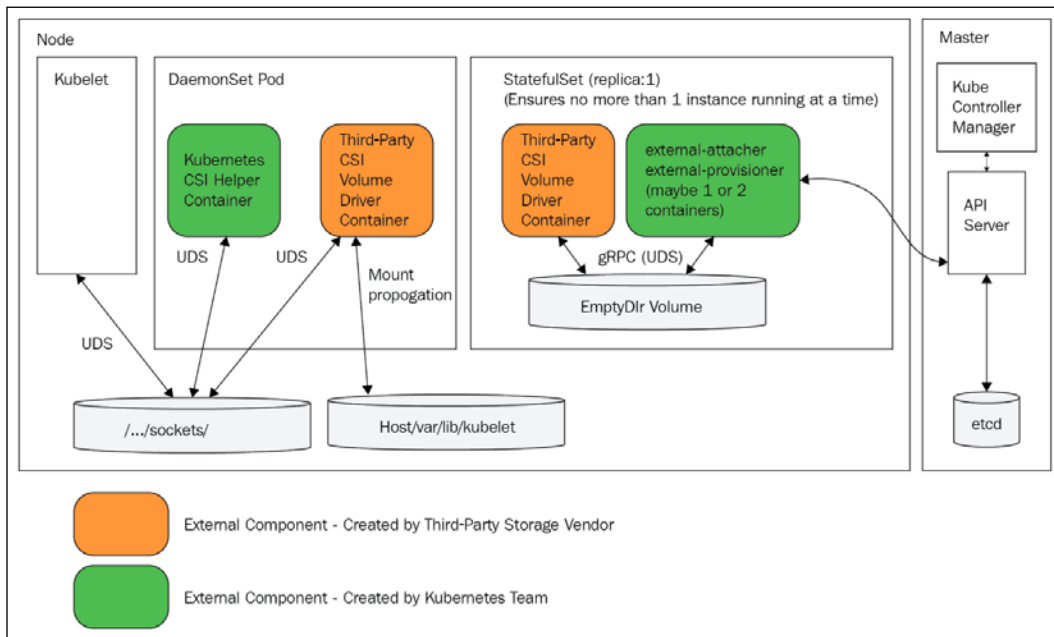


Figure 6.5: CSI within Kubernetes

There is currently a major migration effort to port all in-tree plugins to out-of-tree CSI drivers.

See <https://kubernetes-csi.github.io> for more details.

Volume snapshotting and cloning

These features are available only to CSI drivers. They represent the benefits of a uniform storage model that allows adding optional advanced functionality across all storage providers with a uniform interface.

Volume snapshots

Volume snapshots are in alpha status as of Kubernetes 1.12. They are exactly what they sound like: a snapshot of a volume at a certain point in time. You can create and later restore volumes from a snapshot. It's interesting that the API objects associated with snapshots are CRDs and not part of the core Kubernetes API. The objects are:

- VolumeSnapshotClass
- VolumeSnapshotContents
- VolumeSnapshot

Volume snapshots work using an external-prosnpshotter sidecar container that the Kubernetes team developed. It watches for snapshot CRDs to be created and interacts with the snapshot controller, which can invoke the `CreateSnapshot` and `DeleteSnapshot` operations of CSI drivers that implement snapshot support.

You can also provision volumes from a snapshot.

Here is a persistent volume claim bound to a snapshot:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: restore-pvc
spec:
  storageClassName: csi-hostpath-sc
  dataSource:
    name: new-snapshot-test
    kind: VolumeSnapshot
    apiGroup: snapshot.storage.k8s.io
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
```

Volume cloning

Volume cloning is available in beta status as of Kubernetes 1.16. Volume clones are new volumes that are populated with the content of an existing volume. Once the volume cloning is complete, there is no relation between the original and clone. Their context will diverge over time. You can perform a clone by creating a snapshot and then creating a new volume from the snapshot. But volume cloning is more streamlined and efficient.

Volume cloning must be enabled with a feature gate: `--feature-gates=VolumePVCDDataSource=true`

It works just for dynamic provisioning and uses the storage class of the source volume for the clone as well. You initiate a volume clone by specifying an existing persistent volume claim as the data source of a new persistent volume claim. That triggers dynamic provisioning of a new volume that clones the source claim's volume:

```
apiVersion: v1
```

```
kind: PersistentVolumeClaim
metadata:
  name: clone-of-pvc-1
  namespace: myns
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: cloning
  resources:
    requests:
      storage: 5Gi
  dataSource:
    kind: PersistentVolumeClaim
    name: pvc-1
```

Summary

In this chapter, we took a deep look into storage in Kubernetes. We've looked at the generic conceptual model based on volumes, claims, and storage classes, as well as the implementation of volume plugins. Kubernetes eventually maps all storage systems into mounted filesystems in containers or devices of raw block storage. This straightforward model allows administrators to configure and hook up any storage system from localhost directories through cloud-based shared storage all the way to enterprise storage systems. The transition of storage provisioners from in-tree to CSI-based out-of-tree drivers bodes well for the storage ecosystem. You should now have a clear understanding of how storage is modeled and implemented in Kubernetes and be able to make intelligent choices regarding how to implement storage in your Kubernetes cluster.

In *Chapter 7, Running Stateful Applications with Kubernetes*, we'll see how Kubernetes can raise the level of abstraction and, on top of storage, how it can help in developing, deploying, and operating stateful applications using concepts such as stateful sets.

7

Running Stateful Applications with Kubernetes

In this chapter, we will learn how to run stateful applications on Kubernetes. Kubernetes takes a lot of work out of our hands by automatically starting and restarting pods across the cluster nodes as needed, based on complex requirements and configurations such as namespaces, limits, and quotas. But when pods run storage-aware software, such as databases and queues, relocating a pod can cause the system to break. First, we'll explore the essence of stateful pods and why they are much more complicated to manage in Kubernetes. We will look at a few ways to manage the complexity, such as shared environment variables and DNS records. In some situations, a redundant in-memory state, a DaemonSet, or persistent storage claims can do the trick. The main solution that Kubernetes promotes for state-aware pods is the StatefulSet (previously called PetSet) resource, which allows us to manage an indexed collection of pods with stable properties. Finally, we will dive deep into a full-fledged example of running a Cassandra cluster on top of Kubernetes.

Stateful versus stateless applications in Kubernetes

A stateless Kubernetes application is an application that doesn't manage its state in the Kubernetes cluster. All of the state is stored outside the cluster and the cluster containers access it in some manner. In this section, we'll learn why state management is critical to the design of a distributed system and the benefits of managing state within the Kubernetes cluster.

Understanding the nature of distributed data-intensive apps

Let's start from the basics here. Distributed applications are a collection of processes that run on multiple machines, process inputs, manipulate data, expose APIs, and possibly have other side effects. Each process is a combination of its program, its runtime environment, and its inputs and outputs. The programs you write at school get their input as command-line arguments, maybe they read a file or access a database, and then write their results to the screen or a file or a database. Some programs keep state in memory and can serve requests over the network. Simple programs run on a single machine and can hold all their state in memory or read from a file. Their runtime environment is their operating system. If they crash, the user has to restart them manually. They are tied to their machine. A distributed application is a different animal. A single machine is not enough to process all the data or serve all the requests quickly enough. A single machine can't hold all the data. The data that needs to be processed is so large that it can't be downloaded cost-effectively into each processing machine. Machines can fail and need to be replaced. Upgrades need to be performed over all the processing machines. Users may be distributed across the globe.

Taking all these issues into account, it becomes clear that the traditional approach doesn't work. The limiting factor becomes the data. Users/clients must receive only summary or processed data. All massive data processing must be done close to the data itself because transferring data is prohibitively slow and expensive. Instead, the bulk of processing code must run in the same data center and network environment of the data.

Why manage state in Kubernetes?

The main reason to manage state in Kubernetes itself as opposed to a separate cluster is that a lot of the infrastructure needed to monitor, scale, allocate, secure, and operate a storage cluster is already provided by Kubernetes. Running a parallel storage cluster will lead to a lot of duplicated effort.

Why manage state outside of Kubernetes?

Let's not rule out the other option. It may be better in some situations to manage state in a separate non-Kubernetes cluster, as long as it shares the same internal network (data proximity trumps everything).

Some valid reasons are as follows:

- You already have a separate storage cluster and you don't want to rock the boat
- Your storage cluster is used by other non-Kubernetes applications
- Kubernetes support for your storage cluster is not stable or mature enough
- You may want to approach stateful applications in Kubernetes incrementally, starting with a separate storage cluster and integrating more tightly with Kubernetes later

Shared environment variables versus DNS records for discovery

Kubernetes provides several mechanisms for global discovery across the cluster. If your storage cluster is not managed by Kubernetes, you still need to tell Kubernetes pods how to find it and access it. There are two common methods:

- DNS
- Environment variables

In some cases, you may want to use both, as environment variables can override DNS.

Accessing external data stores via DNS

The DNS approach is simple and straightforward. Assuming your external storage cluster is load balanced and can provide a stable endpoint, then pods can just hit that endpoint directly and connect to the external cluster.

Accessing external data stores via environment variables

Another simple approach is to use environment variables to pass connection information to an external storage cluster. Kubernetes offers the ConfigMap resource as a way to keep configuration separate from the container image. The configuration is a set of key-value pairs. The configuration information can be exposed as an environment variable inside the container as well as volumes. You may prefer to use secrets for sensitive connection information.

Creating a ConfigMap

The following file is a ConfigMap that keeps a list of addresses:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: db-config
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8
```

Save it as `db-config-map.yaml` and run:

```
$ kubectl create -f db-config-map.yaml
configmap/db-config created
```

The data section contains all the key-value pairs (in this case, just a single pair with a key name of `db-ip-addresses`). It will be important later when consuming the ConfigMap in a pod. You can check out the content to make sure it's OK:

```
$ kubectl get configmap db-config -o yaml
apiVersion: v1
data:
  db-ip-addresses: 1.2.3.4,5.6.7.8
kind: ConfigMap
metadata:
  creationTimestamp: "2020-06-08T14:25:39Z"
  name: db-config
  namespace: default
  resourceVersion: "366427"
  selfLink: /api/v1/namespaces/default/configmaps/db-config
  uid: 2d0a357a-e38e-11e9-90a4-0242ac120002
```

There are other ways to create a ConfigMap. You can directly create them using the `--from-value` or `--from-file` command-line arguments.

Consuming a ConfigMap as an environment variable

When you are creating a pod, you can specify a ConfigMap and consume its values in several ways. Here is how to consume our configuration map as an environment variable:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod
spec:
  containers:
  - name: some-container
    image: busybox
    command: ["/bin/sh", "-c", "env"]
    env:
    - name: DB_IP_ADDRESSES
      valueFrom:
        configMapKeyRef:
          name: db-config
          key: db-ip-addresses
    restartPolicy: Never
```

This pod runs the busybox minimal container and executes an `env` bash command and immediately exits. The `db-ip-addresses` key from `db-configmap` is mapped to the `DB_IP_ADDRESSES` environment variable, and is reflected in the output:

```
$ kubectl create -f pod-with-db.yaml
pod/some-pod created
```

```
$ kubectl logs some-pod | grep DB_IP
DB_IP_ADDRESSES=1.2.3.4,5.6.7.8
```

Using a redundant in-memory state

In some cases, you may want to keep a transient state in memory. Distributed caching is a common case. Time-sensitive information is another one. For these use cases, there is no need for persistent storage, and multiple pods accessed through a service may be just the right solution. We can use standard Kubernetes techniques, such as labeling, to identify pods that belong to the distributed cache, store redundant copies of the same state, and expose them through a service. If a pod dies, Kubernetes will create a new one and, until it catches up, the other pods will serve the state. We can even use the pod's anti-affinity feature to ensure that pods that maintain redundant copies of the same state are not scheduled to the same node.

Of course, you could also use something like Memcached or Redis.

Using DaemonSet for redundant persistent storage

Some stateful applications, such as distributed databases or queues, manage their state redundantly and sync their nodes automatically (we'll take a very deep look into Cassandra later). In these cases, it is important that pods are scheduled to separate nodes. It is also important that pods are scheduled to nodes with a particular hardware configuration or are even dedicated to the stateful application. The DaemonSet feature is perfect for this use case. We can label a set of nodes and make sure that the stateful pods are scheduled on a one-by-one basis to the selected group of nodes.

Applying persistent volume claims

If the stateful application can use effectively shared persistent storage, then using a persistent volume claim in each pod is the way to go, as we demonstrated in *Chapter 6, Managing Storage*. The stateful application will be presented with a mounted volume that looks just like a local filesystem.

Utilizing StatefulSets

StatefulSets are especially designed to support distributed stateful applications where the identities of the members are important, and if a pod is restarted it must retain its identity in the set. It provides ordered deployment and scaling. Unlike regular pods, the pods of a StatefulSet are associated with persistent storage.

When to use a StatefulSet

StatefulSets are great for applications that require one or more of the following:

- Stable, unique network identifiers
- Stable, persistent storage
- Ordered, graceful deployment and scaling
- Ordered, graceful deletion and termination

The components of a StatefulSet

There are several pieces that need to be configured correctly in order to have a working StatefulSet:

- A headless service responsible for managing the network identity of the StatefulSet pods

- The StatefulSet itself with a number of replicas
- Persistent storage provisioned dynamically or by an administrator

Here is an example of a headless service called `nginx` that will be used for a StatefulSet:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
```

Now, the StatefulSet configuration file will reference the service:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  serviceName: "nginx"
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

The next part is the pod template, which includes a mounted volume named `www`:

```
spec:
  terminationGracePeriodSeconds: 1800
  containers:
  - name: nginx
    image: gcr.io/google_containers/nginx-slim:0.8
    imagePullPolicy: Always
    ports:
    - containerPort: 80
      name: web
    volumeMounts:
    - name: www
      mountPath: /usr/share/nginx/html
```

Last but not least, `volumeClaimTemplates` uses a claim named `www` matching the mounted volume. The claim requests 1 GiB of storage with `ReadWriteOnce` access:

```
volumeClaimTemplates:
- metadata:
  name: www
  spec:
    accessModes: ["ReadWriteOnce"]
    resources:
      requests:
        storage: 1Gi
```

Running a Cassandra cluster in Kubernetes

In this section, we will explore in detail a very large example of configuring a Cassandra cluster to run on a Kubernetes cluster. The full example can be accessed here:

<https://kubernetes.io/docs/tutorials/stateful-application/cassandra/>

First, we'll learn a little bit about Cassandra and its idiosyncrasies, and then follow a step-by-step procedure to get it running using several of the techniques and strategies we've covered in the previous section.

Quick introduction to Cassandra

Cassandra is a distributed columnar data store. It was designed from the get-go for big data. Cassandra is fast, robust (no single point of failure), highly available, and linearly scalable. It also has multi-datacenter support. It achieves all this by having a laser focus and carefully crafting the features it supports – and just as importantly – the features it doesn't support. In a previous company, I ran a Kubernetes cluster that used Cassandra as the main data store for sensor data (about 100 TB). Cassandra allocates the data to a set of nodes (node ring) based on a **distributed hash table (DHT)** algorithm. The cluster nodes talk to each other via a gossip protocol and learn quickly about the overall state of the cluster (what nodes joined and what nodes left or are unavailable). Cassandra constantly compacts the data and balances the cluster. The data is typically replicated multiple times for redundancy, robustness, and high availability. From a developer's point of view, Cassandra is very good for time-series data and provides a flexible model where you can specify the consistency level in each query. It is also idempotent (a very important feature for a distributed database), which means repeated inserts or updates are allowed.

Here is a diagram that shows how a Cassandra cluster is organized and how a client can access any node and how the request will be forwarded automatically to the nodes that have the requested data:

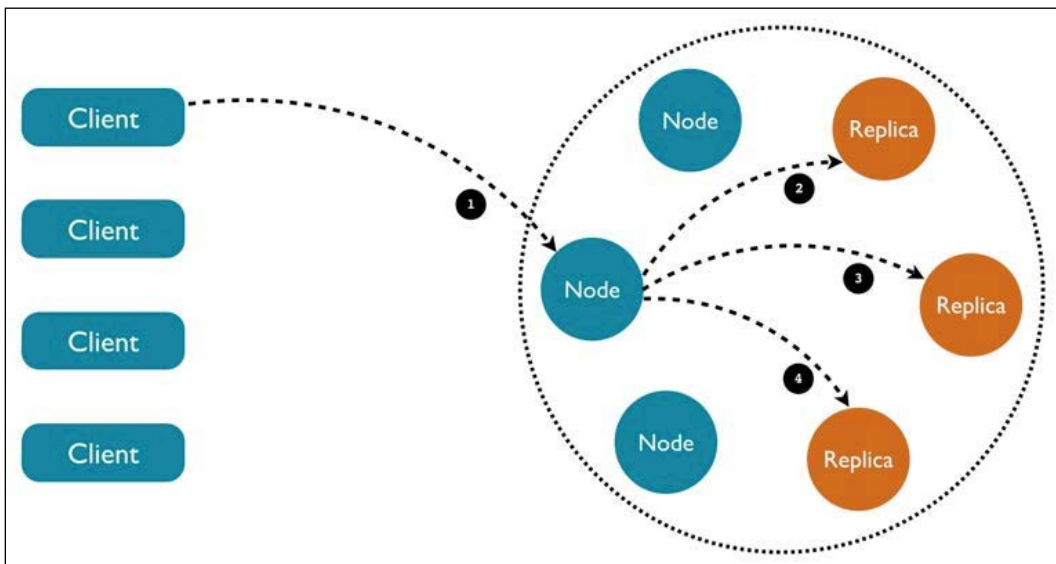


Figure 7.1: Request interacting with a Cassandra cluster

The Cassandra Docker image

Deploying Cassandra on Kubernetes as opposed to a standalone Cassandra cluster deployment requires a special Docker image. This is an important step because it means we can use Kubernetes to keep track of our Cassandra pods. The image is available here:

<https://github.com/kubernetes/examples/blob/master/cassandra/image/Dockerfile>

The Dockerfile is coming up. The base image is a flavor of Debian designed for use in containers (see <https://github.com/kubernetes/kubernetes/tree/master/build/debian-base>).

The Cassandra Dockerfile defines some build arguments that must be set when the image is built, creates a bunch of labels, defines many environment variables, adds all the files to the root directory inside the container, runs the `build.sh` script, declares the Cassandra data volume (where the data is stored), exposes a bunch of ports, and finally uses `dumb-init` to execute the `run.sh` script:

```
FROM k8s.gcr.io/debian-base-amd64:0.3

ARG BUILD_DATE
ARG VCS_REF
ARG CASSANDRA_VERSION
ARG DEV_CONTAINER

LABEL \
    org.label-schema.build-date=$BUILD_DATE \
    org.label-schema.docker.dockerfile="/Dockerfile" \
    org.label-schema.license="Apache License 2.0" \
    org.label-schema.name="k8s-for-greeks/docker-cassandra-k8s" \
    org.label-schema.url="https://github.com/k8s-for-greeks/" \
    org.label-schema.vcs-ref=$VCS_REF \
    org.label-schema.vcs-type="Git" \
    org.label-schema.vcs-url="https://github.com/k8s-for-greeks/docker-cassandra-k8s"

ENV CASSANDRA_HOME=/usr/local/apache-cassandra-${CASSANDRA_VERSION} \
    CASSANDRA_CONF=/etc/cassandra \
    CASSANDRA_DATA=/cassandra_data \
    CASSANDRA_LOGS=/var/log/cassandra \
    JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64 \
    PATH=${PATH}:/usr/lib/jvm/java-8-openjdk-amd64/bin:/usr/local/
```

```
apache-cassandra-${CASSANDRA_VERSION}/bin

ADD files /

RUN clean-install bash \
    && /build.sh \
    && rm /build.sh

VOLUME ["/$CASSANDRA_DATA"]

# 7000: intra-node communication
# 7001: TLS intra-node communication
# 7199: JMX
# 9042: CQL
# 9160: thrift service
EXPOSE 7000 7001 7199 9042 9160

CMD ["/usr/bin/dumb-init", "/bin/bash", "/run.sh"]
```

Here are all the files used by the Dockerfile:

- build.sh
- cassandra-seed.h
- cassandra.yaml
- jvm.options
- kubernetes-cassandra.jar
- logback.xml
- ready-probe.sh
- run.sh

We will not cover all of them; we'll focus on the build.sh and run.sh scripts.

Exploring the build.sh script

Cassandra is a Java program. The build script installs the Java runtime environment and a few necessary libraries and tools. It then sets a few variables that will be used later, such as CASSANDRA_PATH.

It downloads the correct version of Cassandra from the Apache organization (Cassandra is an Apache open source project), creates the `/cassandra_data/` data directory where Cassandra will store its SSTables and the `/etc/cassandra` configuration directory, copies files into the configuration directory, adds a Cassandra user, sets the readiness probe, installs Python, moves the Cassandra jar file and the seed shared library to their target destination, and then cleans up all the intermediate files generated during this process:

```
apt-get update && apt-get dist-upgrade -y

clean-install \
    openjdk-8-jre-headless \
    libjemalloc1 \
    localepurge \
    dumb-init \
    wget

CASSANDRA_PATH="cassandra/${CASSANDRA_VERSION}/apache-cassandra-
${CASSANDRA_VERSION}-bin.tar.gz"
CASSANDRA_DOWNLOAD=http://www.apache.org/dyn/closer.
cgi?path=${CASSANDRA_PATH}&as_json=1
CASSANDRA_MIRROR='wget -q -O - ${CASSANDRA_DOWNLOAD} | grep -oP
"(<=|"preferred": \")[^\"]+"'

echo "Downloading Apache Cassandra from ${CASSANDRA_MIRROR}${CASSANDRA_
PATH}..."
wget -q -O - ${CASSANDRA_MIRROR}${CASSANDRA_PATH} \
    | tar -xzf - -C /usr/local

mkdir -p /cassandra_data/data
mkdir -p /etc/Cassandra

mv /logback.xml /cassandra.yaml /jvm.options /etc/cassandra/
mv /usr/local/apache-cassandra-${CASSANDRA_VERSION}/conf/cassandra-env.
sh /etc/cassandra/

adduser --disabled-password --no-create-home --gecos '' --disabled-
login cassandra
chmod +x /ready-probe.sh
chown cassandra: /ready-probe.sh

DEV_IMAGE=${DEV_CONTAINER:-}
if [ ! -z "$DEV_IMAGE" ]; then
```

```

    clean-install python;
else
    rm -rf $CASSANDRA_HOME/pylib;
fi

mv /kubernetes-cassandra.jar /usr/local/apache-cassandra-${CASSANDRA_
VERSION}/lib
mv /cassandra-seed.so /etc/cassandra/
mv /cassandra-seed.h /usr/local/lib/include

apt-get -y purge localepurge
apt-get -y autoremove
apt-get clean

rm <many files and directories>

```

Exploring the run.sh script

The `run.sh` script requires some shell skills and knowledge of Cassandra to understand, but it's worth the effort.

First, some local variables are set for the Cassandra configuration file at `/etc/cassandra/cassandra.yaml`. The `CASSANDRA_CFG` variable will be used in the rest of the script:

```

set -e
CASSANDRA_CONF_DIR=/etc/Cassandra
CASSANDRA_CFG=$CASSANDRA_CONF_DIR/cassandra.yaml

```

If no `CASSANDRA_SEEDS` were specified, then set the `HOSTNAME`, which is used by the `StatefulSet` later:

```

# we are doing StatefulSet or just setting our seeds
if [ -z "$CASSANDRA_SEEDS" ]; then
    HOSTNAME=$(hostname -f)
    CASSANDRA_SEEDS=$(hostname -f)
fi

```

Then comes a long list of environment variables with defaults. The syntax `${VAR_NAME:-}` uses the `VAR_NAME` environment variable, if it's defined, or the default value.

A similar syntax, `${VAR_NAME:=}`, does the same thing, but also assigns the default value to the environment variable if it's not defined. This is a subtle but important difference.

Both variations are used here:

```
# The following vars relate to their counter parts in $CASSANDRA_CFG
# for instance rpc_address
CASSANDRA_RPC_ADDRESS="${CASSANDRA_RPC_ADDRESS:-0.0.0.0}"
CASSANDRA_NUM_TOKENS="${CASSANDRA_NUM_TOKENS:-32}"
CASSANDRA_CLUSTER_NAME="${CASSANDRA_CLUSTER_NAME:='Test Cluster'}"
CASSANDRA_LISTEN_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_BROADCAST_RPC_ADDRESS=${POD_IP:-$HOSTNAME}
CASSANDRA_DISK_OPTIMIZATION_STRATEGY="${CASSANDRA_DISK_OPTIMIZATION_
TRATEGY:-ssd}"
CASSANDRA_MIGRATION_WAIT="${CASSANDRA_MIGRATION_WAIT:-1}"
CASSANDRA_ENDPOINT_SNITCH="${CASSANDRA_ENDPOINT_SNITCH:-SimpleSnitch}"
CASSANDRA_DC="${CASSANDRA_DC}"
CASSANDRA_RACK="${CASSANDRA_RACK}"
CASSANDRA_RING_DELAY="${CASSANDRA_RING_DELAY:-30000}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:-true}"
CASSANDRA_SEEDS="${CASSANDRA_SEEDS:false}"
CASSANDRA_SEED_PROVIDER="${CASSANDRA_SEED_PROVIDER:-org.apache.
cassandra.locator.SimpleSeedProvider}"
CASSANDRA_AUTO_BOOTSTRAP="${CASSANDRA_AUTO_BOOTSTRAP:false}"
```

By the way, I contributed my part to Kubernetes by opening a pull request to fix a minor typo here. See <https://github.com/kubernetes/examples/pull/348>.

The next part configures monitoring **Java Management Exceptions (JMX)** and controls garbage collection output:

```
# Turn off JMX auth
CASSANDRA_OPEN_JMX="${CASSANDRA_OPEN_JMX:-false}"
# send GC to STDOUT
CASSANDRA_GC_STDOUT="${CASSANDRA_GC_STDOUT:-false}"
```

Then comes a section where all the variables are printed to the screen. Let's skip most of it:

```
echo Starting Cassandra on ${CASSANDRA_LISTEN_ADDRESS}
echo CASSANDRA_CONF_DIR ${CASSANDRA_CONF_DIR}
echo CASSANDRA_CFG ${CASSANDRA_CFG}
echo CASSANDRA_AUTO_BOOTSTRAP ${CASSANDRA_AUTO_BOOTSTRAP}
...
```

The next section is very important. By default, Cassandra uses a simple snitch, which is unaware of racks and data centers. This is not optimal when the cluster spans multiple data centers and racks.

Cassandra is rack-aware and datacenter-aware and can optimize both for redundancy and high availability while limiting communication across data centers appropriately:

```
# if DC and RACK are set, use GossipingPropertyFileSnitch
if [[ $CASSANDRA_DC && $CASSANDRA_RACK ]]; then
    echo "dc=$CASSANDRA_DC" > $CASSANDRA_CONF_DIR/cassandra-rackdc.
properties
    echo "rack=$CASSANDRA_RACK" >> $CASSANDRA_CONF_DIR/cassandra-rackdc.
properties
    CASSANDRA_ENDPOINT_SNITCH="GossipingPropertyFileSnitch"
fi
```

Memory management is also important, and you can control the maximum heap size to ensure Cassandra doesn't start thrashing and swapping to disk:

```
if [ -n "$CASSANDRA_MAX_HEAP" ]; then
    sed -ri "s/^(#)?-Xmx[0-9]+.*-/Xmx$CASSANDRA_MAX_HEAP/" "$CASSANDRA_
CONF_DIR/jvm.options"
    sed -ri "s/^(#)?-Xms[0-9]+.*-/Xms$CASSANDRA_MAX_HEAP/" "$CASSANDRA_
CONF_DIR/jvm.options"
fi

if [ -n "$CASSANDRA_REPLACE_NODE" ]; then
    echo "-Dcassandra.replace_address=$CASSANDRA_REPLACE_NODE/" >>
"$CASSANDRA_CONF_DIR/jvm.options"
fi
```

The rack and data center information is stored in a simple Java properties file:

```
for rackdc in dc rack; do
    var="CASSANDRA_${rackdc^^}"
    val="${!var}"
    if [ "$val" ]; then
        sed -ri 's/^(("$rackdc"=).*\/\1 "$val"\/' "$CASSANDRA_CONF_DIR/
cassandra-rackdc.properties"
    fi
done
```

The next section loops over all the variables defined earlier, finds the corresponding key in the `Cassandra.yaml` configuration files, and overwrites them. That ensures that each configuration file is customized on the fly just before it launches Cassandra:

```
for yaml in \
    broadcast_address \
    broadcast_rpc_address \
    cluster_name \
    disk_optimization_strategy \
    endpoint_snitch \
    listen_address \
    num_tokens \
    rpc_address \
    start_rpc \
    key_cache_size_in_mb \
    concurrent_reads \
    concurrent_writes \
    memtable_cleanup_threshold \
    memtable_allocation_type \
    memtable_flush_writers \
    concurrent_compactors \
    compaction_throughput_mb_per_sec \
    counter_cache_size_in_mb \
    internode_compression \
    endpoint_snitch \
    gc_warn_threshold_in_ms \
    listen_interface \
    rpc_interface \
; do
    var="CASSANDRA_${yaml^^}"
    val="${!var}"
    if [ "$val" ]; then
        sed -ri 's/^(# )?("$yaml":).*/\2 "$val"/' "$CASSANDRA_CFG"
    fi
done

echo "auto_bootstrap: ${CASSANDRA_AUTO_BOOTSTRAP}" >> $CASSANDRA_CFG
```

The next section is all about setting the seeds or seed provider depending on the deployment solution (StatefulSet or not). There is a little trick for the first pod to bootstrap as its own seed:

```
# set the seed to itself. This is only for the first pod, otherwise
# it will be able to get seeds from the seed provider
if [[ $CASSANDRA_SEEDS == 'false' ]]; then
    sed -ri 's/- seeds:.*-/ seeds: "'"$POD_IP"'"/' $CASSANDRA_CFG
else # if we have seeds set them. Probably StatefulSet
    sed -ri 's/- seeds:.*-/ seeds: "'"$CASSANDRA_SEEDS"'"/' $CASSANDRA_CFG
fi

sed -ri 's/- class_name: SEED_PROVIDER/- class_name: "'"$CASSANDRA_SEED_PROVIDER"'/' $CASSANDRA_CFG
```

The following section sets up various options for remote management and JMX monitoring. It's critical in complicated distributed systems to have proper administration tools.

Cassandra has deep support for the ubiquitous JMX standard:

```
# send gc to stdout
if [[ $CASSANDRA_GC_STDOUT == 'true' ]]; then
    sed -ri 's/ -Xloggc:\var\log\cassandra\gc.log/' $CASSANDRA_CONF_DIR/cassandra-env.sh
fi

# enable RMI and JMX to work on one port
echo "JVM_OPTS=\"\$JVM_OPTS -Djava.rmi.server.hostname=$POD_IP\"" >>
$CASSANDRA_CONF_DIR/cassandra-env.sh

# getting WARNING messages with Migration Service
echo "-Dcassandra.migration_task_wait_in_seconds=${CASSANDRA_MIGRATION_WAIT}" >> $CASSANDRA_CONF_DIR/jvm.options
echo "-Dcassandra.ring_delay_ms=${CASSANDRA_RING_DELAY}" >> $CASSANDRA_CONF_DIR/jvm.options

if [[ $CASSANDRA_OPEN_JMX == 'true' ]]; then
    export LOCAL_JMX=no
    sed -ri 's/ -Dcom.sun.management.jmxremote.authenticate=true/-Dcom.sun.management.jmxremote.authenticate=false/' $CASSANDRA_CONF_DIR/cassandra-env.sh
    sed -ri 's/ -Dcom.sun.management.jmxremote.password.file=\etc\cassandra\jmxremote.password/' $CASSANDRA_CONF_DIR/cassandra-env.sh
fi
```

Finally, it protects the data directory such that only the `cassandra` user can access it, the `CLASSPATH` is set to the Cassandra jar file, and it launches Cassandra in the foreground (not daemonized) as the `cassandra` user:

```
chmod 700 "${CASSANDRA_DATA}"
chown -c -R cassandra "${CASSANDRA_DATA}" "${CASSANDRA_CONF_DIR}"

export CLASSPATH=/kubernetes-cassandra.jar

su cassandra -c "${CASSANDRA_HOME}/bin/cassandra -f"
```

Hooking up Kubernetes and Cassandra

Connecting Kubernetes and Cassandra takes some work because Cassandra was designed to be very self-sufficient, but we want to let it hook into Kubernetes at the right time to provide capabilities such as automatically restarting failed nodes, monitoring, allocating Cassandra pods, and providing a unified view of the Cassandra pods side by side with other pods.

Cassandra is a complicated beast and has many knobs to control it. It comes with a `Cassandra.yaml` configuration file, and you can override all the options with environment variables.

Digging into the Cassandra configuration file

There are two settings that are particularly relevant: the seed provider and the snitch. The seed provider is responsible for publishing a list of IP addresses (seeds) of nodes in the cluster. Every node that starts running connects to the seeds (there are usually at least three) and if it successfully reaches one of them they immediately exchange information about all the nodes in the cluster. This information is updated constantly for each node as the nodes gossip with each other.

The default seed provider configured in `Cassandra.yaml` is just a static list of IP addresses, in this case just the loopback interface:

```
# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and
    learn
    # the topology of the ring. You must change this if you are
    running
```

```
# multiple nodes!
#- class_name: io.k8s.cassandra.KubernetesSeedProvider
- class_name: SEED_PROVIDER
  parameters:
    # seeds is actually a comma-delimited list of addresses.
    # Ex: "<ip1>,<ip2>,<ip3>"
    - seeds: "127.0.0.1"
```

The other important setting is the snitch. It has two roles: it teaches Cassandra enough about your network topology to route requests efficiently, and it allows Cassandra to spread replicas around your cluster to avoid correlated failures. It does this by grouping machines into data centers and racks. Cassandra will do its best not to have more than one replica on the same rack (which may not actually be a physical location).

Cassandra comes pre-loaded with several snitch classes, but none of them are Kubernetes-aware. The default is `SimpleSnitch`, but it can be overridden:

```
# You can use a custom Snitch by setting this to the full class
# name of the snitch, which will be assumed to be on your classpath.
endpoint_snitch: SimpleSnitch
```

Other snitches are:

- `GossipingPropertyFileSnitch`
- `PropertyFileSnitch`
- `Ec2Snitch`
- `Ec2MultiRegionSnitch`
- `RackInferringSnitch`

The custom seed provider

When running Cassandra nodes as pods in Kubernetes, Kubernetes may move pods around, including seeds. To accommodate that, a Cassandra seed provider needs to interact with the Kubernetes API server.

Here is a short snippet from the custom `KubernetesSeedProvider` Java class that implements the Cassandra `SeedProvider` API:

```
public class KubernetesSeedProvider implements SeedProvider {

    ...
```



```
/**
 * Call Kubernetes API to collect a list of seed providers
 *
 * @return List of seed providers
 */
public List<InetAddress> getSeeds() {
    GoInterface go = (GoInterface) Native.loadLibrary("cassandra-seed.
so", GoInterface.class);

    String service = getEnvOrDefault("CASSANDRA_SERVICE", "cassandra");
    String namespace = getEnvOrDefault("POD_NAMESPACE", "default");

    String initialSeeds = getEnvOrDefault("CASSANDRA_SEEDS", "");

    if ("".equals(initialSeeds)) {
        initialSeeds = getEnvOrDefault("POD_IP", "");
    }

    String seedSizeVar = getEnvOrDefault("CASSANDRA_SERVICE_NUM_SEEDS",
"8");
    Integer seedSize = Integer.valueOf(seedSizeVar);

    String data = go.GetEndpoints(namespace, service, initialSeeds);
    ObjectMapper mapper = new ObjectMapper();

    try {
        Endpoints = mapper.readValue(data, Endpoints.class);
        logger.info("cassandra seeds: {}", endpoints.ips.toString());
        return Collections.unmodifiableList(endpoints.ips);
    } catch (IOException e) {
        // This should not happen
        logger.error("unexpected error building cassandra seeds: {}" ,
e.getMessage());
        return Collections.emptyList();
    }
}
```

Creating a Cassandra headless service

The role of the headless service is to allow clients in the Kubernetes cluster to connect to the Cassandra cluster through a standard Kubernetes service instead of keeping track of the network identities of the nodes or putting a dedicated load balancer in front of all the nodes. Kubernetes provides all that out of the box through its services.

Here is the configuration file:

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

The `app: Cassandra` label will group all the pods to participate in the service. Kubernetes will create endpoint records and the DNS will return a record for discovery. `clusterIP` is set to `None`, which means the service is headless and Kubernetes will not do any load balancing or proxying. This is important because Cassandra nodes do their own communication directly.

The 9042 port is used by Cassandra to serve CQL requests. Those can be queries, inserts/updates (it's always an upsert with Cassandra), or deletes.

Using StatefulSets to create the Cassandra cluster

Declaring a StatefulSet is not trivial. It is arguably the most complex Kubernetes resource. It has a lot of moving parts: standard metadata, the StatefulSet spec, the pod template (which is often pretty complex itself), and volume claim templates.

Dissecting the StatefulSet YAML file

Let's go methodically over this example StatefulSet YAML file that declares a three-node Cassandra cluster.

Here is the basic metadata. Note the `apiVersion` string is `apps/v1` (StatefulSets became generally available in Kubernetes 1.9):

```
apiVersion: "apps/v1"
kind: StatefulSet
metadata:
  name: Cassandra
  labels:
    app: cassandra
```

The StatefulSet spec defines the headless service name, the label selector (`app: cassandra`), how many pods there are in the StatefulSet, and the pod template (explained later). The `replicas` field specifies how many pods are in the StatefulSet:

```
spec:
  serviceName: Cassandra
  replicas: 3
  selector:
    matchLabels:
      app: Cassandra
  template:
    ...
```

The term `replicas` for the pods is an unfortunate choice because the pods are not replicas of each other. They share the same pod template, but they have a unique identity and they are responsible for different subsets of the state in general. This is even more confusing in the case of Cassandra, which uses the same term, `replicas`, to refer to groups of nodes that redundantly duplicate some subset of the state (but are not identical, because each can manage additional state too). I opened a GitHub issue with the Kubernetes project to change the term from `replicas` to `members`:

<https://github.com/kubernetes/kubernetes.github.io/issues/2103>

The pod template contains a single container based on the custom Cassandra image. It also sets the termination grace period to 30 minutes. This means that when Kubernetes needs to terminate the pod, it will send the containers a `SIGTERM` signal notifying them they should exit and giving them a chance to do it gracefully. Any container that is still running after the grace period will be killed via `SIGKILL`.

Here is the pod template with the `app: cassandra` label:

```
template:
  metadata:
    labels:
```

```

    app: Cassandra
spec:
  terminationGracePeriodSeconds: 1800
  containers:
  ...

```

The containers section has multiple important parts. It starts with a name and the image we looked at earlier:

```

containers:
- name: Cassandra
  image: gcr.io/google-samples/cassandra:v14
  imagePullPolicy: Always

```

Then, it defines multiple container ports needed for external and internal communication by Cassandra nodes:

```

ports:
- containerPort: 7000
  name: intra-node
- containerPort: 7001
  name: tls-intra-node
- containerPort: 7199
  name: jmx
- containerPort: 9042
  name: cql

```

The resources section specifies the CPU and memory needed by the container. This is critical because the storage management layer should never be a performance bottleneck due to CPU or memory. Note that it follows the best practice of identical requests and limits to ensure the resources are always available once allocated:

```

resources:
  limits:
    cpu: "500m"
    memory: 1Gi
  requests:
    cpu: "500m"
    memory: 1Gi

```

Cassandra needs access to **Inter Process Communication (IPC)**, which the container requests through the security context's capabilities:

```

securityContext:

```

```
capabilities:
  add:
    - IPC_LOCK
```

The `lifecycle` section runs the Cassandra `nodetool drain` command to make sure data on the node is transferred to other nodes in the Cassandra cluster when the container needs to shut down. This is the reason a 30-minute grace period is needed. Node draining involves moving a lot of data around:

```
lifecycle:
  preStop:
    exec:
      command:
        - /bin/sh
        - -c
        - nodetool drain
```

The `env` section specifies environment variables that will be available inside the container. The following is a partial list of the necessary variables. The `CASSANDRA_SEEDS` variable is set to the headless service so a Cassandra node can talk to seed nodes on startup and discover the whole cluster. Note that in this configuration we don't use the special Kubernetes seed provider. `POD_IP` is interesting because it utilizes the Downward API to populate its value via the field reference to `status.podIP`:

```
env:
  - name: MAX_HEAP_SIZE
    value: 512M
  - name: HEAP_NEWSIZE
    value: 100M
  - name: CASSANDRA_SEEDS
    value: "cassandra-0.cassandra.default.svc.cluster.local"
  - name: CASSANDRA_CLUSTER_NAME
    value: "K8Demo"
  - name: CASSANDRA_DC
    value: "DC1-K8Demo"
  - name: CASSANDRA_RACK
    value: "Rack1-K8Demo"
  - name: CASSANDRA_SEED_PROVIDER
    value: io.k8s.cassandra.KubernetesSeedProvider
  - name: POD_IP
    valueFrom:
      fieldRef:
        fieldPath: status.podIP
```

The readiness probe makes sure that requests are not sent to the node until it is actually ready to service them. The `ready-probe.sh` script utilizes Cassandra's `nodetool status` command:

```
readinessProbe:
  exec:
    command:
      - /bin/bash
      - -c
      - /ready-probe.sh
  initialDelaySeconds: 15
  timeoutSeconds: 5
```

The last part of the container spec is the volume mount, which must match a persistent volume claim:

```
volumeMounts:
- name: cassandra-data
  mountPath: /var/lib/cassandra
```

That's it for the container spec. The last part is the volume claim templates. In this case, dynamic provisioning is used. It's highly recommended to use SSD drives for Cassandra storage, and especially its journal. The requested storage in this example is 1 GiB. I discovered through experimentation that 1-2 TB is ideal for a single Cassandra node. The reason is that Cassandra does a lot of data shuffling under the covers, compacting and rebalancing the data. If a node leaves the cluster or a new one joins the cluster, you have to wait until the data is properly rebalanced before the data from the node that left is properly re-distributed or a new node is populated. Note that Cassandra needs a lot of disk space to do all this shuffling. It is recommended to have 50% free disk space. When you consider that you also need replication (typically 3x), then the required storage space can be 6x your data size. You can get by with 30% free space if you're adventurous and maybe use just 2x replication depending on your use case. But don't get below 10% free disk space, even on a single node. I learned the hard way that Cassandra will simply get stuck and will be unable to compact and rebalance such nodes without extreme measures.

A storage class called `fast` must be defined in this case. Usually, for Cassandra, you need a special storage class and can't use the Kubernetes cluster default storage class.

The access mode is, of course, `ReadWriteOnce`:

```
volumeClaimTemplates:
- metadata:
    name: cassandra-data
```

```
spec:
  storageClassName: fast
  accessModes: [ "ReadWriteOnce" ]
  resources:
    requests:
      storage: 1Gi
```

When deploying a StatefulSet, Kubernetes creates the pod in order according to its index number. When scaling up or down, it also does it in order. For Cassandra, this is not important because it can handle nodes joining or leaving the cluster in any order. When a Cassandra pod is destroyed (ungracefully), the persistent volume remains. If a pod with the same index is created later, the original persistent volume will be mounted into it. This stable connection between a particular pod and its storage enables Cassandra to manage the state properly.

Summary

In this chapter, we covered the topic of stateful applications and how to integrate them with Kubernetes. We discovered that stateful applications are complicated and considered several mechanisms for discovery, such as DNS and environment variables. We also discussed several state management solutions, such as in-memory redundant storage and persistent storage. The bulk of the chapter revolved around deploying a Cassandra cluster inside a Kubernetes cluster using a StatefulSet. We drilled down into the low-level details in order to appreciate what it really takes to integrate a third-party complex distributed system such as Cassandra into Kubernetes. At this point, you should have a thorough understanding of stateful applications and how to apply them in your Kubernetes-based system. You are armed with multiple methods for various use cases, and maybe you've even learned a little bit about Cassandra.

In the next chapter, we will continue our journey and explore the important topic of scalability, in particular auto-scalability, and how to deploy and do live upgrades and updates as the cluster dynamically grows. These issues are very intricate, especially when the cluster has stateful apps running on it.

8

Deploying and Updating Applications

In this chapter, we will explore the automated pod scalability that Kubernetes provides, how it affects rolling updates, and how it interacts with quotas. We will touch on the important topic of provisioning and how to choose and manage the size of the cluster. Finally, we will go over how the Kubernetes team improved the performance of Kubernetes and how they test the limits of Kubernetes with the Kubemark tool. Here are the main points we will cover:

- Horizontal pod autoscaling
- Performing rolling updates with autoscaling
- Handling scarce resources with quotas and limits
- Pushing the envelope with Kubernetes performance

At the end of this chapter, you will have the ability to plan a large-scale cluster, provision it economically, and make informed decisions about the various trade-offs between performance, cost, and availability. You will also understand how to set up horizontal pod autoscaling and use resource quotas intelligently to let Kubernetes automatically handle intermittent fluctuations in volume as well as deploy software safely to your cluster.

Horizontal pod autoscaling

Kubernetes can watch over your pods and scale them when the CPU utilization or some other metric crosses a threshold. The autoscaling resource specifies the details (percentage of CPU, how often to check) and the corresponding autoscale controller adjusts the number of replicas, if needed.

The following diagram illustrates the different players and their relationships:

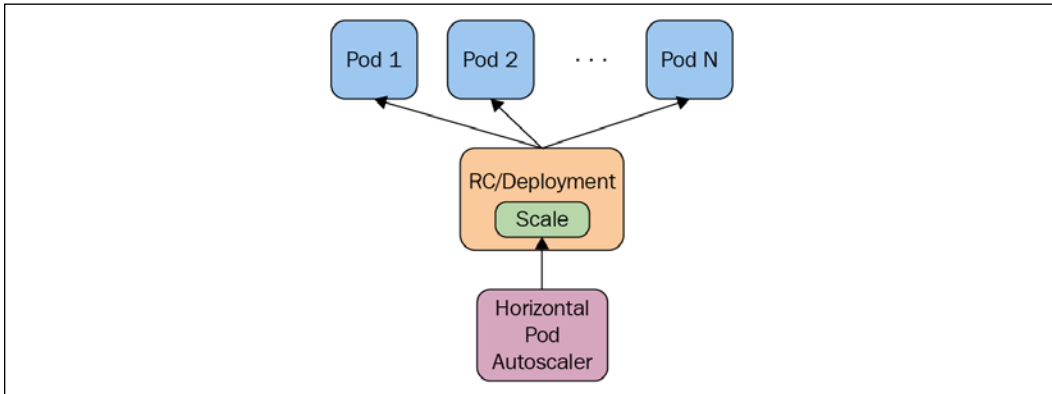


Figure 8.1: HPA interacting with pods

As you can see, the **horizontal pod autoscaler (HPA)** doesn't create or destroy pods directly. It relies instead on the replication controller or deployment resources. This is very smart because you don't need to deal with situations where autoscaling conflicts with the replication controller or deployments trying to scale the number of pods, unaware of the autoscaler efforts.

The autoscaler automatically does what we had to do ourselves before. Without the autoscaler, if we had a replication controller with replicas set to 3, but we determined that based on average CPU utilization we actually needed 4, then we would update the replication controller from 3 to 4 and keep monitoring the CPU utilization manually in all pods. The autoscaler will do it for us.

Declaring an HPA

To declare an HPA, we need a replication controller, or a deployment, and an autoscaling resource. Here is a simple deployment configured to maintain 3 Nginx pods:

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 400m
        ports:
        - containerPort: 80
```

Note that in order to participate in autoscaling, the containers must request a specific amount of CPU.

The HPA references the Nginx deployment in `scaleTargetRef`:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx
spec:
  maxReplicas: 4
  minReplicas: 2
  targetCPUUtilizationPercentage: 90
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: nginx
```

The `minReplicas` and `maxReplicas` values specify the range of scaling. This is needed to avoid runaway situations that could occur because of some problem. Imagine that, due to some bug, every pod immediately uses 100% CPU regardless of the actual load. Without the `maxReplicas` limit, Kubernetes will keep creating more and more pods until all cluster resources are exhausted.

If we are running in a cloud environment with autoscaling of VMs, then we will incur a significant cost. The other side of this problem is that, if there is no `minReplicas` and there is a lull in activity, then all pods could be terminated, and when new requests come in all the pods will have to be created and scheduled again. If there are patterns of on and off activity, then this cycle can repeat multiple times. Keeping the minimum of replicas running can smooth this phenomenon. In the preceding example, `minReplicas` is set to 2 and `maxReplicas` is set to 4. Kubernetes will ensure that there are always between 2 to 4 Nginx instances running.

The target CPU utilization percentage is a mouthful. Let's abbreviate it to TCUP. You specify a single number like 80%, but Kubernetes doesn't start scaling up and down immediately when the threshold is crossed. This could lead to constant thrashing if the average load hovers around the TCUP. Kubernetes will alternate frequently between adding more replicas and removing replicas. A new scale-up algorithm was added in Kubernetes 1.12 that can handle automatically scaling up your cluster. Scaling down is left to the cluster administrator, who can configure how long the autoscaler will wait before scaling down a pod. The mechanism is a special flag to the controller-manager called `--horizontal-pod-autoscaler-downscale-stabilization`. It determines the minimum wait between consecutive downscale operations. The default value is five minutes.

Let's check the HPA:

```
$ kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
nginx	Deployment/nginx	<unknown>/90%	2	4	0	4s

As you can see, the targets are unknown. The HPA requires a metrics server to measure the CPU percentage. One of the easiest ways to install the metrics server is by using Helm. We have installed Helm in *Chapter 3, High Availability and Reliability* already. Here is the command to install the Kubernetes metrics server into the monitoring namespace:

```
$ helm install metrics-server bitnami/metrics-server \
    --version 4.2.1 \
    --namespace monitoring
```

After redeploying nginx and the HPA, you can see the utilization and that the replica count is 3, which is within the range of 2-4:

```
$ kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
nginx	Deployment/nginx	0%/90%	2	4	3	109s

Custom metrics

CPU utilization is an important metric to gauge if pods that are bombarded with too many requests should be scaled up, or if they are mostly idle and can be scaled down. But CPU is not the only and sometimes not even the best metric to keep track of. Memory may be the limiting factor, or even more specialized metrics, such as the depth of a pod's internal on-disk queue, the average latency on a request, or the average number of service timeouts.

The horizontal pod custom metrics were added as an alpha extension in version 1.2. In version 1.6 they were upgraded to beta status. You can now autoscale your pods based on multiple custom metrics. The autoscaler will evaluate all the metrics and will autoscale based on the largest number of replicas required, so the requirements of all the metrics are respected.

Using the HPA with custom metrics requires some configuration when launching your cluster. First, you need to enable the API aggregation layer. Then you need to register your resource metrics API and your custom metrics API. Heapster provides an implementation of the resource metrics API you can use. Just start Heapster with the `--api-server` flag set to true, but note that Heapster is deprecated as of Kubernetes 1.11. You need to run a separate server that exposes the custom metrics API. A good starting point is <https://github.com/kubernetes-incubator/custom-metrics-apiserver>.

The next step is to start kube-controller-manager with the following flags:

- `--horizontal-pod-autoscaler-use-rest-clients=true`
- `--kubeconfig` or `--master`

The `--master` flag will override `--kubeconfig` if both are specified. These flags specify the location of the API aggregation layer, allowing the controller manager to communicate to the API server.

In Kubernetes 1.7, the standard aggregation layer that Kubernetes provides runs in-process with the kube-apiserver, so the target IP address can be found with:

```
$ kubectl get pods --selector k8s-app=kube-apiserver -n kube-system -o jsonpath='{.items[0].status.podIP}'
```

Autoscaling with Kubectl

Kubectl can create an autoscale resource using the standard `create` command and a configuration file. But Kubectl also has a special command, `autoscale`, that lets you easily set an autoscaler in one command without a special configuration file.

First, let's start a deployment that makes sure there are three replicas of a simple pod that just runs an infinite bash loop:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: bash-loop
spec:
  replicas: 3
  selector:
    matchLabels:
      name: bash-loop
  template:
    metadata:
      labels:
        name: bash-loop
    spec:
      containers:
        - name: bash-loop
          image: glg1/py-kube:0.2
          resources:
            requests:
              cpu: 100m
          command: ["/bin/bash", "-c", "while true; do sleep 10; done"]
```

```
$ kubectl create -f bash-loop-deployment.yaml
deployment.apps/bash-loop created
```

Here is the resulting deployment:

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
bash-loop	3/3	3	3	61m

You can see that the desired and current count are both three, meaning three pods are running. Let's make sure:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
bash-loop-6746f7f75f-2w8ft	1/1	Running	0	62m
bash-loop-6746f7f75f-b2nks	1/1	Running	1	62m
bash-loop-6746f7f75f-g9j8t	1/1	Running	0	62m

Now, let's create an autoscaler. To make it interesting, we'll set the minimum number of replicas to 4 and the maximum number to 6:

```
$ kubectl autoscale deployment bash-loop --min=4 --max=6 --cpu-percent=50
horizontalpodautoscaler.autoscaling/bash-loop autoscaled
```

Here is the resulting HPA (you can use `hpa`). It shows the referenced deployment, the target and current CPU percentage, and the min/max pods. The name matches the referenced deployment, `bash-loop`:

```
$ kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
bash-loop	Deployment/bash-loop	0%/50%	4	6	4

58s

Originally, the deployment was set to have three replicas, but the autoscaler has a minimum of four pods. What's the effect on the deployment? Now the desired number of replicas is four. If the average CPU utilization goes above 50%, then it may climb to five or even six:

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
bash-loop	4/4	4	4	65m

Just to make sure everything works, here is another look at the pods. Note the new pod (2 minutes and 23 seconds old) that was created because of the autoscaling:

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
bash-loop-6746f7f75f-2w8ft	1/1	Running	0	66m
bash-loop-6746f7f75f-b2nks	1/1	Running	1	66m
bash-loop-6746f7f75f-g9j8t	1/1	Running	0	66m
bash-loop-6746f7f75f-mvv74	1/1	Running	0	2m23s

When we delete the HPA, the deployment retains the last desired number of replicas (four in this case). Nobody remembers that deployment was created with three replicas:

```
$ kubectl delete hpa bash-loop
horizontalpodautoscaler.autoscaling "bash-loop" deleted
```

As you can see, the deployment wasn't reset and still maintains four pods even when the autoscaler is gone:

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
bash-loop	4/4	4	4	68m

Let's try something else. What happens if we create a new HPA with a range of 2 to 6 and the same CPU target of 50%?

```
$ kubectl autoscale deployment bash-loop --min=2 --max=6 --cpu-percent=50
horizontalpodautoscaler.autoscaling/bash-loop autoscaled
```

Well, the deployment still maintains its four replicas, which is within the range:

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
bash-loop	4/4	4	4	73m

However, the actual CPU utilization is zero, or close to zero. The replica count should have been scaled down to two replicas, but because the HPA doesn't scale down immediately we have to wait a few minutes:

```
$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
bash-loop	2/2	2	2	78m

Let's check out the HPA itself:

```
$ kubectl get hpa
```

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS
bash-loop	Deployment/bash-loop	0%/50%	2	6	2
AGE					
8m43s					

Performing rolling updates with autoscaling

Rolling updates are the cornerstone of managing large clusters. Kubernetes supports rolling updates at the replication controller level and by using deployments. Rolling updates using replication controllers are incompatible with the HPA. The reason is that during a rolling deployment, a new replication controller is created and the HPA remains bound to the old replication controller. Unfortunately, the intuitive `kubectl rolling-update` command triggers a replication controller rolling update.

Since rolling updates are such an important capability, I recommend that you always bind HPAs to a deployment object instead of a replication controller or a replica set. When the HPA is bound to a deployment, it can set the replicas in the deployment spec and let the deployment take care of the necessary underlying rolling update and replication.

Here is a deployment configuration file we've used for deploying the hue-reminders service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hue-reminders
spec:
  replicas: 2
  selector:
    matchLabels:
      app: hue
      service: reminders
  template:
    metadata:
      name: hue-reminders
      labels:
        app: hue
        service: reminders
    spec:
      containers:
        - name: hue-reminders
          image: glg1/hue-reminders:2.2
          resources:
            requests:
              cpu: 100m
          ports:
            - containerPort: 80
```

To support it with autoscaling and ensure we always have between 10 to 15 instances running, we can create an autoscaler configuration file:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: hue-reminders
spec:
```



```
maxReplicas: 15
minReplicas: 10
targetCPUUtilizationPercentage: 90
scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: hue-reminders
```

Alternatively, we can use the `kubectl autoscale` command:

```
$ kubectl autoscale deployment hue-reminders --min=10 --max=15 --cpu-
percent=90
```

Let's perform a rolling update from version 2.2 to 3.0:

```
$ kubectl set image deployment/hue-reminders hue-reminders=g1g1/hue-
reminders:3.0 --record
```

We can check the status using `rollout status`:

```
$ kubectl rollout status deployment hue-reminders
```

```
Waiting for deployment "hue-reminders" rollout to finish: 7 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 7 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 7 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 8 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 8 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 8 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 8 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 9 out of 10 new
replicas have been updated...
Waiting for deployment "hue-reminders" rollout to finish: 4 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 3 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 3 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 3 old replicas are
```

```

pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 2 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 2 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 2 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 1 old replicas are
pending termination...
Waiting for deployment "hue-reminders" rollout to finish: 8 of 10 updated
replicas are available...
Waiting for deployment "hue-reminders" rollout to finish: 9 of 10 updated
replicas are available...
deployment "hue-reminders" successfully rolled out

```

Finally, we review the history of the deployment:

```

$ kubectl rollout history deployment hue-reminders
deployment.extensions/hue-reminders
REVISION  CHANGE-CAUSE
1          <none>
2          kubectl set image deployment/hue-reminders hue-reminders=g1g1/
hue-reminders:3.0 --record=true

```

Handling scarce resources with limits and quotas

With the HPA creating pods on the fly, we need to think about managing our resources. Scheduling can easily get out of control, and inefficient use of resources is a real concern. There are several factors that can interact with each other in subtle ways:

- Overall cluster capacity
- Resource granularity per node
- Division of workloads per namespace
- DaemonSets
- StatefulSets
- Affinity, anti-affinity, taints, and tolerations

First, let's understand the core issue. The Kubernetes scheduler has to take into account all these factors when it schedules pods. If there are conflicts or a lot of overlapping requirements, then Kubernetes may have a problem finding room to schedule new pods. For example, a very extreme yet simple scenario is that a DaemonSet runs on every node a pod that requires 50% of the available memory. Now, Kubernetes can't schedule any pod that needs more than 50% memory because the DaemonSet pod gets priority. Even if you provision new nodes, the DaemonSet will immediately commandeer half of the memory.

StatefulSets are similar to DaemonSets in that they require new nodes to expand. The trigger to adding new members to the stateful set is growth in data, but the impact is taking resources from the pool available for Kubernetes to schedule other members. In a multi-tenant situation, the noisy neighbor problem can rear its head in a provisioning or resource allocation context. You may plan exact rations meticulously in your namespace between different pods and their resource requirements, but you share the actual nodes with your neighbors from other namespaces that you may not even have visibility into.

Most of these problems can be mitigated by judiciously using namespace resource quotas and careful management of the cluster capacity across multiple resource types such as CPU, memory, and storage.

But, in most situations, a more robust and dynamic approach is to take advantage of the cluster autoscaler, which can add capacity to the cluster when needed.

Enabling resource quotas

Most Kubernetes distributions support ResourceQuota out of the box. The API server's `--admission-control` flag must have ResourceQuota as one of its arguments. You will also have to create a ResourceQuota object to enforce it. Note that there may be at most one ResourceQuota object per namespace to prevent potential conflicts. This is enforced by Kubernetes.

Resource quota types

There are different types of quota we can manage and control. The categories are compute, storage, and objects.

Compute resource quota

Compute resources are CPU and memory. For each one, you can specify a limit or request a certain amount. Here is the list of compute-related fields. Note that `requests.cpu` can be specified as just `cpu`, and `requests.memory` can be specified as just `memory`:

- `limits.cpu`: Across all pods in a non-terminal state, the sum of CPU limits cannot exceed this value
- `limits.memory`: Across all pods in a non-terminal state, the sum of memory limits cannot exceed this value
- `requests.cpu`: Across all pods in a non-terminal state, the sum of CPU requests cannot exceed this value
- `requests.memory`: Across all pods in a non-terminal state, the sum of memory requests cannot exceed this value

Since Kubernetes 1.10 you can also specify a quota for extended resources such as GPU resources. Here is an example:

```
requests.nvidia.com/gpu: 10
```

Storage resource quota

The storage resource quota type is a little more complicated. There are two entities you can restrict per namespace: the amount of storage and the number of persistent volume claims. However, in addition to just globally setting the quota on the total storage or the total number of persistent volume claims, you can also do that per storage class. The notation for storage class resource quota is a little verbose, but it gets the job done:

- `requests.storage`: The total amount of requested storage across all persistent volume claims
- `persistentvolumeclaims`: The maximum number of persistent volume claims allowed in the namespace
- `.storageclass.storage.k8s.io/requests.storage`: The total amount of requested storage across all persistent volume claims associated with the storage class name
- `.storageclass.storage.k8s.io/persistentvolumeclaims`: The maximum number of persistent volume claims allowed in the namespace that are associated with the storage class name

Kubernetes 1.8 added alpha support for ephemeral storage quotas too:

- `requests.ephemeral-storage`: The total amount of requested ephemeral storage across all pods in the namespace claims
- `limits.ephemeral-storage`: The total amount of limits for ephemeral storage across all pods in the namespace claims

Object count quota

Kubernetes has another category of resource quotas, which is API objects. My guess is that the goal is to protect the Kubernetes API server from having to manage too many objects. Remember that Kubernetes does a lot of work under the hood. It often has to query multiple objects to authenticate, authorize, and ensure that an operation doesn't violate any of the many policies that may be in place. A simple example is pod scheduling based on replication controllers. Imagine that you have 1,000,000,000 replication controller objects. Maybe you just have three pods and most of the replication controllers have zero replicas. Still, Kubernetes will spend all its time just verifying that indeed all those billion replication controllers have no replicas of their pod templates and that they don't need to kill any pods. This is an extreme example, but the concept applies. Too many API objects means a lot of work for Kubernetes.

Since Kubernetes 1.9 you can restrict the number of any namespaced resource (prior to that coverage of objects that can be restricted was a little spotty). The syntax is interesting: `count/<resource type>.<group>`. Typically, in the YAML files and `kubectl` you identify objects by group first as in `<group>/<resource type>`.

Here are some objects you may want to limit (note that deployments can be limited for two separate API groups):

```
count/configmaps
count/deployments.apps
count/deployments.extensions
count/persistentvolumeclaims
count/replicasets.apps
count/replicationcontrollers
count/secrets
count/services
count/statefulsets.apps
count/jobs.batch
count/cronjobs.batch
```

Since Kubernetes 1.5 you can restrict the number of custom resources too. Note that while the custom resource definition is cluster-wide this allows you to restrict the actual number of the custom resources in each namespace. For example:

```
count/awesome.custom.resource
```

The most glaring omission is namespaces. There is no limit to the number of namespaces. Since all limits are per namespace, you can easily overwhelm Kubernetes by creating too many namespaces, where each namespace has only a small number of API objects.

But, the ability to create namespaces, which don't need resource quotas to constrain them, should be reserved for the cluster administrator only.

Quota scopes

Some resources, such as pods, may be in different states, and it is useful to have different quotas for these different states. For example, if there are many pods that are terminating (this happens a lot during rolling updates) then it is OK to create more pods even if the total number exceeds the quota. This can be achieved by only applying a pod object count quota to non-terminating pods. Here are the existing scopes:

- **Terminating:** Match pods where `activeDeadlineSeconds >= 0`
- **NotTerminating:** Match pods where `activeDeadlineSeconds` is nil
- **BestEffort:** Match pods that have best effort quality of service
- **NotBestEffort:** Match pods that do not have best effort quality of service

While the **BestEffort** scope applies only to pods, the **Terminating**, **NotTerminating**, and **NotBestEffort** scopes apply to CPU and memory too. This is interesting because a resource quota limit can prevent a pod from terminating. Here are the supported objects:

- `cpu`
- `memory`
- `limits.cpu`
- `limits.memory`
- `requests.cpu`
- `requests.memory`
- `pods`

Resource quotas and priority classes

Kubernetes 1.9 introduced priority classes as a way to prioritize scheduling pods when resources are scarce. In Kubernetes 1.14 priority classes became stable. However, as of Kubernetes 1.12 resource quotas support separate resource quotas per priority class (in beta). That means that with priority classes you can sculpt your resource quotas in a very fine-grained manner even within a namespace.

Requests and limits

The meaning of requests and limits in the context of resource quotas is that it requires the containers to explicitly specify the target attribute. This way, Kubernetes can manage the total quota because it knows exactly what range of resources is allocated to each container.

Working with quotas

That was a lot of theory. It's time to get hands on. Let's create a namespace first:

```
$ kubectl create namespace ns
namespace/ns created
```

Using namespace-specific context

When working with namespaces other than the default, I prefer to use a context, so I don't have to keep typing `--namespace=ns` for every command:

```
$ kubectl config set-context ns --namespace=ns --user=default
--cluster=default
Context "ns" created.
```

```
$ kubectl config use-context ns
Switched to context "ns".
```

Creating quotas

Here is a quota for compute:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-quota
spec:
  hard:
    pods: 2
    requests.cpu: 1
    requests.memory: 20Mi
    limits.cpu: 2
    limits.memory: 2Gi
```

We create it by typing:

```
$ kubectl apply -f compute-quota.yaml
resourcequota/compute-quota created
```

And here is a count quota:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: object-counts-quota
spec:
  hard:
    count/configmaps: 10
    count/persistentvolumeclaims: 4
    count/jobs.batch: 20
    count/secrets: 3
```

We create it by typing:

```
$ kubectl apply -f object-count-quota.yaml
resourcequota/object-counts-quota created
```

We can observe all the quotas:

```
$ kubectl get quota
NAME                CREATED AT
compute-quota       2020-06-08T16:44:28Z
object-counts-quota 2020-06-08T18:14:01Z
```

We can drill down to get all the information by using describe for both resource quotas:

```
$ kubectl describe quota compute-quota
Name:                compute-quota
Namespace:           ns
Resource             Used   Hard
-----
limits.cpu           0    2
limits.memory        0    2Gi
pods                 0    2
requests.cpu         0    1
requests.memory      0    20Mi
```



```
$ kubectl describe quota object-counts-quota
Name:                object-counts-quota
Namespace:           ns
Resource             Used  Hard
-----
count/configmaps     0    10
count/jobs.batch      0    20
count/persistentvolumeclaims 0    4
count/secrets        1    3
```

As you can see, it reflects exactly the specification and it is defined in the ns namespace.

This view gives us an instant understanding of global resource usage of important resources across the cluster without diving into too many separate objects.

Let's add an Nginx server to our namespace:

```
$ kubectl create -f nginx-deployment.yaml
deployment.apps/nginx created
```

Let's check the pods:

```
$ kubectl get pods
No resources found.
```

Uh-oh. No resources found. But, there was no error when the deployment was created. Let's check out the deployment then:

```
$ kubectl describe deployment nginx
Name:                nginx
Namespace:           ns
CreationTimestamp:   Mon, 8 Jun 2020 21:13:02 -0700
Labels:              <none>
Annotations:         deployment.kubernetes.io/revision: 1
Selector:            run=nginx
Replicas:            3 desired | 0 updated | 0 total | 0 available | 3
unavailable
StrategyType:        RollingUpdate
MinReadySeconds:     0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels:  run=nginx
  Containers:
    nginx:
```

```

Image:      nginx
Port:       80/TCP
Host Port:  0/TCP
Requests:
  cpu:      400m
Environment: <none>
Mounts:     <none>
Volumes:    <none>
Conditions:
  Type          Status Reason
  ----          -
  Progressing   True   NewReplicaSetCreated
  Available     False  MinimumReplicasUnavailable
  ReplicaFailure True   FailedCreate
OldReplicaSets: <none>
NewReplicaSet:  nginx-5759dd6b5c (0/3 replicas created)
Events:
  Type          Reason          Age    From          Message
  ----          -
  Normal        ScalingReplicaSet 72s    deployment-controller Scaled up replica
set nginx-5759dd6b5c to 3

```

There it is, in the conditions section – the `ReplicaFailure` status is `True` and the reason is `FailedCreate`. You can see that the deployment created a new replica set called `5759dd6b5c`, but it couldn't create the pods it was supposed to create. We still don't know why. Let's check out the `ReplicaSet` object. I use the JSON output format (`-o json`) and pipe it to `jq` for its nice layout, which is much better than the `jsonpath` output format that `kubect1` supports natively:

```

$ kubect1 get rs nginx-5759dd6b5c -o json | jq .status.conditions
[
  {
    "lastTransitionTime": "2020-06-08T04:13:02Z",
    "message": "pods \"nginx-5759dd6b5c-9wj7k7\" is forbidden: failed quota:
compute-quota: must specify limits.cpu,limits.memory,requests.memory",
    "reason": "FailedCreate",
    "status": "True",
    "type": "ReplicaFailure"
  }
]

```

The message is crystal clear. Since there is a compute quota in the namespace, every container must specify its CPU, memory requests, and limit. The quota controller must account for all container compute resources usage to ensure the total namespace quota is respected.

OK. We understand the problem, but how to resolve it? We can create a dedicated deployment object for each pod type we want to use and carefully set the CPU and memory requests and limit.

For example, we can define nginx deployment with resources. Since the resource quota specifies a hard limit of 2 pods, let's reduce the number of replicas from 3 to 2 as well:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
  selector:
    matchLabels:
      run: nginx
  template:
    metadata:
      labels:
        run: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          requests:
            cpu: 400m
            memory: 6Mi
          limits:
            cpu: 400m
            memory: 6Mi
        ports:
        - containerPort: 80
```

Let's create it and check the pods:

```
$ kubectl create -f nginx-deployment-with-resources.yaml
```

```
deployment.apps/nginx created
```

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-c6db6d7d-zpz96	1/1	Running	0	36s
nginx-c6db6d7d-dztkr	1/1	Running	0	36s

Yeah, it works! However, specifying the limit and resources for each pod type can be exhausting. Is there an easier or better way?

Using limit ranges for default compute quotas

A better way is to specify default compute limits. Enter limit ranges. Here is a configuration file that sets some defaults for containers:

```
apiVersion: v1
kind: LimitRange
metadata:
  name: limits
spec:
  limits:
  - default:
      cpu: 400m
      memory: 5Mi
    defaultRequest:
      cpu: 400m
      memory: 5Mi
    type: Container
```

Let's create it and observe the default limits:

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

```
limitrange 'limits' created
```

```
$ kubectl describe limits
```

```
Name:          limits
Namespace:     ns
Type           Resource  Min  Max  Default Request  Default Limit  Max Limit/
Request Ratio
-----
Container      cpu          -    -    100m             200m           -
Container      memory       -    -    5Mi              6Mi            -
```

To test it, let's delete our current nginx deployment with the explicit limits and deploy again our original nginx:

```
$ kubectl delete deployment nginx
deployment.extensions "nginx" deleted

$ kubectl create -f nginx-deployment.yaml
deployment.apps/nginx created
```

```
$ kubectl get deployment
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
nginx     2/3     2            2           26s
```

As you can see, only 2 out of 3 pods are ready. What happened? The default limits worked, but if you recall, the compute quota had a hard limit of 2 pods for the namespace. There is no way to override it with the `RangeLimit` object, so the deployment was able to create only two nginx pods.

Choosing and managing the cluster capacity

With Kubernetes' horizontal pod autoscaling, DaemonSets, StatefulSets, and quotas, we can scale and control our pods, storage, and other objects. However, in the end, we're limited by the physical (virtual) resources available to our Kubernetes cluster. If all your nodes are running at 100% capacity, you need to add more nodes to your cluster. There is no way around it. Kubernetes will just fail to scale. On the other hand, if you have very dynamic workloads then Kubernetes can scale down your pods, but if you don't scale down your nodes correspondingly you will still pay for the excess capacity. In the cloud you can stop and start instances on demand. Combining it with the cluster autoscaler can solve the compute capacity problem automatically. That's the theory. In practice there are always nuances.

Choosing your node types

The simplest solution is to choose a single node type with a known quantity of CPU, memory, and local storage. But that is typically not the most efficient and cost-effective solution. It makes capacity planning simple because the only question is how many nodes are needed. Whenever you add a node, you add a known quantity of CPU and memory to your cluster, but most Kubernetes clusters and components within the cluster handle different workloads. We may have a stream processing pipeline where many pods receive some data and process it in one place.

This workload is CPU-heavy and may or may not need a lot of memory. Other components, such as a distributed memory cache, need a lot of memory, but very little CPU. Other components, such as a Cassandra cluster, need multiple SSD disks attached to each node.

For each type of node you should consider proper labeling and making sure that Kubernetes schedules the pods that are designed to run on that node type.

Choosing your storage solutions

Storage is a huge factor in scaling a cluster. There are three categories of scalable storage solution:

- Roll your own
- Use your cloud platform storage solution
- Use an out-of-cluster solution

When you use roll your own, you install some type of storage solution in your Kubernetes cluster. The benefits are flexibility and full control, but you have to manage and scale it yourself.

When you use your cloud platform storage solution, you get a lot out of the box, but you lose control, you typically pay more, and depending on the service you may be locked in to that provider.

When you use an out-of-cluster solution, the performance and cost of data transfer may be much greater. You typically use this option if you need to integrate with an existing system.

Of course, large clusters may have multiple data stores from all categories. This is one of the most critical decisions you have to make, and your storage needs may change and evolve over time.

Trading off cost and response time

If money is not an issue you can just over-provision your cluster. Every node will have the best hardware configuration available, you'll have way more nodes than are needed to process your workloads, and you'll have copious amounts of available storage. But guess what? Money is always an issue!

You may get by with over-provisioning when you're just starting and your cluster doesn't handle a lot of traffic. You may just run five nodes, even if two nodes are enough most of the time. Multiply everything by 1,000 and someone will come asking questions if you have thousands of idle machines and petabytes of empty storage.

OK. So, you measure and optimize carefully and you get 99.99999% utilization of every resource. Congratulations, you just created a system that can't handle an iota of extra load or the failure of a single node without dropping requests on the floor or delaying responses.

You need to find the middle ground. Understand the typical fluctuations of your workloads and consider the cost/benefit ratio of having excess capacity versus having reduced response time or processing ability.

Sometimes, if you have strict availability and reliability requirements, you can build redundancy into the system and then you over-provision by design. For example, you want to be able to hot swap a failed component with no downtime and no noticeable effects. Maybe you can't lose even a single transaction. In this case, you'll have a live backup for all critical components, and that extra capacity can be used to mitigate temporary fluctuations without any special actions.

Using multiple node configurations effectively

Effective capacity planning requires you to understand the usage patterns of your system and the load each component can handle. That may include a lot of data streams generated inside the system. When you have a solid understanding of the typical workloads, you can look at workflows and which components handle which parts of the load. Then you can compute the number of pods and their resource requirements. In my experience, there are some relatively fixed workloads, some workloads that vary predictably (such as office hours versus non-office hours), and then you have your completely crazy workloads that behave erratically. You have to plan according for each workload, and you can design several families of node configurations that can be used to schedule pods that match a particular workload.

Benefiting from elastic cloud resources

Most cloud providers let you scale instances automatically, which is a perfect complement to Kubernetes' horizontal pod autoscaling. If you use cloud storage, it also grows magically without you having to do anything. However, there are some gotchas that you need to be aware of.

Autoscaling instances

All the big cloud providers have instance autoscaling in place. There are some differences, but scaling up and down based on CPU utilization is always available, and sometimes custom metrics are available too. Sometimes, load balancing is offered as well. As you can see, there is some overlap with Kubernetes here.

If your cloud provider doesn't have adequate autoscaling with proper control, it is relatively easy to roll your own, where you monitor your cluster resource usage and invoke cloud APIs to add or remove instances. You can extract the metrics from Kubernetes.

Here is a diagram that shows how two new instances are added based on a CPU load monitor:

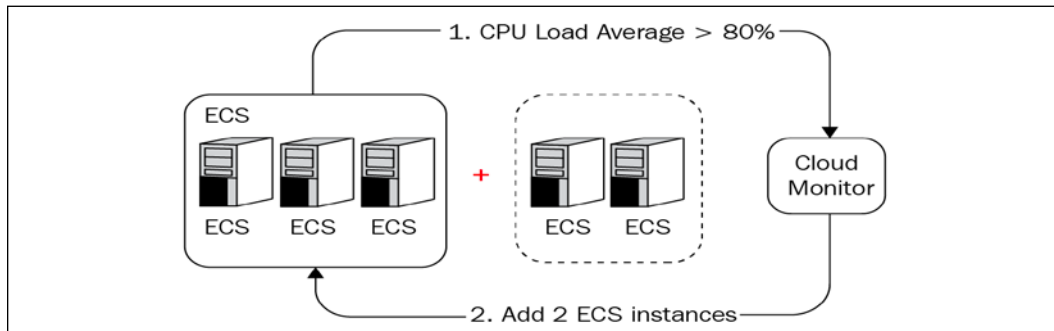


Figure 8.2: Adding load instances

Mind your cloud quotas

When working with cloud providers, some of the most annoying things are quotas. I've worked with four different cloud providers (AWS, GCP, Azure, and Alibaba cloud) and I was always bitten by quotas at some point. The quotas exist to let the cloud providers do their own capacity planning (and also to protect you from inadvertently starting 1,000,000 instances that you won't be able to pay for), but from your point of view it is yet one more thing that can trip you up. Imagine that you set up a beautiful autoscaling system that works like magic, and suddenly the system doesn't scale when you hit 100 nodes. You quickly discover that you are limited to 100 nodes and you open a support request to increase the quota. However, a human must approve quota requests, and that can take a day or two. In the meantime, your system is unable to handle the load.

Manage regions carefully

Cloud platforms are organized in regions and availability zones. Some services and machine configurations are available only in some regions. Cloud quotas are also managed at the regional level. Performance and cost of data transfers within regions is much lower (often free) than across regions. When planning your cluster, you should consider carefully your geo-distribution strategy. If you need to run your cluster across multiple regions, you may have some tough decisions to make regarding redundancy, availability, performance, and cost.

Considering container-native solutions

A container-native solution is when your cloud provider offers a way to deploy containers directly into their infrastructure. You don't need to provision instances and then install a container runtime (like the Docker daemon) and only then deploy your containers. Instead, you just provide your containers and the platform is responsible for finding a machine to run your container. You are totally separated from the actual machines your containers are running on.

All the major cloud providers now provide solutions that abstract instances completely:

- AWS Fargate
- **Azure Container Instances (ACI)**
- Google Cloud Run

These solutions are not Kubernetes-specific, but they can work great with Kubernetes. The cloud providers already provide managed Kubernetes control plane with Google's **Google Kubernetes Engine (GKE)**, Microsoft's **Azure Kubernetes Service (AKS)**, and Amazon Web Services' **Elastic Kubernetes Service (EKS)**. But managing the data plane (the nodes) was left to the cluster administrator.

The container-native solution allows the cloud provider to do that on your behalf. Google Run for GKE and AKS with ACI already provide it. AWS EKS will support Fargate in the near future.

For example, in AKS you can provision virtual nodes. A virtual node is not backed up by an actual VM. Instead it utilizes ACI to deploy containers when necessary. You pay for it only when it the cluster needs to scale beyond the capacity of the regular nodes. It is faster to scale then using the cluster autoscaler that needs to provision an actual VM-backed node.

The following diagram illustrates this burst to the ACI approach:

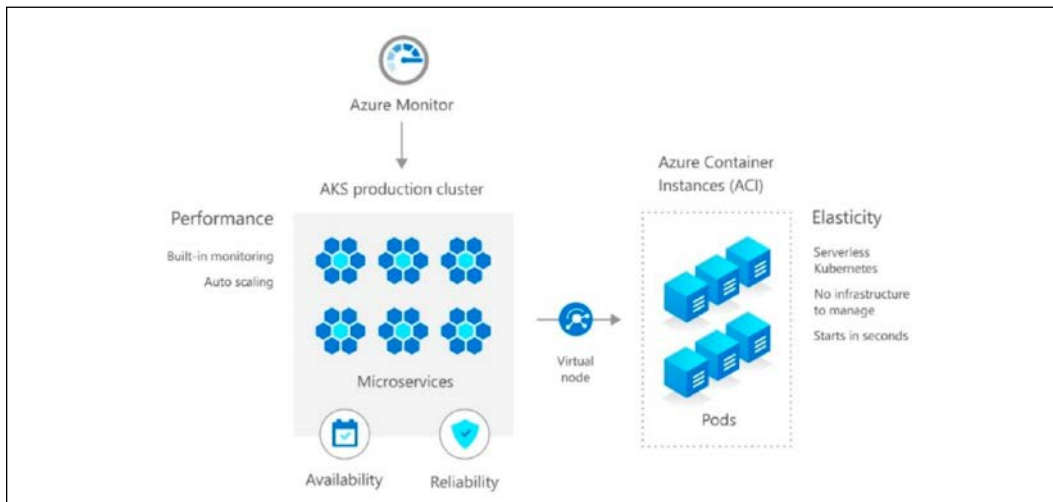


Figure 8.3: Virtual node scaling

Pushing the envelope with Kubernetes

In this section, we will see how the Kubernetes team pushes Kubernetes to its limit. The numbers are quite telling, but some of the tools and techniques, such as Kubemark, are ingenious, and you may even use them to test your clusters. In the wild, there are some Kubernetes clusters with 3,000 - 5,000 nodes. At CERN, the OpenStack team achieved 2 million requests per second:

<http://superuser.openstack.org/articles/scaling-magnum-and-kubernetes-2-million-requests-per-second/>

Mirantis conducted a performance and scaling test in their scaling lab where they deployed 5,000 Kubernetes nodes (in VMs) on 500 physical servers.

OpenAI scaled their machine learning Kubernetes cluster to 2,500 nodes and learned some valuable lessons such as minding the query load of logging agents and storing events in a separate etcd cluster:

<https://blog.openai.com/scaling-kubernetes-to-2500-nodes/>

There are many more interesting use cases here:

<https://www.cncf.io/projects/case-studies/>

By the end of this section you'll appreciate the effort and creativeness that goes into improving Kubernetes on a large scale, you will know how far you can push a single Kubernetes cluster and what performance to expect, and you'll get an inside look at some tools and techniques that can help you evaluate the performance of your own Kubernetes clusters.

Improving the performance and scalability of Kubernetes

The Kubernetes team focused heavily on performance and scalability in Kubernetes 1.6. When Kubernetes 1.2 was released, it supported clusters of up to 1,000 nodes within the Kubernetes service-level objectives. Kubernetes 1.3 doubled the number to 2,000 nodes, and Kubernetes 1.6 brought it to a staggering 5,000 nodes per cluster. We will get into the numbers later, but first let's look under the hood and see how Kubernetes achieved these impressive improvements.

Caching reads in the API server

Kubernetes keeps the state of the system in etcd, which is very reliable, though not superfast (although etcd 3 delivered massive improvement specifically to enable larger Kubernetes clusters). The various Kubernetes components operate on snapshots of that state and don't rely on real-time updates. That fact allows the trading of some latency for throughput. All the snapshots used to be updated by etcd watches. Now, the API server has an in-memory read cache that is used for updating state snapshots. The in-memory read cache is updated by etcd watches. These schemes significantly reduce the load on etcd and increase the overall throughput of the API server.

The pod lifecycle event generator

Increasing the number of nodes in a cluster is key for horizontal scalability, but pod density is crucial too. Pod density is the number of pods that the Kubelet can manage efficiently on one node.

If pod density is low, then you can't run too many pods on one node. That means that you might not benefit from more powerful nodes (more CPU and memory per node) because the Kubelet will not be able to manage more pods. The other alternative is to force the developers to compromise their design and create coarse-grained pods that do more work per pod. Ideally, Kubernetes should not force your hand when it comes to pod granularity. The Kubernetes team understands this very well and invested a lot of work in improving pod density.

In Kubernetes 1.1, the official (tested and advertised) number was 30 pods per node. I actually ran 40 pods per node on Kubernetes 1.1, but I paid for it in excessive Kubelet overhead that stole CPU from the worker pods. In Kubernetes 1.2, the number jumped to 100 pods per node.

The Kubelet used to poll the container runtime constantly for each pod in its own goroutine. That put a lot of pressure on the container runtime that during peaks to performance has reliability issues, in particular CPU utilization. The solution was the **Pod Lifecycle Event Generator (PLEG)**. The way the PLEG works is that it lists the state of all the pods and containers and compares it to the previous state. This is done once for all the pods and containers. Then, by comparing the state to the previous state, the PLEG knows which pods need to sync again and invokes only those pods. That change resulted in a significant four-times-lower CPU usage by the Kubelet and the container runtime. It also reduced the polling period, which improves responsiveness.

The following diagram shows the CPU utilization for 120 pods on Kubernetes 1.1 versus Kubernetes 1.2. You can see the 4X factor very clearly:

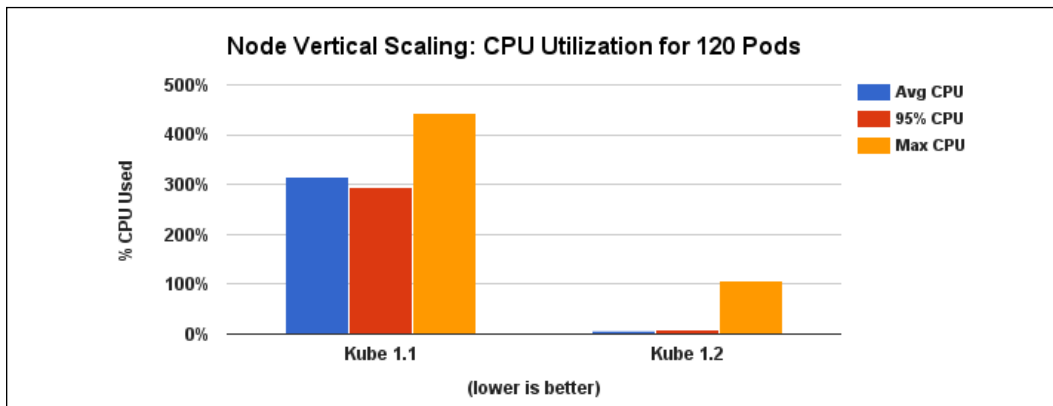


Figure 8.4: CPU utilization for 120 pods with Kube 1.1 and Kube 1.2

Serializing API objects with protocol buffers

The API server has a REST API. REST APIs typically use JSON as their serialization format, and the Kubernetes API server was no different. However, JSON serialization implies marshaling and unmarshaling JSON to native data structures. This is an expensive operation. In a large-scale Kubernetes cluster, a lot of components need to query or update the API server frequently. The cost of all that JSON parsing and composition adds up quickly. In Kubernetes 1.3, the Kubernetes team added an efficient protocol buffers serialization format. The JSON format is still there, but all internal communication between Kubernetes components uses the protocol buffers serialization format.

etcd3

Kubernetes switched from etcd2 to etcd3 in Kubernetes 1.6. This was a big deal. Scaling Kubernetes to 5,000 nodes wasn't possible due to limitations of etcd2, especially related to the watch implementation. The scalability needs of Kubernetes drove many of the improvements of etcd3, as CoreOS used Kubernetes as a measuring stick. Some of the big ticket items are talked about here.

GRPC instead of REST

etcd2 has a REST API, etcd3 has a gRPC API (and a REST API via gRPC gateway). The http/2 protocol at the base of gRPC can use a single TCP connections for multiple streams of requests and responses.

Leases instead of TTLs

etcd2 uses **Time to Live (TTL)** per key as the mechanism to expire keys, while etcd3 uses leases with TTLs where multiple keys can share the same key. This significantly reduces keep-alive traffic.

Watch implementation

The watch implementation of etcd3 takes advantage of gRPC bi-directional streams and maintain a single TCP connection to send multiple events, which reduced the memory footprint by at least an order of magnitude.

State storage

With etcd3 Kubernetes started storing all the state as protocol buffers, which eliminated a lot of wasteful JSON serialization overhead.

Other optimizations

The Kubernetes team made many other optimizations such as:

- Optimizing the scheduler (which resulted in 5-10x higher scheduling throughput)
- Switching all controllers to a new recommended design using shared informers, which reduced resource consumption of controller-manager
- Optimizing individual operations in the API server (conversions, deep copies, patch)
- Reducing memory allocation in the API server (which significantly impacts the latency of API calls)

Measuring the performance and scalability of Kubernetes

In order to improve performance and scalability, you need a sound idea of what you want to improve and how you're going to measure the improvements. You must also make sure that you don't violate basic properties and guarantees in the quest for improved performance and scalability. What I love about performance improvements is that they often buy you scalability improvements for free. For example, if a pod needs 50% of the CPU of a node to do its job and you improve performance so that the pod can do the same work using 33% CPU, then you can suddenly run three pods instead of two on that node, and you've improved the scalability of your cluster by 50% overall (or reduced your cost by 33%).

The Kubernetes SLOs

Kubernetes has **Service Level Objectives (SLOs)**. Those guarantees must be respected when trying to improve performance and scalability. Kubernetes has a one-second response time for API calls. That's 1,000 milliseconds. It actually achieves an order of magnitude faster response times most of the time.

Measuring API responsiveness

The API has many different endpoints. There is no simple API responsiveness number. Each call has to be measured separately. In addition, due to the complexity and the distributed nature of the system, not to mention networking issues, there can be a lot of volatility to the results. A solid methodology is to break the API measurements into separate endpoints and then run a lot of tests over time and look at percentiles (which is standard practice).

It's also important to use enough hardware to manage a large number of objects. The Kubernetes team used a 32-core VM with 120 GB for the master in this test.

The following diagram describes the 50th, 90th, and 99th percentile of various important API call latencies for Kubernetes 1.3. You can see that the 90th percentile is very low, below 20 milliseconds. Even the 99th percentile is less than 125 milliseconds for the DELETE pods operation, and less than 100 milliseconds for all other operations:

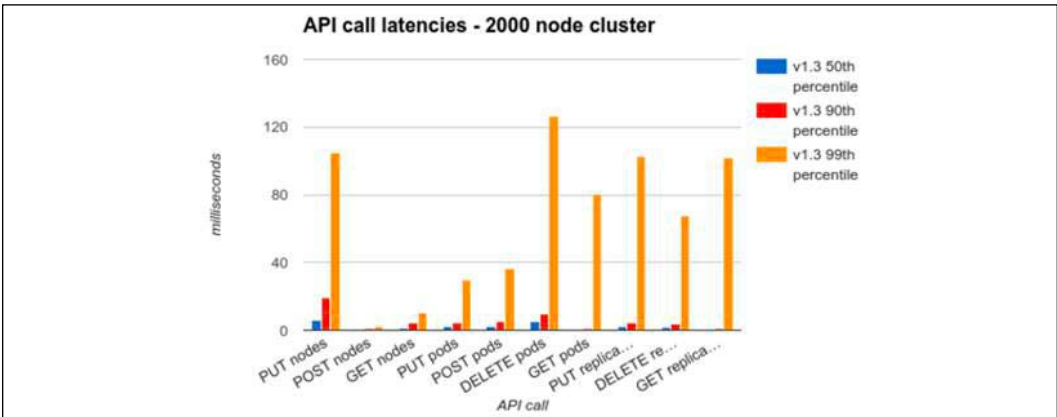


Figure 8.5: API call latencies

Another category of API calls is LIST operations. Those calls are more expansive because they need to collect a lot of information in a large cluster, compose the response, and send a potential large response. This is where performance improvements such as the in-memory read cache and the protocol buffers serialization really shine. The response time is understandably greater than the single API calls, but it is still way below the SLO of one second (1,000 milliseconds):

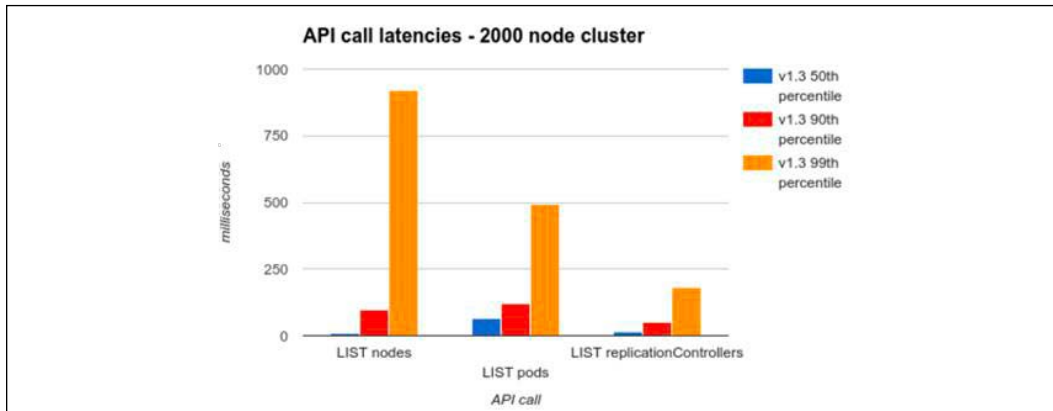


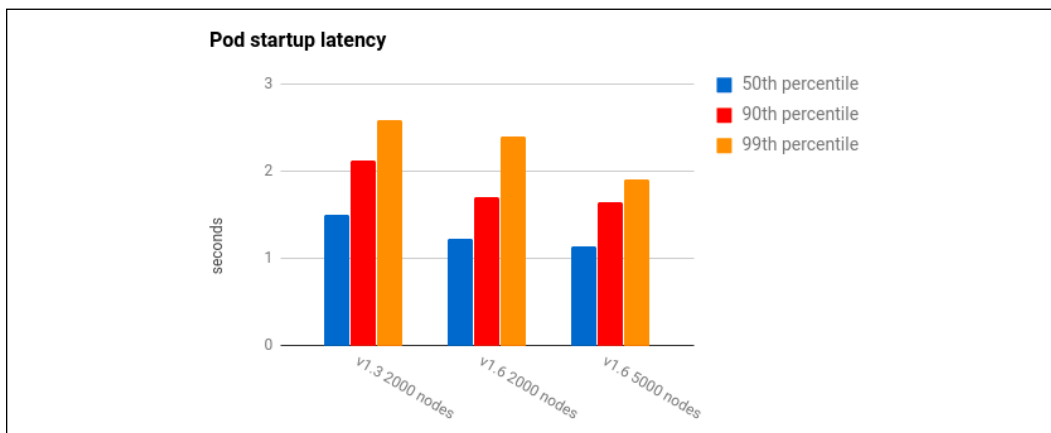
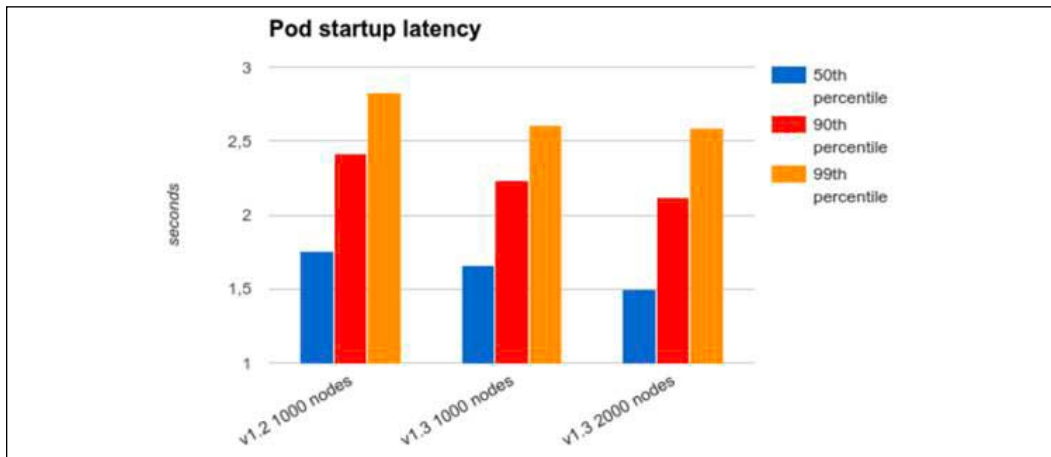
Figure 8.6: API LIST call latencies

Measuring end-to-end pod startup time

One of the most important performance characteristics of a large dynamic cluster is end-to-end pod startup time. Kubernetes creates, destroys, and shuffles pods around all the time. You could say that the primary function of Kubernetes is to schedule pods.

In the following diagram, you can see that pod startup time is less volatile than API calls. This makes sense since there is a lot of work that needs to be done, such as launching a new instance of a runtime, that doesn't depend on cluster size. With Kubernetes 1.2 on a 1,000-node cluster, the 99th percentile end-to-end time to launch a pod was less than 3 seconds. With Kubernetes 1.3, the 99th percentile end-to-end time to launch a pod was a little over 2.5 seconds.

It's remarkable that the time is very close, but a little better with Kubernetes 1.3 on a 2,000-node cluster versus a 1,000-node cluster:



Figures 8.7 and 8.8: Pod startup latencies

Testing Kubernetes at scale

Clusters with thousands of nodes are expensive. Even a project such as Kubernetes that enjoys the support of Google and other industry giants still needs to come up with reasonable ways to test without breaking the bank.

The Kubernetes team runs a full-fledged test on a real cluster at least once per release to collect real-world performance and scalability data. However, there is also a need for a lightweight and cheaper way to experiment with potential improvements and to detect regressions. Enter the Kubemark.

Introducing the Kubemark tool

The Kubemark is a Kubernetes cluster that runs mock nodes called hollow nodes used for running lightweight benchmarks against large-scale (hollow) clusters. Some of the Kubernetes components that are available on a real node such as the Kubelet are replaced with a hollow Kubelet. The hollow Kubelet fakes a lot of the functionality of a real Kubelet. A hollow Kubelet doesn't actually start any containers, and it doesn't mount any volumes. But from the Kubernetes cluster point of view – the state stored in etcd – all those objects exist and you can query the API server. The hollow Kubelet is actually the real Kubelet with an injected mock Docker client that doesn't do anything.

Another important hollow component is the hollow proxy, which mocks the Kubeproxy component. It again uses the real Kubeproxy code with a mock proxier interface that does nothing and avoids touching iptables.

Setting up a Kubemark cluster

A Kubemark cluster uses the power of Kubernetes. To set up a Kubemark cluster, perform the following steps:

1. Create a regular Kubernetes cluster where we can run N hollow nodes.
2. Create a dedicated VM to start all master components for the Kubemark cluster.
3. Schedule N hollow node pods on the base Kubernetes cluster. Those hollow nodes are configured to talk to the Kubemark API server running on the dedicated VM.
4. Create add-on pods by scheduling them on the base cluster and configuring them to talk to the Kubemark API server.

A full-fledged guide is available here:

<https://github.com/kubernetes/community/blob/master/contributors/devel/sig-scalability/kubemark-guide.md>

Comparing a Kubemark cluster to a real-world cluster

The performance of Kubemark clusters is mostly similar to the performance of real clusters. For the pod startup end-to-end latency, the difference is negligible. For the API-responsiveness, the differences are greater, though generally less than a factor of two. However, trends are exactly the same: an improvement/regression on a real cluster is visible as a similar percentage drop/increase in metrics on Kubemark.

Summary

In this chapter, we've covered many topics relating to scaling Kubernetes clusters. We discussed how the HPA can automatically manage the number of running pods based on CPU utilization or other metrics, how to perform rolling updates correctly and safely in the context of autoscaling, and how to handle scarce resources via resource quotas. Then we moved on to overall capacity planning and management of the cluster's physical or virtual resources. Finally, we delved into the ins and outs of performance benchmarking on Kubernetes.

At this point, you have a good understanding of all the factors that come into play when a Kubernetes cluster is facing dynamic and growing workloads. You have multiple tools to choose from for planning and designing your own scaling strategy.

In the next chapter, we will learn how to package applications for deployment on Kubernetes. We will discuss Helm as well as Kustomize and other solutions.

9

Packaging Applications

In this chapter, we are going to look at Helm, the Kubernetes package manager. Every successful and non-trivial platform must have a good packaging system. Helm was developed by Deis (acquired by Microsoft in April 2017) and later contributed to the Kubernetes project directly. It became a CNCF project in 2018. We will start by understanding the motivation for Helm, its architecture, and its components. Then, we'll get hands-on and demonstrate how to use Helm and its charts within Kubernetes. That includes finding, installing, customizing, deleting, and managing charts. Last but not least, we'll cover how to create your own charts and handle versioning, dependencies, and templating.

The topics we will cover are as follows:

- Understanding Helm
- Using Helm
- Creating your own charts

Understanding Helm

Kubernetes provides many ways to organize and orchestrate your containers at runtime, but it lacks a higher-level organization for grouping sets of images together. This is where Helm comes in. In this section, we'll go over the motivation for Helm, its architecture and components, and discuss what has changed in the transition from Helm 2 to Helm 3.

The motivation for Helm

Helm provides support for several important use cases:

- Managing complexity
- Easy upgrades
- Simple sharing
- Safe rollbacks

Charts can describe even the most complex apps, provide repeatable application installation, and serve as a single point of authority. In-place upgrades and custom hooks allow easy updates. It's simple to share charts that can be versioned and hosted on public or private servers. When you need to roll back recent upgrades, Helm provides a single command that allows you to roll back a cohesive set of changes to your infrastructure.

The Helm 2 architecture

Helm is designed to perform the following:

- Create new charts from scratch
- Package charts into chart archive (TGZ) files
- Interact with chart repositories where charts are stored
- Install and uninstall charts into an existing Kubernetes cluster
- Manage the release cycle of charts that have been installed with Helm

Helm uses a client-server architecture to achieve these goals.

Helm 2 components

Helm has a server component that runs on your Kubernetes cluster and a client component that you can run on a local machine.

The Tiller server

This server is responsible for managing releases. It interacts with the Helm clients as well as the Kubernetes API server. Its main functions are as follows:

- Listening for incoming requests from the Helm client
- Combining a chart and configuration to build a release

- Installing charts into Kubernetes
- Tracking the subsequent release
- Upgrading and uninstalling charts by interacting with Kubernetes

The Helm client

You install the Helm client on your machine. It is responsible for the following:

- Local chart development
- Managing repositories
- Interacting with the Tiller server
- Sending charts to be installed
- Asking for information about releases
- Requesting upgrades or the uninstallation of existing releases

Helm 3

Helm 2 is great and plays a very important role in the Kubernetes ecosystem. However, there was a lot of criticism about Tiller – its server-side component. Helm 2 was designed and implemented before RBAC became the official access control method. In the interest of usability, Tiller is installed by default with a very open set of permissions. It wasn't easy to lock it down for production usage. This is especially challenging in multi-tenant clusters.

The Helm team listened to the criticisms and came up with the Helm 3 design. Instead of the Tiller in-cluster component, Helm 3 utilizes the Kubernetes API server itself via CRDs to manage the state of releases. The bottom line is that Helm 3 is a client-only program. It can still manage releases and perform the same tasks as Helm 2, but without needing to install a server-side component.

This approach is more Kubernetes-native, is less complicated, and the security concerns are gone. Helm users can perform via Helm only as much as their Kube config allows.

Using Helm

Helm is a rich package management system that lets you perform all the necessary steps to manage the applications installed on your cluster. Let's roll up our sleeves and get going. We'll look at installing both Helm 2 and Helm 3, but we will use Helm 3 for all of our hands-on experiments and demonstrations.

Installing Helm

Installing Helm involves installing the client and the server. Helm is implemented in Go. The Helm 2 executable can serve as either the client or the server. Helm 3, as mentioned before, is a client-only program.

Installing the Helm client

You must have Kubectl configured properly to talk to your Kubernetes cluster because the Helm client uses the Kubectl configuration to talk to the Helm server (Tiller).

Helm provides binary releases for all platforms here:

<https://github.com/helm/helm/releases>

For Windows, the chocolatey package manager is the best option (and is usually up to date):

```
choco install kubernetes-helm
```

For macOS and Linux, you can install the client from a script:

```
$ curl https://raw.githubusercontent.com/helm/helm/master/scripts/get >  
get_helm.sh  
$ chmod 700 get_helm.sh  
$ ./get_helm.sh
```

On macOS, you can also use Homebrew:

```
$ brew install kubernetes-helm
```

```
$ helm version
```

```
version.BuildInfo{Version:"v3.0.0", GitCommit:"e29ce2a54e96cd02ccfce88bee4f58bb6e2a28b6", GitTreeState:"clean", GoVersion:"go1.13.4"}
```

Installing the Tiller server for Helm 2

If you run Helm 2 for some reason, then you need to install Tiller – the server-side component – which is not necessary for Helm 3. Tiller typically runs inside your cluster. For development, it is sometimes easier to run Tiller locally.

Installing Tiller in-cluster

The easiest way to install Tiller is from a machine where the Helm 2 client is installed. Run the following command: `helm init`.

This will initialize both the client as well as the Tiller server on the remote Kubernetes cluster. When the installation is complete, you will have a running Tiller pod in the `kube-system` namespace of your cluster:

```
$ kubectl get po --namespace=kube-system -l name=tiller
```

NAME	READY	STATUS	RESTARTS	AGE
tiller-deploy-3210613906-2j5sh	1/1	Running	0	1m

You can also run `helm version` to check both the client's version and the server's version:

```
$ helm version
```

```
Client: &version.Version{SemVer:"2.16.8", GitCommit:"1402a4d6ec9fb349e17b912e32fe259ca21181e3", GitTreeState:"clean"}
```

```
Server: &version.Version{SemVer:"2.16.8", GitCommit:"1402a4d6ec9fb349e17b912e32fe259ca21181e3", GitTreeState:"clean"}
```

Finding charts

In order to install useful applications and software with Helm, you need to find their charts first. Helm was designed to work with multiple repositories of charts. Helm 2 was configured to search the stable repository by default, but you could add additional repositories. Helm 3 comes with no default, but you can search Helm Hub (<https://hub.helm.sh/>) or specific repositories. Helm Hub was launched in December 2018, and it was designed to make it easier for you to discover charts and repositories hosted outside the stable or incubator repositories.

This is where the `helm search` command comes in. Helm can search the Helm Hub or a specific repository.

The hub contains 1,300 charts at the moment:

```
$ helm search hub | wc -l
1300
```


We can search the hub for a specific keyword like mariadb:

```
$ helm search hub mariadb
```

URL	CHART	VERSION	APP
VERSION DESCRIPTION			
https://hub.helm.sh/charts/ibm-charts/ibm-galer... Galera Cluster is a multi-master solution for M...	1.1.0		
https://hub.helm.sh/charts/ibm-charts/ibm-maria... MariaDB is developed as open source software an...	1.1.2		
https://hub.helm.sh/charts/bitnami/mariadb Fast, reliable, scalable, and easy to use open-...	7.5.1		10.3.23
https://hub.helm.sh/charts/bitnami/phpmyadmin phpMyAdmin is an mysql administration frontend	6.2.0		5.0.2
https://hub.helm.sh/charts/bitnami/mariadb-cluster Chart to create a Highly available MariaDB cluster	1.0.1		10.2.14
https://hub.helm.sh/charts/bitnami/mariadb-galera MariaDB Galera is a multi-master database clust...	3.1.3		10.4.13

As you can see, there are several charts that match the keyword MariaDB. You can investigate them further and find the best one for your use case.

Adding repositories

By default, Helm 3 comes with no repositories set up, so you can only search the hub. Let's add the bitnami repository, so we can limit our search to that repository only:

```
$ helm repo add bitnami https://charts.bitnami.com/bitnami
"bitnami" has been added to your repositories
```

Now, we can search the bitnami repo:

```
$ helm search repo mariadb
```

NAME	CHART	VERSION	APP	VERSION	DESCRIPTION
bitnami/mariadb		7.5.1	10.3.23		Fast, reliable, scalable, and easy to use open-...
bitnami/mariadb-cluster		1.0.1	10.2.14		Chart to create a Highly available MariaDB cluster
bitnami/mariadb-galera		3.1.3	10.4.13		MariaDB Galera is a multi-master database clust...
stable/mariadb		7.3.14	10.3.22		DEPRECATED Fast, reliable, scalable, and easy t...
bitnami/phpmyadmin		6.2.0	5.0.2		phpMyAdmin is an mysql administration frontend

stable/phpmyadmin	4.3.5	5.0.1	DEPRECATED phpMyAdmin
-------------------	-------	-------	-----------------------

is an mysql administratio...

The results are a subset of the results returned from the hub.

The official repository has a rich library of charts that represent all of the modern open source databases, monitoring systems, Kubernetes-specific helpers, and a slew of other offerings, such as a Minecraft server. Searching for Helm charts is a good way to find interesting projects and tools. I often search for the kube keyword:

```
$ helm search repo kube
```

NAME	CHART VERSION	APP VERSION	DESCRIPTION
bitnami/kube-state-metrics	0.3.2	1.9.7	kube-
state-metrics is a simple service that lis...			
bitnami/kubeapps	3.7.1	v1.10.1	Kubeapps is a
dashboard for your Kubernetes clu...			
bitnami/kubewatch	1.0.14	0.0.4	Kubewatch
notifies your slack rooms when change...			
kubefed-charts/kubefed	0.3.0		KubeFed helm cha
rt			
kubefed-charts/federation-v2	0.0.10		Kubernetes
Federation V2 helm chart			
bitnami/external-dns	3.2.0	0.7.2	ExternalDNS is a
Kubernetes addon that configur...			
bitnami/metallb	0.1.14	0.9.3	The
Metal LB for Kubernetes			
bitnami/metrics-server	4.2.0	0.3.7	Metrics
Server is a cluster-wide aggregator of ...			
bitnami/prometheus-operator	0.20.0	0.39.0	The
Prometheus Operator for Kubernetes provides...			

To get more information about a specific chart, we can use the show command (you can use the inspectalias command too). Let's look at bitnami/mariadb:

```
$ helm show chart bitnami/mariadb
Error: failed to download "bitnami/mariadb" (hint: running 'helm repo
update' may help)
```

Ha-ha. Helm requires that the repositories are up to date. Let's update our repositories:

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
```

...Successfully got an update from the "bitnami" chart repository
Update Complete. * Happy Helming!*

Now, it works:

```
$ helm show chart bitnami/mariadb
apiVersion: v1
appVersion: 10.3.22
description: Fast, reliable, scalable, and easy to use open-source
relational database
  system. MariaDB Server is intended for mission-critical, heavy-load
production systems
  as well as for embedding into mass-deployed software. Highly available
MariaDB cluster.
home: https://mariadb.org
icon: https://bitnami.com/assets/stacks/mariadb/img/mariadb-stack-220x234.
png
keywords:
- mariadb
- mysql
- database
- sql
- prometheus
maintainers:
- email: containers@bitnami.com
  name: Bitnami
name: mariadb
sources:
- https://github.com/bitnami/bitnami-docker-mariadb
- https://github.com/prometheus/mysqld_exporter
version: 7.5.1
```

You can also ask Helm to show you the README file, the values, or all of the information associated with a chart. This can be overwhelming at times.

Installing packages

OK. You've found the package of your dreams. Now, you probably want to install it on your Kubernetes cluster. When you install a package, Helm creates a release that you can use to keep track of the installation progress. Let's install MariaDB using the `helm install` command. Let's go over the output in detail.

The first part of the output lists the name of the release that we provided mariadb, when it was deployed, the namespace, and the revision:

```
$ helm install mariadb bitnami/mariadb
```

```
NAME: mariadb
LAST DEPLOYED: Mon Jun 8 12:26:34 2020
NAMESPACE: ns
STATUS: deployed
REVISION: 1
```

The next part includes custom notes, which can be pretty wordy. There is a lot of useful information here about verifying, getting credentials, connecting to the database, and upgrading the chart if necessary:

NOTES:

Please be patient while the chart is being deployed

Tip:

Watch the deployment status using the command: `kubectl get pods -w --namespace default -l release=mariadb`

Services:

```
echo Master: mariadb.ns.svc.cluster.local:3306
echo Slave:  mariadb-slave.ns.svc.cluster.local:3306
```

Administrator credentials:

Username: root

Password : `$(kubectl get secret --namespace default mariadb -o jsonpath="{.data.mariadb-root-password}" | base64 --decode)`

To connect to your database:

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

```
kubectl run mariadb-client --rm --tty -i --restart='Never' --image
docker.io/bitnami/mariadb:10.3.18-debian-9-r36 --namespace default
--command - bash
```

2. To connect to master service (read/write):

```
mysql -h mariadb.ns.svc.cluster.local -uroot -p my_database
```

3. To connect to slave service (read-only):

```
mysql -h mariadb-slave.ns.svc.cluster.local -uroot -p my_database
```

To upgrade this helm chart:

1. Obtain the password as described on the 'Administrator credentials'

section and set the 'rootUser.password' parameter as shown below:

```
ROOT_PASSWORD=$(kubectl get secret --namespace default mariadb -o
jsonpath="{.data.mariadb-root-password}" | base64 --decode)
helm upgrade mariadb bitnami/mariadb --set rootUser.password=$ROOT_
PASSWORD
```

Checking the installation status

Helm doesn't wait for the installation to complete because it may take a while. The `helm status` command displays the latest information on a release in the same format as the output of the initial `helm install` command. In the output of the `install` command, you can see that the persistent volume claim had a pending status. Let's check it out now:

```
$ kubectl get pods -w -l release=mariadb
NAME                READY   STATUS    RESTARTS   AGE
mariadb-master-0    0/1     Pending   0           4m21s
mariadb-slave-0     0/1     Pending   0           4m21s
```

Oh, no. The pods are pending. A quick investigation shows that MariaDB declares a persistent volume claim; however, since there is no default storage class in the cluster, there is no way to provide the storage needed:

```
$ kubectl describe pvc data-mariadb-master-0
Name:                data-mariadb-master-0
Namespace:          default
StorageClass:
Status:              Pending
Volume:
Labels:              app=mariadb
                    component=master
                    heritage=Helm
                    release=mariadb
Annotations:         <none>
Finalizers:          [kubernetes.io/pvc-protection]
Capacity:
Access Modes:
VolumeMode:          Filesystem
Events:
  Type            Reason              Age           From
  Message
  ----            -
  -----
```

```

Normal      FailedBinding  3m3s (x42 over 13m)  persistentvolume-
controller  no persistent volumes available for this claim and no storage
class is set
Mounted By:  mariadb-master-0

```

That's OK. We can create a default storage class with a dynamic provisioner. First, let's use Helm to install a dynamic host path provisioner. Refer to <https://github.com/rimusz/hostpath-provisioner> for details. We add a new Helm repo, update our repo list, and then install the proper chart:

```

$ helm repo add rimusz https://charts.rimusz.net
"rimusz" has been added to your repositories

```

```

$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "rimusz" chart repository
...Successfully got an update from the "bitnami" chart repository
Update Complete. * Happy Helming!*

```

```

$ helm upgrade --install hostpath-provisioner --namespace kube-system
rimusz/hostpath-provisioner
Release "hostpath-provisioner" does not exist. Installing it now.
NAME: hostpath-provisioner
LAST DEPLOYED: Mon Jun 8 17:52:56 2020
NAMESPACE: kube-system
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
The Hostpath Provisioner service has now been installed.

```

A storage class named 'hostpath' has now been created and is available to provision dynamic volumes.

You can use this storageclass by creating a 'PersistentVolumeClaim' with the correct storageClassName attribute. For example:

```

---
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-dynamic-volume-claim
spec:

```

```
storageClassName: "hostpath"
accessModes:
  - ReadWriteOnce
resources:
  requests:
    storage: 100Mi
```

Since we don't control the persistent volume claim that the MariaDB chart is creating, we can't specify the new "hostpath" storage class. However, we can make sure it is the default storage class!

```
$ kubectl get sc
```

NAME	PROVISIONER	AGE
hostpath (default)	hostpath	6m26s

If you have another storage class set as the default, you can make it non-default like so:

```
kubectl patch storageclass <your-class-name> -p '{"metadata":
{"annotations":{"storageclass.kubernetes.io/is-default-class":"false"}}}'
```

We have to perform one more non-conventional step. Because we run our tests on k3d + k3s where the nodes are virtual, the host directory of the host path provisioner is actually allocated inside the Docker container that corresponds to the node. For some reason, the permissions for those directories allow only the root to create directories. This can be fixed by running the following command on each of the Docker containers that correspond to the k3s nodes:

```
$ docker exec -it <container name> chmod -R 0777 /mnt/hostpath
```

Now, we can try again. This time everything works. Yay!

Here are the pods, the volumes, the persistent volume claims, and the StatefulSets created by the MariaDB release:

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
mariadb-master-0	1/1	Running	0	24m
mariadb-slave-0	1/1	Running	9	24m

```
$ kubectl get pv
```

NAME	RECLAIM POLICY	STATUS	CLAIM	CAPACITY	ACCESS MODES	STORAGECLASS
REASON	AGE					
pvc-b51aeb37-4a43-4f97-ad52-40e6b6eda4f4				8Gi	RWO	Delete

Bound	default/data-mariadb-master-0	hostpath	30m	
pvc-58c7e42e-a01b-4544-8691-3e56de4676eb	8Gi	RWO		Delete
Bound	default/data-mariadb-slave-0	hostpath	30m	

```
$ kubectl get pvc
```

NAME	CAPACITY	ACCESS MODES	STATUS	VOLUME	STORAGECLASS	AGE
data-mariadb-master-0	8Gi	RWO	Bound	pvc-b51aeb37-4a43-4f97-ad52-40e6b6eda4f4	hostpath	30m
data-mariadb-slave-0	8Gi	RWO	Bound	pvc-58c7e42e-a01b-4544-8691-3e56de4676eb	hostpath	30m

```
$ kubectl get sts
```

NAME	READY	AGE
mariadb-master	1/1	30m
mariadb-slave	1/1	30m

Let's try to connect and verify that MariaDB is indeed accessible. Let's modify the suggested commands from the notes a little bit in order to connect. Instead of running `bash` and then running `mysql`, we can directly run the `mysql` command on the container. First, let's get the root password and copy it to the clipboard (on macOS):

```
$ kubectl get secret -o yaml mariadb -o jsonpath="{.data.mariadb-root-password}" | base64 --decode | pbcopy
```

Then we can connect using `mariadb-client` and paste the password when you see If you don't see a command prompt, try pressing enter.:

```
$ kubectl run --generator=run-pod/v1 mariadb-client --rm -it --image bitnami/mariadb --command -- mysql -h mariadb.default.svc.cluster.local -uroot -p
```

If you don't see a command prompt, try pressing enter.

```
Welcome to the MariaDB monitor.  Commands end with ; or \g.
Your MariaDB connection id is 1364
Server version: 10.3.18-MariaDB-log Source distribution
Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others.
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
```

```
MariaDB [(none)]>
```


Then, we can start playing with our MariaDB database:

```
MariaDB [(none)]> show databases;
```

```
+-----+
| Database          |
+-----+
| information_schema |
| my_database       |
| mysql             |
| performance_schema|
| test              |
+-----+
5 rows in set (0.001 sec)
```

Customizing a chart

Very often, as a user, you will want to customize or configure the charts that you install. Helm fully supports customization via config files. To learn about possible customizations, you can use the `helm show` command again; however, this time, focus on the values. Here is a partial output:

```
$ helm show values bitnami/mariadb
```

```
db:
```

```
  forcePassword: false
  name: my_database
  password: null
  user: null
```

```
image:
```

```
  debug: false
  pullPolicy: IfNotPresent
  registry: docker.io
  repository: bitnami/mariadb
  tag: 10.3.18-debian-9-r36
```

```
master:
```

```
  affinity: {}
  antiAffinity: soft
  config: |-
    [mysqld]
    skip-name-resolve
    explicit_defaults_for_timestamp
    basedir=/opt/bitnami/mariadb
    port=3306
```

```

socket=/opt/bitnami/mariadb/tmp/mysql.sock
tmpdir=/opt/bitnami/mariadb/tmp
max_allowed_packet=16M
bind-address=0.0.0.0
pid-file=/opt/bitnami/mariadb/tmp/mysqld.pid
log-error=/opt/bitnami/mariadb/logs/mysqld.log
character-set-server=UTF8
collation-server=utf8_general_ci

[client]
port=3306
socket=/opt/bitnami/mariadb/tmp/mysql.sock
default-character-set=UTF8
...
rbac:
  create: false
replication:
  enabled: true
  forcePassword: false
  password: null
  user: replicator
rootUser:
  forcePassword: false
  password: null

```

For example, if you want to set a root password and create a database when installing MariaDB, you can create the following YAML file and save it as `mariadb-config.yaml`:

```

mariadbRootPassword: supersecret
mariadbDatabase: awesome_stuff

```

First uninstall the existing mariadb release:

```
$ helm uninstall mariadb
```

Then, run Helm and pass it the YAML file:

```
$ helm install -f mariadb-config.yaml mariadb bitnami/mariadb
```

You can also set individual values on the command line with `--set`. If both `--f` and `--set` try to set the same values, then `--set` takes precedence.

For example, in this case, the root password will be `evenbettersecret`:

```
$ helm install -f mariadb-config.yaml --set mariadbRootPassword=evenbettersecret bitnami/mariadb
```

You can specify multiple values using comma-separated lists: `--set a=1, b=2`.

Additional installation options

The `helm install` command can install from several sources:

- A chart repository (as we've seen)
- A local chart archive (`helm install foo-0.1.1.tgz`)
- An unpacked chart folder (`helm install path/to/foo`)
- A full URL (`helm install https://example.com/charts/foo-1.2.3.tgz`)

Upgrading and rolling back a release

You may want to upgrade a package that you have installed to the latest and greatest version. Helm provides the `upgrade` command, which operates intelligently and only updates things that have changed. For example, let's check the current values of our `mariadb` installation:

```
$ helm get values mariadb
USER-SUPPLIED VALUES:
mariadbDatabase: awesome_stuff
mariadbRootPassword: evenbettersecret
```

Now, let's run, `upgrade`, and change the name of the database:

```
$ helm upgrade mariadb --set mariadbDatabase=awesome_sauce bitnami/mariadb
$ helm get values mariadb
USER-SUPPLIED VALUES:
mariadbDatabase: awesome_sauce
```

Note that we've lost our root password. All of the existing values are replaced when you upgrade. OK, let's roll back. The `helm history` command shows us all of the available revisions we can roll back to:

```
$ helm history mariadb
```

REVISION	UPDATED	STATUS	CHART	APP
VERSION	DESCRIPTION			
1	Mon Jun 8 09:14:10 2020	superseded	mariadb-7.3.14	10.3.22
Install complete				

```

2           Mon Jun 8 09:22:22 2020    superseded mariadb-7.3.14 10.3.22
Upgrade complete
3           Mon Jun 8 09:23:47 2020    superseded mariadb-7.3.14 10.3.22
Upgrade complete
4           Mon Jun 8 09:24:17 2020    deployed   mariadb-7.3.14 10.3.22
Upgrade complete

```

Let's roll back to revision 3:

```

$ helm rollback mariadb 3
Rollback was a success! Happy Helming!

```

```

$ helm history mariadb
REVISION      UPDATED              STATUS      CHART          APP
VERSION DESCRIPTION
1             Mon Jun 8 09:14:10 2020    superseded mariadb-7.3.14 10.3.22
Install complete
2             Mon Jun 8 09:22:22 2020    superseded mariadb-7.3.14 10.3.22
Upgrade complete
3             Mon Jun 8 09:23:47 2020    superseded mariadb-7.3.14 10.3.22
Upgrade complete
4             Mon Jun 8 09:24:17 2019    superseded mariadb-7.3.14 10.3.22
Upgrade complete
5             Mon Jun 8 09:26:04 2019    deployed   mariadb-7.3.14 10.3.22
Rollback to 3

```

As you can see, the rollback created a new revision number 5. Revision 4 is still there in case we want to go back to it.

Let's verify that our changes were rolled back:

```

$ helm get values mariadb
USER-SUPPLIED VALUES:
mariadbDatabase: awesome_stuff
mariadbRootPassword: evenbettersecret

```

Yep. The database name was rolled back to `awesome_stuff` and we got the root password back.

Deleting a release

You can, of course, uninstall a release too using the `helm uninstall` command.

First, let's examine the list of releases. We have only the `mariadb` release:

```

$ helm list

```

NAME	NAMESPACE	REVISION	UPDATED
STATUS	CHART	APP VERSION	
mariadb	default	5	2020-06-08 09:26:04.766743 -0700 PDT
deployed	mariadb-7.3.14	10.3.22	

Now, let's uninstall it:

```
$ helm uninstall mariadb
release "mariadb" uninstalled
```

So, no more releases:

```
$ helm list
```

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHART	APP VERSION
------	-----------	----------	---------	--------	-------	-------------

Helm can keep track of uninstalled releases too. If you provide `--keep-history` when you uninstall, then you'll be able to see any uninstalled releases using the `--all` or `--uninstalled` flags with `helm list`:

```
$ helm list --all
```

NAME	NAMESPACE	REVISION	UPDATED
STATUS	CHART	APP VERSION	
mariadb	default	1	2020-06-08 09:35:47.641033 -0700 PDT
uninstalled	mariadb-7.3.14	10.3.22	

Working with repositories

Helm stores charts in repositories that are simple HTTP servers. Any standard HTTP server can host a Helm repository. In the cloud, the Helm team verifies that both AWS S3 and Google Cloud storage can serve as Helm repositories in web-enabled mode.

Note that Helm doesn't provide tools to upload charts to remote repositories because that would require the remote server to understand Helm, to know where to put the chart, and to know how to update the `index.yaml` file.

On the client side, the `helm repo` command lets you list, add, remove, index, and update:

```
$ helm repo
```

This command consists of multiple subcommands to interact with chart repositories.

It can be used to add, remove, list, and index chart repositories. Example usage:

```
$ helm repo add [NAME] [REPO_URL]
```

Usage:**helm repo [command]****Available Commands:**

add	add a chart repository
index	generate an index file given a directory containing packaged charts
list	list chart repositories
remove	remove a chart repository
update	update information of available charts locally from chart repositories

We've already used the `helm repo add` command earlier. Let's see how to create our own charts and manage them.

Managing charts with Helm

Helm provides several commands to manage charts.

It can create a new chart for you:

```
$ helm create cool-chart
```

Creating cool-chart

Helm will create the following files and directories under cool-chart:

```
$ tree cool-chart
cool-chart
├── Chart.yaml
├── charts
├── templates
│   ├── NOTES.txt
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── ingress.yaml
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml
```

Once you have edited your chart, you can package it into a tar gzipped archive:

```
$ helm package cool-chart
```

Successfully packaged chart and saved it to: cool-chart-0.1.0.tgz

Helm will create an archive called `cool-chart-0.1.0.tgz` and store it in the local directory.

You can also use `helm lint` to help you to find issues with your chart's formatting or information:

```
$ helm lint cool-chart
==> Linting cool-chart
[INFO] Chart.yaml: icon is recommended
1 chart(s) linted, 0 chart(s) failed
```

Taking advantage of starter packs

The `helm create` command takes an optional `--starter` flag that lets you specify a starter chart.

Starters are just regular charts located in `$HELM_HOME/starters`. As a chart developer, you may author charts that are specifically designed to be used as starters. Such charts should be designed with the following considerations in mind:

- The YAML will be overwritten by the generator
- Users will expect to be able to modify such a chart's contents, so the documentation should indicate how users can do this

At the moment, there is no way to install charts; the only way to add a chart to `$HELM_HOME/starters` is to manually copy it there. Make sure to mention that in your chart's documentation if you develop starter pack charts.

Creating your own charts

A chart is a collection of files that describe a related set of Kubernetes resources. A single chart might be used to deploy something simple, such as a Memcached pod, or something complex, such as a full web app stack with HTTP servers, databases, caches, queues, and so on.

Charts are created as files laid out in a particular directory tree. Then, they can be packaged into versioned archives to be deployed. The key file is `Chart.yaml`.

The Chart.yaml file

The `Chart.yaml` file is the main file of a Helm chart. It requires name and version fields:

- `name`: The name of the chart (same as the directory name)
- `version`: A SemVer 2 version

It may also contain various optional fields:

- `kubeVersion`: A SemVer range of compatible Kubernetes versions
- `description`: A single sentence describing this project
- `keywords`: A list of keywords about this project
- `home`: The URL of this project's home page
- `sources`: A list of URLs to source code for this project
- `dependencies`: A list of (name, version, repository) for each dependency (repository is the URL)
- `maintainers`: A list of (name, email, URL) for each maintainer (name is required)
- `icon`: The URL to an SVG or PNG image to be used as an icon
- `appVersion`: The version of the app that this contains
- `deprecated`: Whether this chart is deprecated (Boolean)

Versioning charts

The version field inside of the `Chart.yaml` is used by many Helm tools. When generating a package, the `helm package` command will use the version that it finds in `Chart.yaml` when constructing the package name. The system assumes that the version number in the chart package name matches the version number in `Chart.yaml`. Violating this assumption will cause an error.

The appVersion field

The `appVersion` field is not related to the version field. It is not used by Helm and serves as metadata or a piece of documentation for users who want to understand what they are deploying. Helm ignores it.

Deprecating charts

From time to time, you may want to deprecate a chart. You can mark a chart as deprecated by setting the optional `deprecated` field in `Chart.yaml` to `true`. This is enough to deprecate the latest version of a chart. You can later reuse the chart name and publish a newer version that is not deprecated. The workflow for deprecating charts is:

- Update the chart's `Chart.yaml` file to mark the chart as deprecated and bump the version
- Release the new version to the chart repository
- Remove the chart from the source repository (for example, Git)

Chart metadata files

Charts may contain various metadata files like `README.md`, `LICENSE`, and `NOTES.txt` that describe the installation, configuration, usage, and license of a chart. The `README.md` file should be formatted as Markdown. It should provide the following information:

1. A description of the application or service the chart provides
2. Any prerequisites or requirements to run the chart
3. Description of options in the YAML and default values
4. Any other information that may be relevant to the installation or configuration of the chart

If the chart contains a `templates/NOTES.txt` file, it will be displayed after the installation or when viewing the release status. The notes should be concise to avoid clutter and point to the `README.md` file for detailed explanations. It's common to put usage notes and any next steps in this `NOTES.txt` file. Remember that the file is evaluated as a template. The notes are printed to the screen when you run `helm install` as well as `helm status`.

Managing chart dependencies

In Helm, a chart may depend on other charts. These dependencies are expressed explicitly by listing them in a `requirements.yaml` file or by copying the dependency charts into the `charts` sub-folder during installation. This provides a great way to benefit from and reuse the knowledge and work of others. A dependency can be either a chart archive (`foo-1.2.3.tgz`) or an unpacked chart folder. But its name cannot start with `_` or `..`. Such files are ignored by the chart loader.

Managing dependencies with requirements.yaml

Instead of manually placing charts in the charts sub-folder, it is better to declare dependencies using a `requirements.yaml` file inside of your chart. The following is just an illustration. The charts are fictional.

A `requirements.yaml` file is a simple file used for listing the chart dependencies:

```
dependencies:
- name: foo
  version: 1.2.3
  repository: http://example.com/charts
- name: bar
  version: 43.52.6
  repository: http://another.example.com/charts
```

The name field is the name of the chart you want.

The version field is the version of the chart you want.

The repository field is the full URL to the chart repository.

Note that you must also use `helm repo add` to add the repository locally if it isn't added already.

Once you have a dependencies file, you can run the Helm dependency update, and it will use your dependency file to download all of the specified charts into the charts sub-folder for you:

```
$ helm dep up cool-chart
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "local" chart repository
...Successfully got an update from the "bitnami" chart repository
...Successfully got an update from the "example" chart repository
...Successfully got an update from the "another" chart repository
Update Complete. Happy Helming!
```

Saving 2 charts

Downloading Foo from repo `http://example.com/charts`

Downloading Bar from repo `http://another.example.com/charts`

Helm stores the dependency charts as archives in the charts folder. In our example, the charts sub-folder will contain the following files:

```
charts/
  foo-1.2.3.tgz
  bar-43.52.61.tgz
```

Managing charts and their dependencies with `requirements.yaml` is best practice for explicitly documenting dependencies, sharing across the team, and supporting automated pipelines.

Utilizing special fields in `requirements.yaml`

Each entry in the `requirements.yaml` file may also contain the optional fields `tags` and `condition`.

These fields can be used to dynamically control the loading of charts (by default, all charts are loaded). If the `tags` or `condition` fields are present, Helm will evaluate them and determine if the target chart should be loaded or not:

- **Condition:** The `condition` field holds one or more comma-delimited YAML paths. If a path exists in the top parent's values and resolves to a Boolean value, the chart will be enabled or disabled based on that Boolean value. Only the first valid path found in the list is evaluated, and if no paths exist, then the condition has no effect and the chart will be loaded.
- **Tags:** The `tags` field is a YAML list of labels to associate with this chart. In the top parent's values, all charts with tags can be enabled or disabled by specifying the tag and a Boolean value.

Here are example `requirements.yaml` and `values.yaml` files that make good use of conditions and tags to enable and disable the installation of dependencies. The `requirements.yaml` file defines two conditions for installing its dependencies based on the value of the global `enabled` field and the specific subchart's `enabled` field:

```
# parent/requirements.yaml
dependencies:
- name: subchart1
  repository: http://localhost:10191
  version: 0.1.0
  condition: subchart1.enabled, global.subchart1.enabled
  tags:
    - front-end
    - subchart1
- name: subchart2
  repository: http://localhost:10191
  version: 0.1.0
  condition: subchart2.enabled, global.subchart2.enabled
  tags:
    - back-end
    - subchart2
```

The `values.yaml` file assigns values to some of the condition variables. The `subchart2` tag doesn't get a value, so it is considered to be enabled automatically:

```
# parent/values.yaml
subchart1:
  enabled: true
tags:
  front-end: false
  back-end: true
```

You can set tags and condition values from the command line too when installing a chart, and they'll take precedence over the `values.yaml` file:

```
$ helm install --set subchart2.enabled=false
```

The resolution of tags and conditions is as follows:

- Conditions that are set in values override tags. The first condition path that exists per chart takes effect, while other conditions are ignored.
- If any of a chart's tags are true, the chart is enabled.
- Tags and condition values must be set in the top parent's values.
- The tags' key-in values must be a top-level key. Globals and nested tags tables are not currently supported.

Using templates and values

Any non-trivial application will require you to configure and adapt to the specific use case. Helm charts are templates that use the Go template language to populate placeholders. Helm supports additional functions from the Sprig library and a few other specialized functions. The template files are stored in the `templates/` sub-folder of the chart. Helm will use the template engine to render all of the files in this folder and apply the provided value files.

Writing template files

Template files are just text files that follow the Go template language rules. They can generate Kubernetes configuration files along with any other file. Here is the service template file of the GitLab CE chart:

```
apiVersion: v1
kind: Service
metadata:
  name: {{ template "gitlab-ce.fullname" . }}
```

```
labels:
  app: {{ template "gitlab-ce.fullname" . }}
  chart: "{{ .Chart.Name }}" - "{{ .Chart.Version }}"
  release: "{{ .Release.Name }}"
  heritage: "{{ .Release.Service }}"
spec:
  type: {{ .Values.serviceType }}
  ports:
    - name: ssh
      port: {{ .Values.sshPort | int }}
      targetPort: ssh
    - name: http
      port: {{ .Values.httpPort | int }}
      targetPort: http
    - name: https
      port: {{ .Values.httpsPort | int }}
      targetPort: https
  selector:
    app: {{ template "gitlab-ce.fullname" . }}
```

It is available here: <https://github.com/helm/charts/tree/master/stable/gitlab-ce/templates/svc.yaml>.

Don't worry if it looks confusing. The basic idea is that you have a simple text file with a placeholder for values that can be populated later in various ways, as well as some functions and pipelines that can be applied to those values.

Using pipelines and functions

Helm allows rich and sophisticated syntax in the template files via the built-in Go template functions, sprig functions, and pipelines. Here is an example template that takes advantage of these capabilities. It uses the `repeat`, `quote`, and `upper` functions for the `food` and `drink` keys, and it uses pipelines to chain multiple functions together:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  greeting: "Hello World"
  drink: {{ .Values.favorite.drink | repeat 3 | quote }}
  food: {{ .Values.favorite.food | upper }}
```

Let's add a `values.yaml` file:

```
favorite:
  drink: coffee
  food: pizza
```

Testing and troubleshooting your charts

Now, we can use `helm template` to see the result:

```
$ helm template food food-chart
---
# Source: food-chart/templates/config-map.yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: food-configmap
data:
  greeting: "Hello World"
  drink: "coffeecoffeecoffee"
  food: PIZZA
```

As you can see, our templating worked. The drink coffee was repeated 3 times and quoted. The food pizza became uppercase PIZZA (unquoted).

Another good way of debugging is to run `install` with the `--dry-run` flag. It provides additional information:

```
$ helm install food food-chart --dry-run
NAME: food
LAST DEPLOYED: Mon Jun 8 09:46:19 2020
NAMESPACE: default
STATUS: pending-install
REVISION: 1
TEST SUITE: None
USER-SUPPLIED VALUES:
{}

COMPUTED VALUES:
favorite:
  drink: coffee
  food: pizza
```

HOOKS:

MANIFEST:

Source: food-chart/templates/config-map.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: food-configmap

data:

greeting: "Hello World"

drink: "coffeecoffeecoffee"

food: PIZZA

You can also override values on the command line:

\$ helm template food food-chart --set favorite.drink=water

Source: food-chart/templates/config-map.yaml

apiVersion: v1

kind: ConfigMap

metadata:

name: food-configmap

data:

greeting: "Hello World"

drink: "waterwaterwater"

food: PIZZA

The ultimate test is, of course, to install your chart in your cluster. You don't need to upload your chart to a chart repository for testing; just run `helm install` locally:

\$ helm install food food-chart

NAME: food

LAST DEPLOYED: Mon Jun 8 08:22:36 2020

NAMESPACE: default

STATUS: deployed

REVISION: 1

TEST SUITE: None

There is now a Helm release called `food`:

\$ helm list

NAME	NAMESPACE	REVISION	UPDATED
STATUS	CHART		APP VERSION
food	default	1	2020-06-08 08:22:36.217166 -0700 PDT
deployed	food-chart-0.1.0		1.16.0

More importantly, the food ConfigMap was created with the correct data:

```
$ kubectl get cm -o yaml
apiVersion: v1
items:
- apiVersion: v1
  data:
    drink: coffeecoffeecoffee
    food: PIZZA
    greeting: Hello World
  kind: ConfigMap
  metadata:
    creationTimestamp: "2020-06-08T15:22:36Z"
    name: food-configmap
    namespace: default
    resourceVersion: "313012"
    selfLink: /api/v1/namespaces/default/configmaps/food-configmap
    uid: a3c02518-4fe2-4a72-bdd0-99a268c7033f
  kind: List
  metadata:
    resourceVersion: ""
    selfLink: ""
```

Embedding built-in objects

Helm provides some built-in objects that you can use in your templates. In the GitLab chart template, `Release.Name`, `Release.Service`, `Chart.Name`, and `Chart.Version` are examples of Helm's predefined values. Other objects are:

- `Values`
- `Chart`
- `Template`
- `Files`
- `Capabilities`

The `Values` object contains all the values defined in the values file or on the command line. The `Chart` object is the content of `Chart.yaml`. The `Template` object contains information about the current template. The `Files` and `Capabilities` objects are map-like objects that allow access via various functions to non-specialized files and any general information about the Kubernetes cluster.

Note that unknown fields in `Chart.yaml` are ignored by the template engine and cannot be used to pass arbitrarily structured data to templates.

Feeding values from a file

Here is part of the GitLab CE default values file. The values from this file are used to populate multiple templates. The values represent defaults that you can override by copying the file and modifying it to fit your needs. Note the useful comments that explain the purpose and various options for each value:

```
## GitLab CE image
## ref: https://hub.docker.com/r/gitlab/gitlab-ce/tags/
##
image: gitlab/gitlab-ce:9.4.1-ce.0

## For minikube, set this to NodePort, elsewhere use LoadBalancer
## ref: http://kubernetes.io/docs/user-guide/services/#publishing-services-
--service-types
##
serviceType: LoadBalancer

## Ingress configuration options
##
ingress:
  annotations:
    # kubernetes.io/ingress.class: nginx
    # kubernetes.io/tls-acme: "true"
  enabled: false
  tls:
    # - secretName: gitlab.cluster.local
    #   hosts:
    #     - gitlab.cluster.local
  url: gitlab.cluster.local

## Configure external service ports
## ref: http://kubernetes.io/docs/user-guide/services/
sshPort: 22
httpPort: 80
httpsPort: 443
```

Here is how to provide your own YAML values file to override the defaults during the install command:

```
$ helm install --values=custom-values.yaml gitlab-ce
```

Scope, dependencies, and values

value files can declare values for the top-level chart, as well as for any of the charts that are included in that chart's charts folder. For example, let's look at the sentry chart from <https://github.com/sapcc/helm-charts/blob/master/system/sentry>.

This chart has two chart dependencies: postgresql and redis:

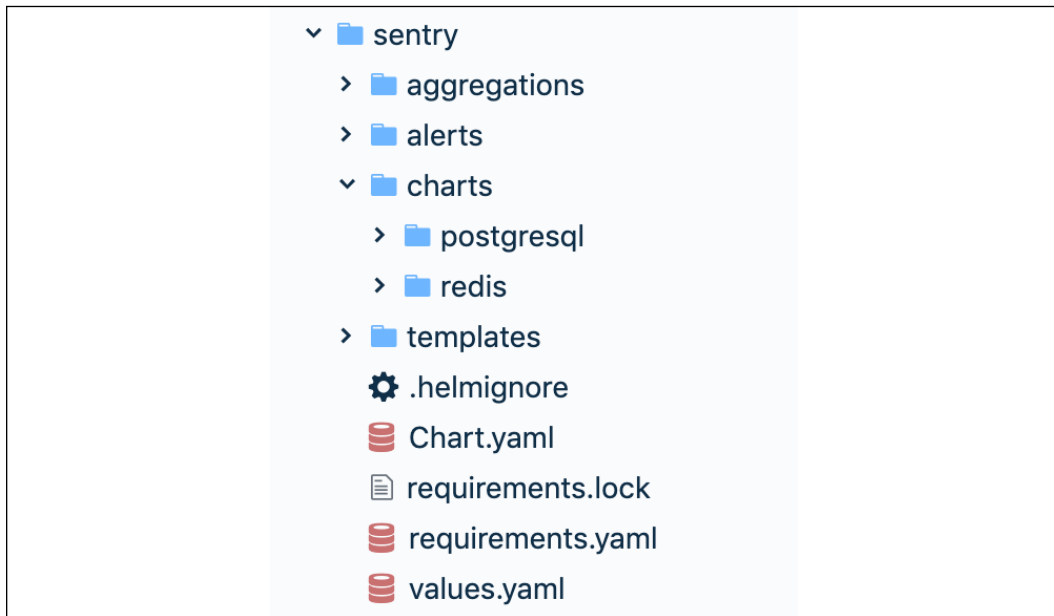


Figure 9.1: sentry chart

Both the postgresql and redis charts have their own values.yaml file with their defaults. However, the top-level values.yaml file contains some default values for its dependency charts, postgresql and redis:

```
postgresql:
  postgresDatabase: sentry
  persistence:
    enabled: true
    accessMode: ReadWriteMany
    size: 50Gi
```

```
resources:
  requests:
    memory: 10Gi
    cpu: 4

redis:
  # redisPassword:
  persistence:
    enabled: true
    accessMode: ReadWriteMany
    size: 10Gi
  resources:
    requests:
      memory: 10Gi
      cpu: 2
```

The top-level chart has access to the values of its dependent charts, but not vice versa. There is also a global value that is accessible to all charts. For example, you could add something like this:

```
global:
  app: cool-app
```

When a global value is present, it will be replicated to each dependent chart's values, as follows:

```
global:
  app: cool-app
postgresql:
  global:
    app: cool-app
  ...
redis:
  global:
    app: cool-app
  ...
```

Summary

In this chapter, we took a look at Helm, the Kubernetes package manager. Helm gives Kubernetes the ability to manage complicated software composed of many Kubernetes resources with inter-dependencies. It serves the same purpose as an OS package manager. It organizes packages and lets you search charts, install and upgrade charts, and share charts with collaborators. You can develop your own charts and store them in repositories. Helm 3 is a client-side-only solution that uses CRDs to manage the status of releases, instead of the Tiller server-side component of Helm 2, which poses a lot of security issues with its default configuration.

At this point, you should be able to understand the important role that Helm serves in the Kubernetes ecosystem and community. You should be able to use it productively and even develop and share your own charts.

In the next chapter, we will look at how Kubernetes does networking at a pretty low level.

10

Exploring Advanced Networking

In this chapter, we will examine the important topic of networking. Kubernetes as an orchestration platform manages containers/pods running on different machines (physical or virtual) and requires an explicit networking model. We will look at the following topics:

- The Kubernetes networking model
- Standard interfaces that Kubernetes supports, such as EXEC, Kubenet, and, in particular, CNI
- Various networking solutions that satisfy the requirements of Kubernetes networking
- Network policies and load balancing options
- Writing a custom CNI plugin

By the end of this chapter, you will understand the Kubernetes approach to networking and be familiar with the solution space for aspects such as standard interfaces, networking implementations, and load balancing. You will even be able to write your very own CNI plugin if you wish.

Understanding the Kubernetes networking model

The Kubernetes networking model is based on a flat address space. All pods in a cluster can directly see one another. Each pod has its own IP address. There is no need to configure any **Network Address Translation (NAT)**. In addition, containers in the same pod share their pod's IP address and can communicate with one another through localhost. This model is pretty opinionated, but once set up, it makes life considerably easier for both developers and administrators. It makes it particularly easy to migrate traditional network applications to Kubernetes. A pod represents a traditional node and each container represents a traditional process.

Intra-pod communication (container to container)

A running pod is always scheduled on one (physical or virtual) node. This means that all the containers run on the same node and can talk to each other in various ways, such as using the local filesystem, any IPC mechanism, or using localhost and well-known ports. There is no danger of port collision between different pods because each pod has its own IP address and, when a container in the pod uses localhost, it applies to the pod's IP address only. So, if container 1 in pod 1 connects to port 1234 that container 2 listens to on pod 1, it will not conflict with another container in pod 2 running on the same node that also listens on port 1234. The only caveat is that if you're exposing ports to the host, then you should be careful about pod-to-node affinity. This can be handled using several mechanisms, including DaemonSet and pod anti-affinity.

Inter-pod communication (pod to pod)

Pods in Kubernetes are allocated a network-visible IP address (not private to the node). Pods can communicate directly without the help of network address translation, tunnels, proxies, or any other obfuscating layer. Well-known port numbers can be used for a configuration-free communication scheme. The pod's internal IP address is the same as its external IP address that other pods see (within the cluster network; not exposed to the outside world). This means that standard naming and discovery mechanisms such as DNS work out of the box.

Pod-to-service communication

Pods can talk to one another directly using their IP addresses and well-known ports, but that requires the pods to know each other's IP addresses. In a Kubernetes cluster, pods can be destroyed and created constantly. There may also be multiple replicas of the same pod spec, each with its own IP address. The service provides a layer of indirection that is very useful because the service is stable even if the set of actual pods that respond to requests is ever-changing. In addition, you get automatic, highly available load balancing because the kube-proxy on each node takes care of redirecting traffic to the correct pod:

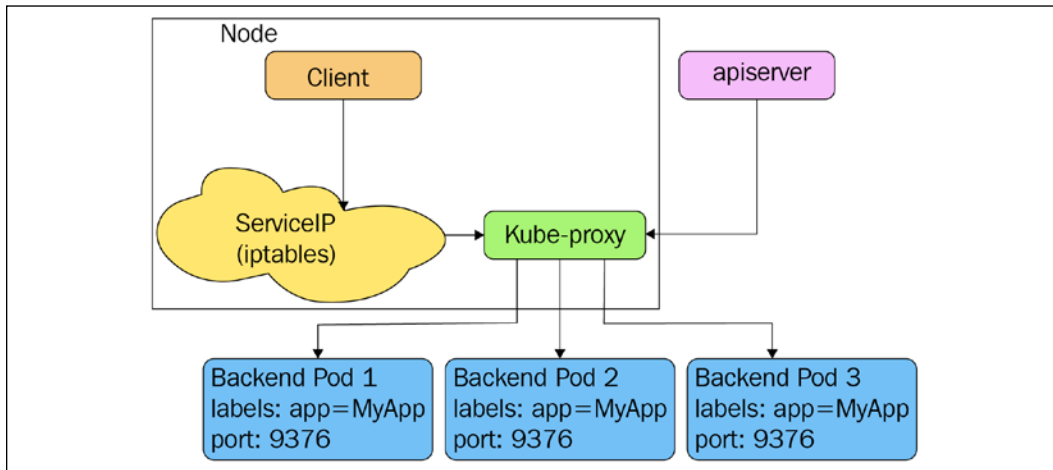


Figure 10.1: kube-proxy redirecting traffic to pods

External access

Eventually, some containers need be accessible from the outside world. The pod IP addresses are not visible externally. The service is the right vehicle, but external access typically requires two redirects. For example, cloud provider load balancers are not Kubernetes aware, so they can't direct traffic to a particular service directly to a node that runs a pod that can process the request. Instead, the public load balancer just directs traffic to any node in the cluster and the kube-proxy on that node will redirect again to an appropriate pod if the current node doesn't run the necessary pod.

The following diagram shows how the external load balancer on the right side just sends traffic to an arbitrary node, where the kube-proxy takes care of further routing if needed:

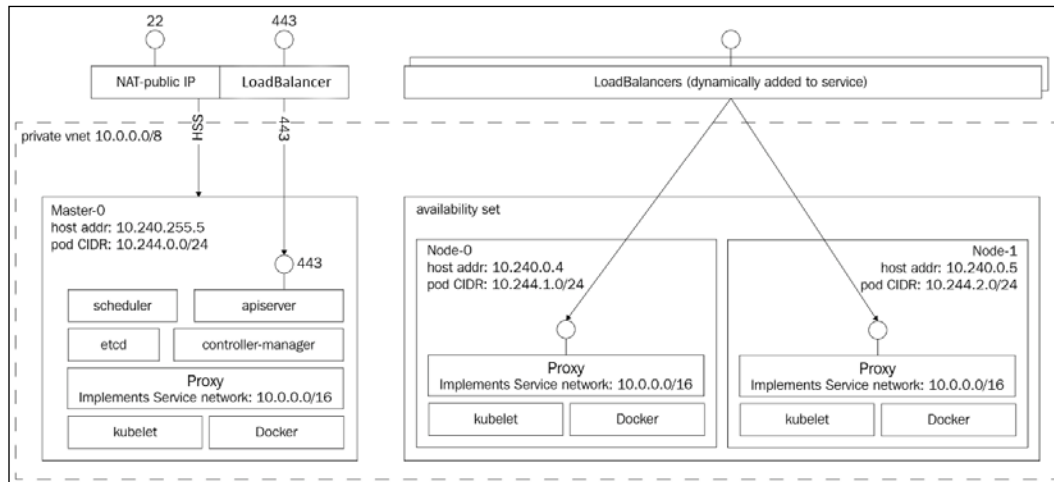


Figure 10.2: External versus internal load balancers

Kubernetes networking versus Docker networking

Docker is the incumbent container runtime. It also has its own separate networking model that is not used by Kubernetes, but it is useful to understand. Docker networking follows a different model by default, although over time, it starts to gravitate toward the Kubernetes model. In Docker networking, each container has its own private IP address from the 172.xxx.xxx.xxx address space confined to its own node. It can talk to other containers on the same node via their own 172.xxx.xxx.xxx different IP addresses. This makes sense for Docker because it doesn't have the notion of a pod with multiple interacting containers, so it models every container as lightweight VMs that have their own network identity. Note that with Kubernetes, containers from different pods that run on the same node can't connect over localhost (unless exposing host ports, which is discouraged). The whole idea is that, in general, Kubernetes can kill and create pods anywhere, so different pods shouldn't rely, in general, on other pods available on the node. DaemonSets are a notable exception, but the Kubernetes networking model is designed to work for all use cases and doesn't add special cases for direct communication between different pods on the same node.

How do Docker containers communicate across nodes? The container must publish ports to the host. This obviously requires port coordination, because if two containers try to publish the same host port, they'll conflict with one another. Then, containers (or other processes) connect to the host's port that gets channeled into the container. A big downside is that containers can't self-register with external services because they don't know what their host's IP address is. You could work around this by passing the host's IP address as an environment variable when you run the container, but that requires external coordination and complicates the process.

The following diagram shows the networking setup with Docker using the bridge network driver. Each container has its own IP address; Docker creates the `docker0` bridge on every node:

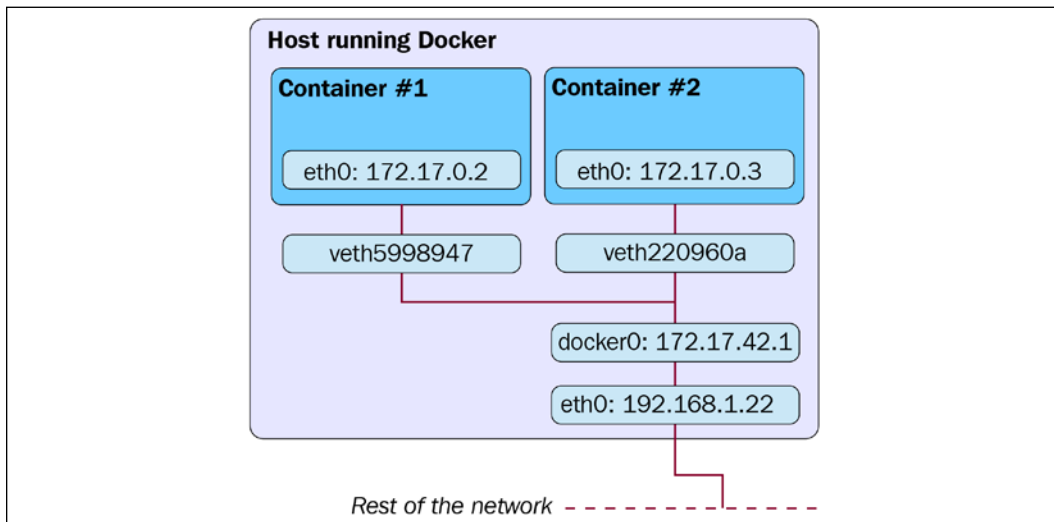


Figure 10.3: Networking setup with Docker using the bridge network driver

Docker now supports other network drivers with their own models:

- `host`: Use host networking directly
- `overlay`: Use an overlay network instead of OS routing to connect across Docker daemons
- `macvlan`: Assign a MAC address to a container and make it look like a physical device
- `none`: Disable networking when using a custom network driver

There are also third-party network plugins.

Now that we understand the differences between Kubernetes and Docker networking models, it's time to talk about how pods and containers find one another.

Lookup and discovery

In order for pods and containers to communicate with each other, they need to find one another. There are several ways for containers to locate other containers or announce themselves. There are also some architectural patterns that allow containers to interact indirectly. Each approach has its own pros and cons.

Self-registration

We've mentioned self-registration several times. Let's understand what it means exactly. When a container runs, it knows its pod's IP address. Each container that wants to be accessible to other containers in the cluster can connect to a registration service and register its IP address and port. Other containers can query the registration service for the IP addresses and ports of all registered containers and connect to them. When a container is destroyed (gracefully), it will unregister itself. If a container dies ungracefully, then a mechanism needs to be established to detect that. For example, the registration service can periodically ping all registered containers, or the containers are required periodically to send a keepalive message to the registration service.

The benefit of self-registration is that once the generic registration service is in place (no need to customize it for different purposes), there is no need to worry about keeping track of containers. Another huge benefit is that containers can employ sophisticated policies and decide to unregister temporarily if they are unavailable based on local conditions; for example, if a container is busy and doesn't want to receive any more requests at the moment. This sort of smart and decentralized dynamic load balancing can be very difficult to achieve globally. The downside is that the registration service is yet another non-standard component that containers need to know about in order to locate other containers.

Services and endpoints

Kubernetes services can be regarded as a registration service. Pods that belong to a service are registered automatically based on their labels. Other pods can look up the endpoints to find all the service pods or take advantage of the service itself and directly send a message to the service that will get routed to one of the backend pods, although, most of the time, pods will just send their message to the service itself that will forward it to one of the backing pods. Dynamic membership can be achieved using a combination of the replica count of deployments, health checks, readiness checks, and horizontal pod autoscaling.

Loosely coupled connectivity with queues

What if containers can talk to one another without knowing their IP addresses and ports or even service IP addresses or network names? What if most of the communication can be asynchronous and decoupled? In many cases, systems can be composed of loosely coupled components that are not only unaware of the identities of other components, but they are unaware that other components even exist. Queues facilitate such loosely coupled systems. Components (containers) listen to messages from the queue, respond to messages, perform their jobs, and post messages to the queue regarding progress, completion status, and errors. Queues have many benefits:

- It is easy to add processing capacity without coordination; just add more containers that listen to the queue
- It is easy to keep track of overall load by means of queue depth
- It is easy to have multiple versions of components running side by side by versioning messages and/or topics
- It is easy to implement load balancing as well as redundancy by having multiple consumers process requests in different modes
- It is easy to add or remove other types of listeners dynamically

The downsides of queues are the following:

- There is a need to ensure that the queue provides appropriate durability and high availability so that it doesn't become a critical **single point of failure (SPOF)**
- Containers need to work with the async queue API (could be abstracted away)
- Implementing request-response requires somewhat cumbersome listening on response queues

Overall, queues are an excellent mechanism for large-scale systems and they can be utilized in large Kubernetes clusters to ease coordination.

Loosely coupled connectivity with data stores

Another loosely coupled method is to use a data store (for example, Redis) to store messages and then other containers can read them. While possible, this is not the design objective of data stores and the result is often cumbersome, fragile, and doesn't have the best performance. Data stores are optimized for data storage and not for communication.

That said, data stores can be used in conjunction with queues, where a component stores some data in a data store and then sends a message to the queue that data is ready for processing. Multiple components listen to the message and all start processing the data in parallel.

Kubernetes ingress

Kubernetes offers an ingress resource and controller that is designed to expose Kubernetes services to the outside world. You can do it yourself, of course, but many tasks involved in defining ingress are common across most applications for a particular type of ingress such as a web application, CDN, or DDoS protector. You can also write your own ingress objects.

The ingress object is often used for smart load balancing and TLS termination. Instead of configuring and deploying your own Nginx server, you can benefit from the built-in ingress controller. If you need a refresher, hop over to *Chapter 5, Using Kubernetes Resources in Practice*, where we discussed the ingress resource with examples.

Kubernetes network plugins

Kubernetes has a network plugin system since networking is so diverse and different and people would like to implement it in different ways. Kubernetes is flexible enough to support any scenario. The primary network plugin is CNI, which we will discuss in depth. But Kubernetes also comes with a simpler network plugin, called **Kubenet**. Before we go over the details, let's get on the same page with the basics of Linux networking (just the tip of the iceberg).

Basic Linux networking

Linux, by default, has a single shared network space. The physical network interfaces are all accessible in this namespace. However, the physical namespace can be divided into multiple logical namespaces, which is very relevant to container networking.

IP addresses and ports

Network entities are identified by their IP address. Servers can listen to incoming connections on multiple ports. Clients can connect to (TCP)s or send/receive data from (UDP), servers within their network.

Network namespaces

Namespaces group a bunch of network devices such that they can reach other servers in the same namespace, but not other servers, even if they are physically on the same network. Linking networks or network segments can be done via bridges, switches, gateways, and routing.

Subnets, netmasks, and CIDRs

The granular division of network segments is very useful when designing and maintaining networks. Dividing networks into smaller subnets with a common prefix is a common practice. These subnets can be defined by bitmasks that represent the size of the subnet (how many hosts it can contain). For example, a netmask of 255.255.255.0 means that the first 3 octets are used for routing and only 256 (actually 254) individual hosts are available. The **Classless Inter-Domain Routing (CIDR)** notation is often used for this purpose because it is more concise, encodes more information, and also allows hosts from multiple legacy classes (A, B, C, D, E) to be combined. For example, 172.27.15.0/24 means that the first 24 bits (3 octets) are used for routing.

Virtual Ethernet devices

Virtual Ethernet (veth) devices represent physical network devices. When you create a veth device that's linked to a physical device, you can assign that veth device (and, by extension, the physical device) to a namespace where devices from other namespaces can't reach it directly, even if, physically, they are on the same local network.

Bridges

Bridges connect multiple network segments to an aggregate network, so that all the nodes can communicate with one another. Bridging is done at the L1 (physical) and L2 (data link) layers of the OSI network model.

Routing

Routing connects separate networks, typically based on routing tables that instruct network devices how to forward packets to their destination. Routing is done through various network devices, such as routers, bridges, gateways, switches, and firewalls, including regular Linux boxes.

Maximum transmission unit

The **maximum transmission unit (MTU)** determines how big packets can be. On Ethernet networks, for example, the MTU is 1,500 bytes. The bigger the MTU, the better the ratio between payload and headers, which is a good thing. However, the downside is that minimum latency is reduced because you have to wait for the entire packet to arrive and, furthermore, in the case of failure, you have to retransmit the entire big packet.

Pod networking

Here is a diagram that describes the relationship between pod, host, and the global internet at the networking level via **veth0**:

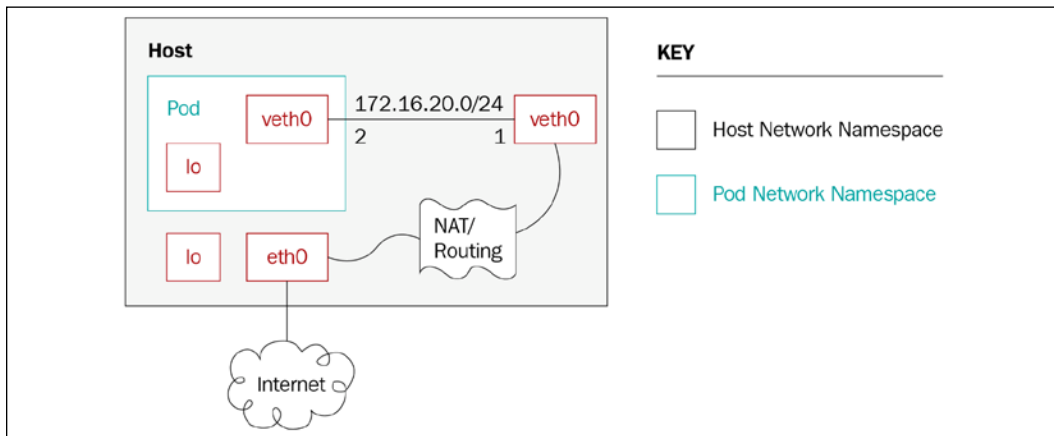


Figure 10.4: veth relationships

Kubenet

Back to Kubernetes. Kubenet is a network plugin. It's very rudimentary and just creates a Linux bridge called **cbr0** and a veth device for each pod. Cloud providers typically use it to set up routing rules for communication between nodes, or in single-node environments. The veth pair connects each pod to its host node using an IP address from the host's IP addresses range.

Requirements

The Kubenet plugin has the following requirements:

- The node must be assigned a subnet to allocate IP addresses for its pods

- The standard CNI bridge, **io**, and host-local plugins are required at version 0.2.0 or greater
- The kubelet must be run with the `--network-plugin=kubenet` argument
- The kubelet must be run with the `--non-masquerade-cidr=<clusterCidr>` argument
- The kubelet must be run with `--pod-cidr`, or the kube-controller-manager must be run with `--allocate-node-cidrs=true --cluster-cidr=<cidr>`

Setting the MTU

The MTU is critical for network performance. Kubernetes network plugins such as Kubenet make their best efforts to deduce optimal MTU, but sometimes they need help. If an existing network interface (for example, the Docker `docker0` bridge) sets a small MTU, then Kubenet will reuse it. Another example is IPSEC, which requires a lowering of the MTU due to the extra overhead from IPSEC encapsulation, but the Kubenet network plugin doesn't take it into consideration. The solution is to avoid relying on the automatic calculation of the MTU and just tell the kubelet what MTU should be used for network plugins via the `--network-plugin-mtu` command-line switch that is provided to all network plugins, although, at the moment, only the Kubenet network plugin accounts for this command-line switch.

Container networking interface

Container Networking Interface (CNI) is a specification as well as a set of libraries for writing network plugins to configure network interfaces in Linux containers (not just Docker). The specification actually evolved from the rkt network proposal. There is a lot of momentum behind CNI and it is the established industry standard. Some of the organizations that use CNI are as follows:

- Kubernetes
- OpenShift
- Mesos
- Kurma
- Cloud Foundry
- Nuage
- IBM
- AWS EKS and ECS
- Lyft

The CNI team maintains some core plugins, but there are a lot of third-party plugins as well that contribute to the success of CNI. Here is a non-exhaustive list:

- **Project Calico:** A layer 3 virtual network
- **Weave:** A multi-host Docker network
- **Contiv networking:** Policy-based networking
- **Cilium:** BPF and XDP for containers
- **Multus:** A multi plugin
- **CNI-Genie:** A generic CNI network plugin
- **Flannel:** A network fabric for containers, designed for Kubernetes
- **Infoblox:** Enterprise IP address management for containers
- **Silk:** A CNI plugin designed for Cloud Foundry
- **Linen:** A CNI plugin designed for overlay networks with Open vSwitch and that fits in an SDN/OpenFlow network environment
- **SR-IOV:** A CNI plugin that supports I/O virtualization
- **ovn-kubernetes:** A CNI plugin built on **Open vSwitch (OVS)** and **Open Virtual Networking (OVN)**
- **DANM:** A CNI-compliant networking solution for TelCo workloads running on Kubernetes

CNI plugins provide a standard networking interface to arbitrary networking solutions.

Container runtime

CNI defines a plugin spec for networking application containers, but the plugin must be plugged into a container runtime that provides some services. In the context of CNI, an application container is a network-addressable entity (has its own IP address). For Docker, each container has its own IP address. For Kubernetes, each pod has its own IP address, and the pod is the CNI container and not the containers within the pod.

Likewise, rkt's app containers are similar to Kubernetes pods in that they may contain multiple Linux containers. If in doubt, just remember that a CNI container must have its own IP address. The runtime's job is to configure a network and then execute one or more CNI plugins, passing them the network configuration in JSON format.

The following diagram shows a container runtime using the CNI plugin interface to communicate with multiple CNI plugins:

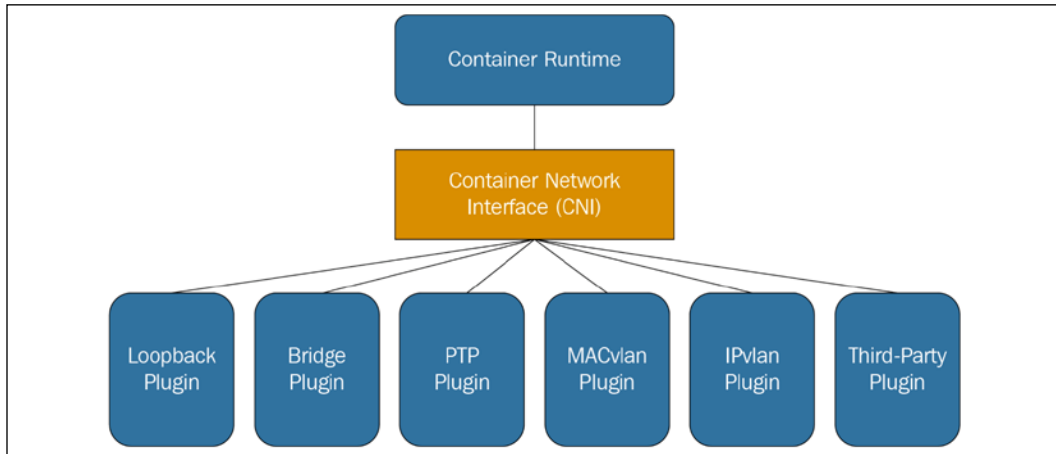


Figure 10.5: The CNI plugin interface in practice

CNI plugin

The CNI plugin's job is to add a network interface to the container network namespace and bridge the container to the host via a veth pair. It should then assign an IP address via an **IP Address Management (IPAM)** plugin and set up routes.

The container runtime (Docker, rkt, or any other CRI-compliant runtime) invokes the CNI plugin as an executable. The plugin needs to support the following operations:

- Add a container to the network
- Remove a container from the network
- Report the version

The plugin uses a simple command-line interface, standard input/output, and environment variables. The network configuration in JSON format is passed to the plugin through standard input. The other arguments are defined as environment variables:

- **CNI_COMMAND**: Indicates the desired operation: ADD, DEL, or VERSION
- **CNI_CONTAINERID**: Container ID
- **CNI_NETNS**: Path to the network namespace file

- CNI_IFNAME: Interface name to set up; the plugin must honor this interface name or return an error
- CNI_ARGS: Extra arguments passed in by the user at invocation time; alphanumeric key-value pairs separated by semicolons, for example, FOO=BAR;ABC=123
- CNI_PATH: List of paths to search for CNI plugin executables; paths are separated by an OS-specific list separator, for example, : on Linux and ; on Windows

If the command succeeds, the plugin returns a zero-exit code and the generated interfaces (in the case of the ADD command) are streamed to standard output as JSON. This low-tech interface is smart in the sense that it doesn't require any specific programming language or component technology or binary API. CNI plugin writers can use their favorite programming language, too.

The result of invoking the CNI plugin with the ADD command appears as follows:

```
{
  "cniVersion": "0.3.0",
  "interfaces": [                (this key omitted by IPAM plugins)
    {
      "name": "<name>",
      "mac": "<MAC address>", (required if L2 addresses are
meaningful)
      "sandbox": "<netns path or hypervisor identifier>" (required
for container/hypervisor interfaces, empty/omitted for host interfaces)
    }
  ],
  "ip": [
    {
      "version": "<4-or-6>",
      "address": "<ip-and-prefix-in-CIDR>",
      "gateway": "<ip-address-of-the-gateway>",    (optional)
      "interface": <numeric index into 'interfaces' list>
    },
    ...
  ],
  "routes": [                    (optional)
    {
      "dst": "<ip-and-prefix-in-cidr>",
```

```

        "gw": "<ip-of-next-hop>"                (optional)
    },
    ...
]
"dns": {
    "nameservers": <list-of-nameservers>        (optional)
    "domain": <name-of-local-domain>            (optional)
    "search": <list-of-additional-search-domains> (optional)
    "options": <list-of-options>                (optional)
}
}

```

The input network configuration contains a lot of information: `cniVersion`, `name`, `type`, `args` (optional), `ipMasq` (optional), `ipam`, and `dns`. The `ipam` and `dns` parameters are dictionaries with their own specific keys. Here is an example of a network configuration:

```

{
  "cniVersion": "0.3.0",
  "name": "dbnet",
  "type": "bridge",
  // type (plugin) specific
  "bridge": "cni0",
  "ipam": {
    "type": "host-local",
    // ipam specific
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1"
  },
  "dns": {
    "nameservers": ["10.1.0.1"]
  }
}

```

Note that additional plugin-specific elements can be added. In this case, the `bridge: cni0` element is a custom one that the specific bridge plugin understands.

The CNI spec also supports network configuration lists where multiple CNI plugins can be invoked in order.

Later in this chapter, we will dig into a fully fledged implementation of a CNI plugin.

Kubernetes networking solutions

Networking is a vast topic. There are many ways to set up networks and connect devices, pods, and containers. Kubernetes can't be opinionated about it. The high-level networking model of a flat address space for pods is all that Kubernetes prescribes. Within that space, many valid solutions are possible, with various capabilities and policies for different environments. In this section, we'll examine some of the available solutions and understand how they map to the Kubernetes networking model.

Bridging on bare metal clusters

The most basic environment is a raw bare metal cluster with just an L2 physical network. You can connect your containers to the physical network with a Linux bridge device. The procedure is quite involved and requires familiarity with low-level Linux network commands such as `brctl`, `ipaddr`, `iproute`, `iplink`, and `nsenter`. If you plan to implement it, this guide can serve as a good start (search for the *With Linux Bridge devices* section): <http://blog.oddbit.com/2014/08/11/four-ways-to-connect-a-docker/>.

Contiv

Contiv is a general-purpose network plugin for container networking that can be used with Docker directly, Mesos, Docker Swarm, and, of course, Kubernetes via a CNI plugin. Contiv is focused on network policies that overlap somewhat with Kubernetes' own network policy object. Here are some of the capabilities of the Contiv net plugin:

- Supports both Libnetwork's CNM and the CNI specification
- A feature-rich policy model for providing secure, predictable application deployment
- Best-in-class throughput for container workloads
- Multi-tenancy, isolation, and overlapping subnets

- Integrated IPAM and service discovery
- A variety of physical topologies:
- Layer2 (VLAN)
- Layer3 (BGP)
- Overlay (VXLAN)
- Cisco SDN solution (ACI)
- IPv6 support
- Scalable policy and route distribution

Integration with application blueprints, including the following:

- Docker Compose
- Kubernetes deployment manager
- Service load balancing is incorporated in east-west microservice load balancing
- Traffic isolation for storage, control (for example, etcd/consul), network, and management traffic

Contiv has many features and capabilities. However, I'm not sure if it's the best choice for Kubernetes due to its broad surface area and the fact that it caters to multiple platforms.

Open vSwitch

Open vSwitch is a mature software-based virtual switch solution endorsed by many big players. The **Open Virtualization Network (OVN)** solution lets you build various virtual networking topologies. It has a dedicated Kubernetes plugin, but it is not easy to set up, as demonstrated by this guide: <https://github.com/openvswitch/ovn-kubernetes>. The Linen CNI plugin may be easier to set up although it doesn't support all the features of OVN:

<https://github.com/John-Lin/linen-cni>

Here is a diagram of the Linen CNI plugin:

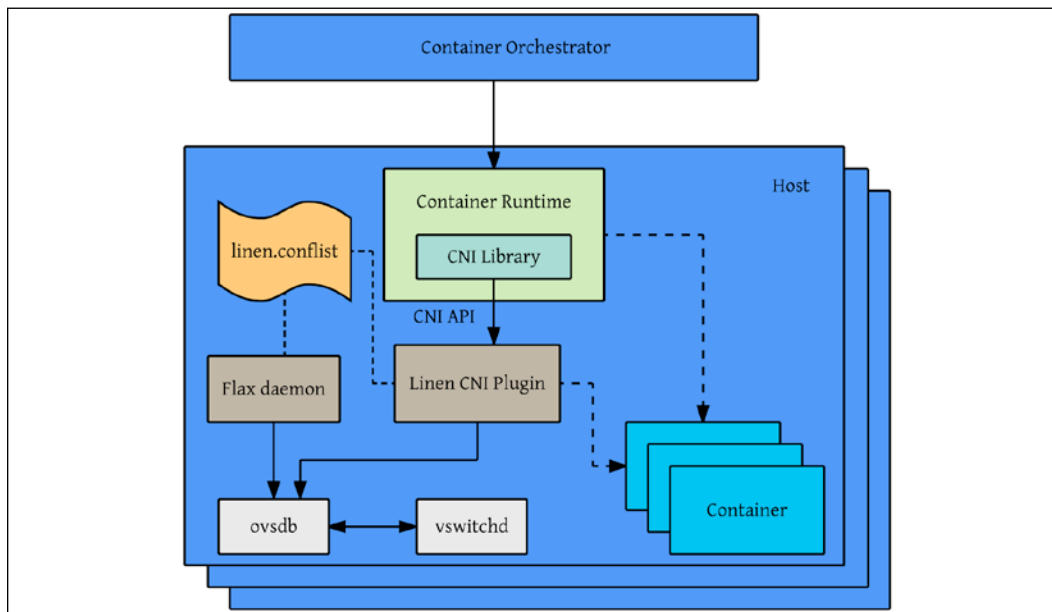


Figure 10.6: The Linen CNI plugin

Open vSwitch can connect bare metal servers, VMs, and pods/containers using the same logical network. It actually supports both overlay and underlay modes.

Here are some of its key features:

- Standard 802.1Q VLAN model with trunk and access ports
- NIC bonding with or without LACP on the upstream switch
- NetFlow, sFlow(R), and mirroring for increased visibility
- **Quality of Service (QoS)** configuration, plus policing
- Geneve, GRE, VXLAN, STT, and LISP tunneling

- 802.1ag connectivity fault management
- OpenFlow 1.0 plus numerous extensions
- Transactional configuration database with C and Python bindings
- High-performance forwarding using a Linux kernel module

Nuage networks VCS

The **Virtualized Cloud Services (VCS)** product from Nuage networks provides a highly scalable, policy-based **Software-Defined Networking (SDN)** platform. It is an enterprise-grade offering that builds on top of the open source open vSwitch for the data plane, along with a feature-rich SDN controller built on open standards.

The Nuage platform uses overlays to provide seamless policy-based networking between Kubernetes pods and non-Kubernetes environments (VMs and bare metal servers). Nuage's policy abstraction model is designed with applications in mind and makes it easy to declare fine-grained policies for applications. The platform's real-time analytics engine enables visibility and security monitoring for Kubernetes applications.

In addition, all VCS components can be installed in containers. There are no special hardware requirements.

Flannel

Flannel is a virtual network that gives a subnet to each host for use with container runtimes. It runs a `flannel` agent on each host that allocates a subnet to the node from a reserved address space stored in `etcd`. Forwarding packets between containers and, ultimately, hosts is done by one of multiple backends. The most common backend uses UDP over a TUN device that tunnels through port 8285 by default (make sure it's open in your firewall).

The following diagram describes in detail the various components of Flannel, the virtual network devices it creates, and how they interact with the host and the pod via the `docker0` bridge.

It also shows the UDP encapsulation of packets and how they are transmitted between hosts:

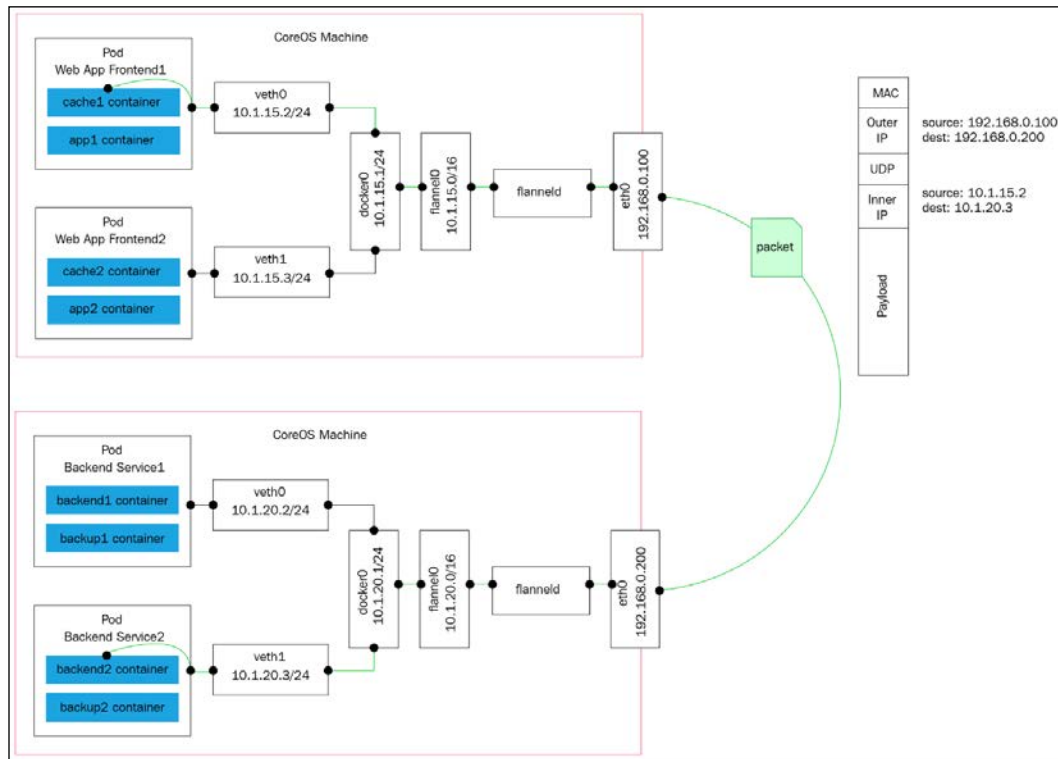


Figure 10.7: Flannel

Other backends include the following:

- **vxlan**: Uses in-kernel VXLAN to encapsulate the packets.
- **host-gw**: Creates IP routes to subnets via remote machine IPs. Note that this requires direct layer2 connectivity between hosts running Flannel.
- **aws-vpc**: Creates IP routes in an Amazon VPC route table.
- **gce**: Creates IP routes in a Google Compute Engine network.
- **alloc**: Only performs subnet allocation (no forwarding of data packets).
- **ali-vpc**: Creates IP routes in an Alibaba Cloud VPC route table.

Calico

Calico is a versatile virtual networking and network security solution for containers. Calico can integrate with all the primary container orchestration frameworks and runtimes:

- Kubernetes (CNI plugin)
- Mesos (CNI plugin)
- Docker (Libnetwork plugin)
- OpenStack (Neutron plugin)

Calico can also be deployed on-premises or on public clouds with its full feature set. Calico's network policy enforcement can be specialized for each workload and ensures that traffic is controlled precisely and packets always go from their source to vetted destinations. Calico can map automatically network policy concepts from orchestration platforms to its own network policy. The reference implementation of Kubernetes' network policy is Calico. Calico can be deployed together with Flannel utilizing the Flannel networking layer and Calico's network policy facilities.

Romana

Romana is a modern cloud-native container networking solution. It operates at layer 3, taking advantage of standard IP address management techniques. Whole networks can become the unit of isolation as Romana uses Linux hosts to create gateways and routes to the networks. Operating at layer 3 means that no encapsulation is needed. Network policy is enforced as a distributed firewall across all endpoints and services. Hybrid deployments across cloud platforms and on-premises deployments are easier as there is no need to configure virtual overlay networks. New Romana virtual IPs allow on-premises users to expose services on layer 2 LANs via external IPs and service specs.

Some of the benefits of using real routable IP addresses are as follows:

- **Performance:** Traffic is forwarded and processed by hosts and network equipment at full speed; no cycles are spent encapsulating packets.
- **Scalability:** Native, routed IP networking offers tremendous scalability, as demonstrated by the internet itself. Romana's use of routed IP addressing for endpoints means that no time, CPU, or memory-intensive tunnels or other encapsulation needs to be managed or maintained and that network equipment can run at optimal efficiency.
- **Visibility:** Packet traces show the real IP addresses, allowing easier troubleshooting and traffic management.

The following diagram shows how Romana eliminates a lot of the overhead by using direct L2 routing, where **ToR** stands for the **top-of-rack** switch:

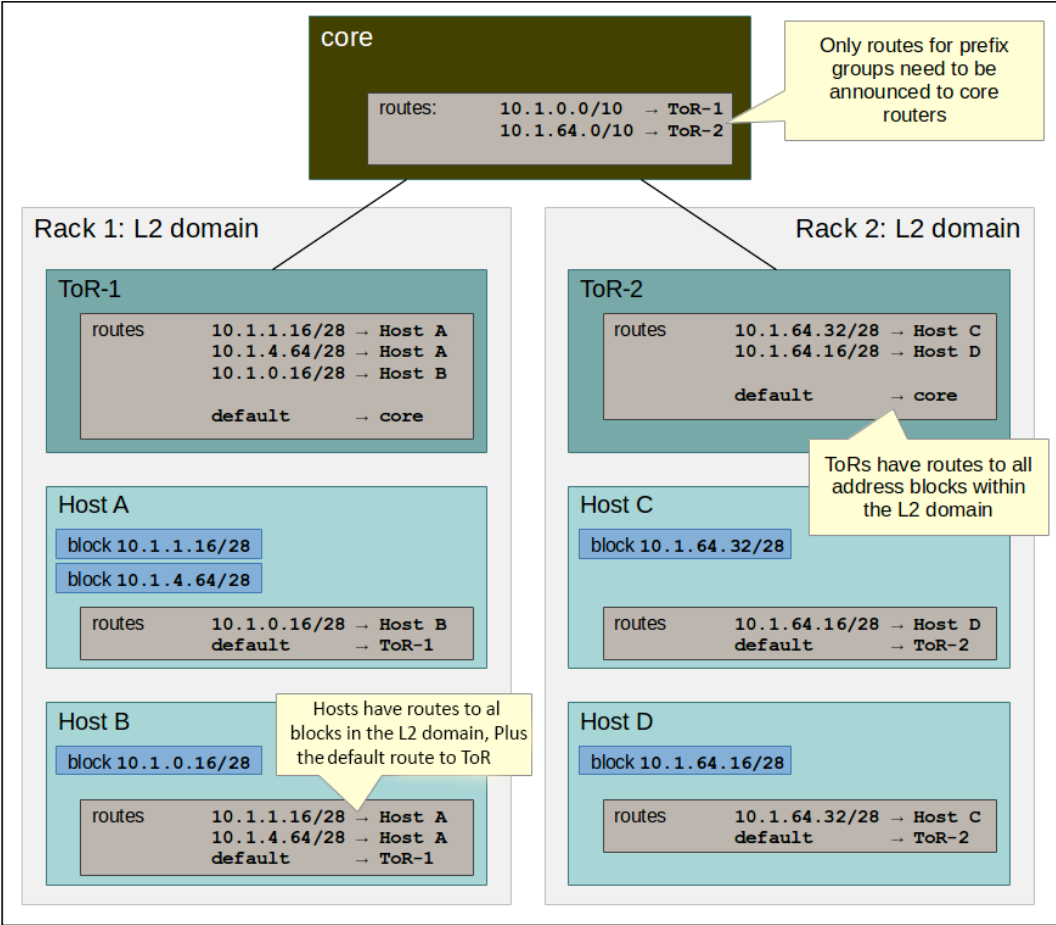


Figure 10.8: Romana blocks and routes in an L2-to-host data center

When networks are configured for L3-to-host routing, where hosts don't necessarily share an L2 segment, in this case, there is no need to configure routes between hosts that use the default route to the ToR switch. Here is a diagram to illustrate this:

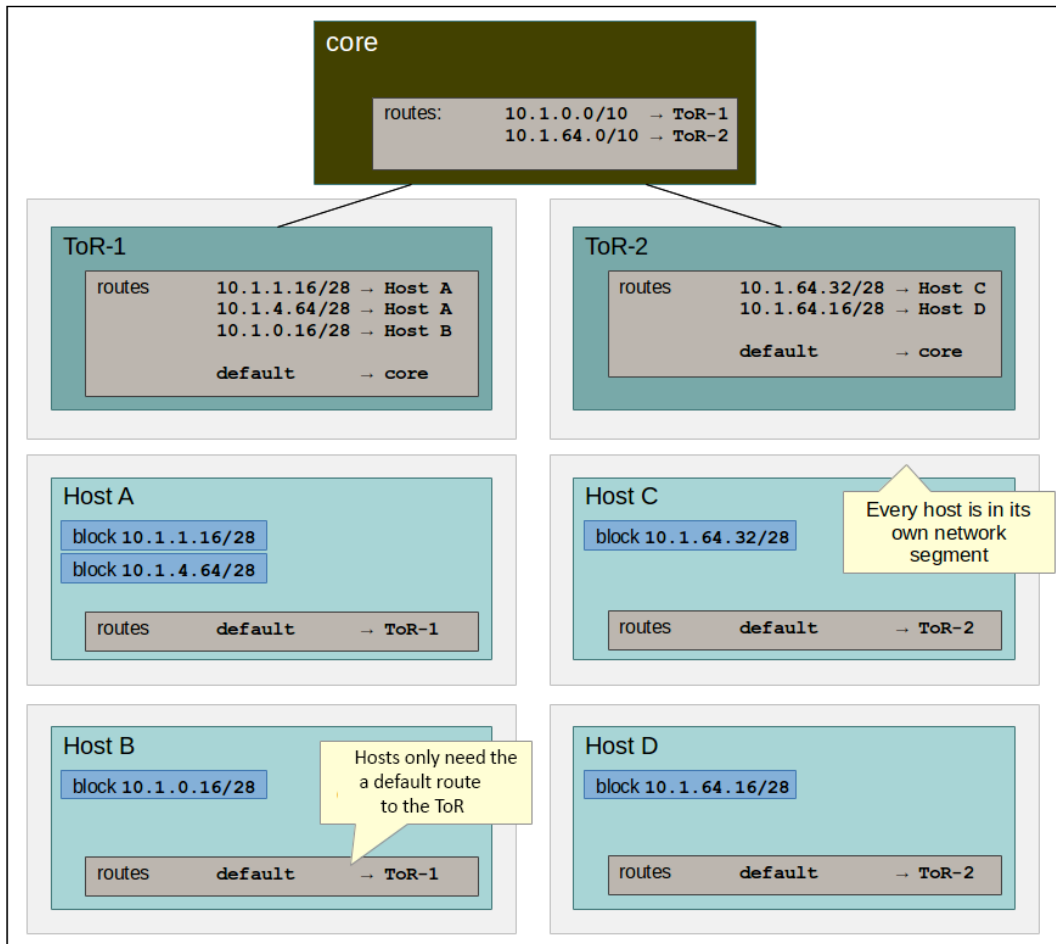


Figure 10.9: Romana blocks and routes in an L3-to-host data center

Weave Net

Weave Net is all about ease of use and zero configuration. It uses VXLAN encapsulation under the hood and micro DNS on each node. As a developer, you operate at a higher abstraction level. You name your containers and Weave Net lets you connect to them and use standard ports for services. That helps to migrate existing applications into containerized applications and microservices. Weave Net has a CNI plugin for interfacing with Kubernetes (and Mesos). On Kubernetes 1.4 and higher, you can integrate Weave Net with Kubernetes by running a single command that deploys a DaemonSet:

```
kubect1 apply -f https://git.io/weave-kube
```

The Weave Net pods on every node will take care of attaching any new pod you create to the Weave network. Weave Net supports the network policy API, while also providing a complete, yet easy-to-setup, solution.

Using network policies effectively

The Kubernetes network policy is about managing network traffic to selected pods and namespaces. In a world of hundreds of microservices deployed and orchestrated, as is often the case with Kubernetes, managing networking and connectivity between pods is essential. It's important to understand that it is not primarily a security mechanism. If an attacker can reach the internal network, they will probably be able to create their own pods that comply with the network policy in place and communicate freely with other pods. In the previous section, we looked at different Kubernetes networking solutions and focused on the container networking interface. In this section, the focus is on network policy, although there are strong connections between the networking solution and how network policy is implemented on top of it.

Understanding the Kubernetes network policy design

A network policy is a specification of how selections of pods can communicate with each other and other network endpoints. Network policy resources use labels to select pods and define whitelist rules that allow traffic to the selected pods in addition to what is allowed by the isolation policy for a given namespace.

Network policies and CNI plugins

There is an intricate relationship between network policies and CNI plugins. Some CNI plugins implement both network connectivity and network policy, while others implement just one aspect, but they can collaborate with another CNI plugin that implements the other aspect (for example, Calico and Flannel).

Configuring network policies

Network policies are configured via the `NetworkPolicy` resource. Here is a sample network policy:

```
apiVersion: extensions/v1beta1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: awesome-project
        - podSelector:
            matchLabels:
              role: frontend
  ports:
    - protocol: tcp
      port: 6379
```

Implementing network policies

While the network policy API itself is generic and is part of the Kubernetes API, the implementation is tightly coupled to the networking solution. That means that on each node, there is a special agent or gatekeeper that does the following:

1. Intercepts all traffic coming into the node
2. Verifies that it adheres to the network policy
3. Forwards or rejects each request

Kubernetes provides the facilities to define and store network policies through the API. Enforcing the network policy is left to the networking solution or a dedicated network policy solution that is tightly integrated with the specific networking solution. Calico and Canal are good examples of this approach. Calico has its own networking solution and a network policy solution that works together. However, it can also provide network policy enforcement on top of Flannel as part of Canal. In both cases, there is tight integration between the two pieces. The following diagram shows how the Kubernetes policy controller manages the network policies and how agents on the nodes execute it:

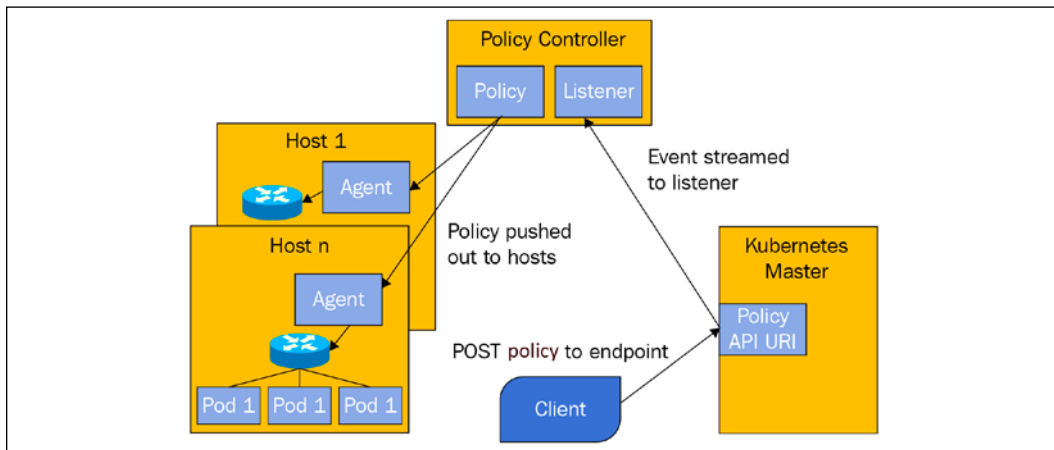


Figure 10.10: Kubernetes policy controller

Load balancing options

Load balancing is a critical capability in dynamic systems, such as a Kubernetes cluster. Nodes, VMs, and pods come and go, but the clients typically can't keep track of which individual entities can service their requests. Even if they could, it requires a complicated dance of managing a dynamic map of the cluster, refreshing it frequently, and handling disconnected, unresponsive, or just slow nodes.

This so-called client-side load balancing is appropriate in special cases only. Server-side load balancing is a battle-tested and well-understood mechanism that adds a layer of indirection that hides the internal turmoil from the clients or consumers outside the cluster. There are options for external as well as internal load balancers. You can also mix and match and use both. The hybrid approach has its own particular pros and cons, such as performance versus flexibility.

External load balancer

An external load balancer is a load balancer that runs outside the Kubernetes cluster. There must be an external load balancer provider that Kubernetes can interact with to configure the external load balancer with health checks, firewall rules, and to get the external IP address of the load balancer.

The following diagram shows the connection between the load balancer (in the cloud), the Kubernetes API server, and the cluster nodes. The external load balancer has an up-to-date picture of which pods run on which nodes and it can direct external service traffic to the right pods:

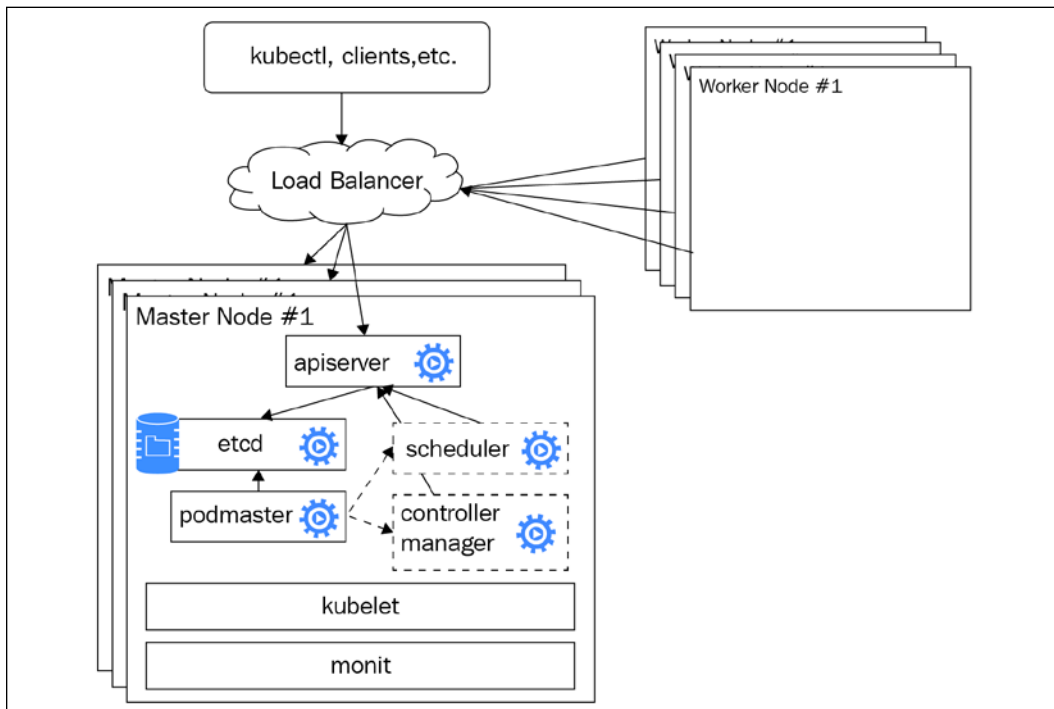


Figure 10.11: Cloud-based load balancer

Configuring an external load balancer

The external load balancer is configured via the service configuration file, or directly through Kubectl. We use a service type of load balancer instead of using a service type of ClusterIP, which directly exposes a Kubernetes node as a load balancer. This depends on an external load balancer provider being properly installed and configured in the cluster. Google's GKE is the most well-tested provider, but other cloud platforms provide their integrated solution on top of their cloud load balancer.

Via a configuration file

Here is an example service configuration file that accomplishes this goal:

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
spec:
  type: LoadBalancer
  ports:
  - port: 80
    targetPort: 5000
  selector:
    svc: api-gateway
    app: delinkcious
```

Via Kubectl

You may also accomplish the same result using a direct kubectl command:

```
$ kubectl expose deployment api-gateway --port=80 --target-port=5000
--name=api-gateway --type=LoadBalancer
```

The decision whether to use a service configuration file or kubectl command is usually determined by the way you set up the rest of your infrastructure and deploy your system. Configuration files are more declarative and arguably more appropriate for production usage where you want a versioned, auditable, and repeatable way to manage your infrastructure.

Finding the load balancer IP addresses

The load balancer will have two IP addresses of interest. The internal IP address can be used inside the cluster to access the service. Clients outside the cluster will use the external IP address.

It's a good practice to create a DNS entry for the external IP address. It is particularly important if you want to use TLS/SSL, which require stable hostnames. To get both addresses, use the `kubectl describe service` command. The `IP` field denotes the internal IP address, and the `LoadBalancer Ingress` field denotes the external IP address:

```
$ kubectl describe services example-service
Name: example-service
Selector: app=example
Type: LoadBalancer
IP: 10.67.252.103
LoadBalancer Ingress: 123.45.678.9
Port: <unnamed> 80/TCP
NodePort: <unnamed> 32445/TCP
Endpoints: 10.64.0.4:80,10.64.1.5:80,10.64.2.4:80
Session Affinity: None
No events.
```

Preserving client IP addresses

Sometimes, the service may be interested in the source IP address of the clients. Up until Kubernetes 1.5, this information wasn't available. In Kubernetes 1.5, there is a beta feature available only on GKE through an annotation to get the source IP address. In Kubernetes 1.7, the capability to preserve the original client IP was added to the API.

Specifying original client IP address preservation

You need to configure the following two fields of the service spec:

- `service.spec.externalTrafficPolicy`: This field determines whether the service should route external traffic to a node-local endpoint or a cluster-wide endpoint, which is the default. The `Cluster` option doesn't reveal the client source IP and might add a hop to a different node, but spreads the load well. The `Local` option keeps the client source IP and doesn't add an extra hop as long as the service type is `LoadBalancer` or `NodePort`. Its downside is that it might not balance the load very well.
- `service.spec.healthCheckNodePort`: This field is optional. If used, then the service health check will use this port number. The default is the allocated node port. This has an effect on services of the `LoadBalancer` type whose `externalTrafficPolicy` is set to `Local`.

Here is an example:

```
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
spec:
  type: LoadBalancer
  externalTrafficPolicy: Local
  ports:
    - port: 80
      targetPort: 5000
  selector:
    svc: api-gateway
    app: delinkcious
```

Understanding even external load balancing

External load balancers operate at the node level; while they direct traffic to a particular pod, the load distribution is done at the node level. This means that if your service has four pods, and three of them are on node A and the last one is on node B, then an external load balancer is likely to divide the load evenly between node A and node B. This will have the three pods on node A handle half of the load (1/6 each) and the single pod on node B handle the other half of the load on its own. Weights may be added in the future to address this issue.

Service load balancing

Service load balancing is designed for funneling internal traffic within the Kubernetes cluster and not for external load balancing. This is done by using a service type of ClusterIP. It is possible to expose a service load balancer directly via a preallocated port by using a service type of NodePort and utilizing it as an external load balancer, but it wasn't designed for that use case. Desirable features such as SSL termination and HTTP caching will not be readily available.

The following diagram shows how the service load balancer (the yellow clouds) can route traffic to one of the backend pods it manages (via labels of course):

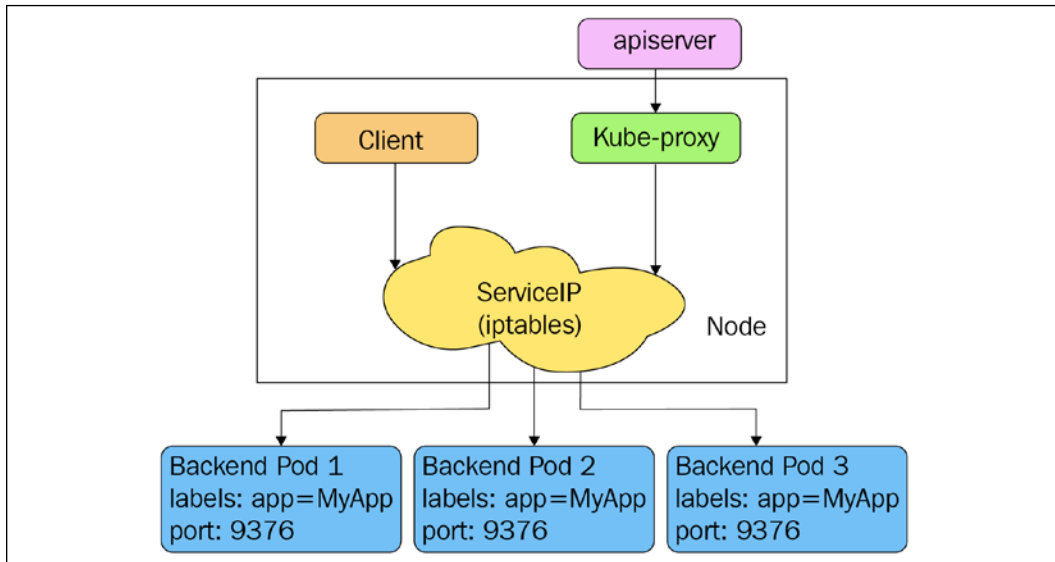


Figure 10.12: Service load balancer routing traffic

Ingress

Ingress in Kubernetes is, at its core, a set of rules that allow inbound connections to reach cluster services. In addition, some ingress controllers support the following:

- Connection algorithms
- Request limits
- URL rewrites and redirects
- TCP/UDP load balancing
- SSL termination
- Access control and authorization

Ingress is specified using an Ingress resource and serviced by an ingress controller. It's important to note that ingress is still in beta (since Kubernetes 1.1) and it doesn't yet demonstrate all the necessary capabilities. Here is an example of an ingress resource that manages traffic into two services.

The rules map the externally visible `http://foo.bar.com/foo` to the `s1` service, and `http://foo.bar.com/bar` to the `s2` service:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: s1
          servicePort: 80
      - path: /bar
        backend:
          serviceName: s2
          servicePort: 80
```

There are two official ingress controllers right now. One of them is an L7 ingress controller for GCE only, while the other is a more general-purpose Nginx ingress controller that lets you configure the Nginx web server through a ConfigMap. The NGINX ingress controller is very sophisticated and brings to bear a lot of features that are not available yet through the ingress resource directly. It uses the endpoint's API to directly forward traffic to pods. It supports Minikube, GCE, AWS, Azure, and bare metal clusters. For more details, check out the following link:

<https://github.com/kubernetes/ingress-nginx>

However, there are many more ingress controllers that may be better for your use case, such as:

- Ambassador
- Traefik
- Contour
- Gloo

HAProxy

We discussed using a cloud provider external load balancer using the load balancer service type and using the internal service load balancer inside the cluster using ClusterIP. If we want a custom external load balancer, we can create a custom external load balancer provider and use LoadBalancer or the third service type, NodePort. **High-Availability Proxy (HAProxy)** is a mature and battle-tested load-balancing solution. It is considered one of the best choices for implementing external load balancing with on-premises clusters. This can be done in several ways:

- Utilize NodePort and carefully manage port allocations
- Implement the custom load balancer provider interface
- Run HAProxy inside your cluster as the only target of your frontend servers at the edge of the cluster (load balanced or not)

You can use all these approaches with HAProxy. Regardless, it is still recommended to use ingress objects. The `service-loadbalancer` project is a community project that implemented a load-balancing solution on top of HAProxy. You can find it here: <https://github.com/kubernetes/contrib/tree/master/service-loadbalancer>.

Utilizing the NodePort

Each service will be allocated a dedicated port from a predefined range. This usually is a high range, such as 30,000 and upward, so as to avoid clashing with other applications using low known ports. HAProxy will run outside the cluster in this case and it will be configured with the correct port for each service. Then it can just forward any traffic to any nodes and Kubernetes via the internal service, and the load balancer will route it to a proper pod (double load balancing). This is, of course, sub-optimal because it introduces another hop. The way to circumvent it is to query the endpoint's API and dynamically manage for each service the list of its backend pods and directly forward traffic to the pods.

Custom load balancer provider using HAProxy

This approach is a little more complicated, but the benefit is that it is better integrated with Kubernetes and can make the transition to/from on-premises from/to the cloud easier.

Running HAProxy inside the Kubernetes cluster

In this approach, we use the internal HAProxy load balancer inside the cluster. There may be multiple nodes running HAProxy and they will share the same configuration to map incoming requests and load-balance them across the backend servers (the Apache servers in the following diagram):

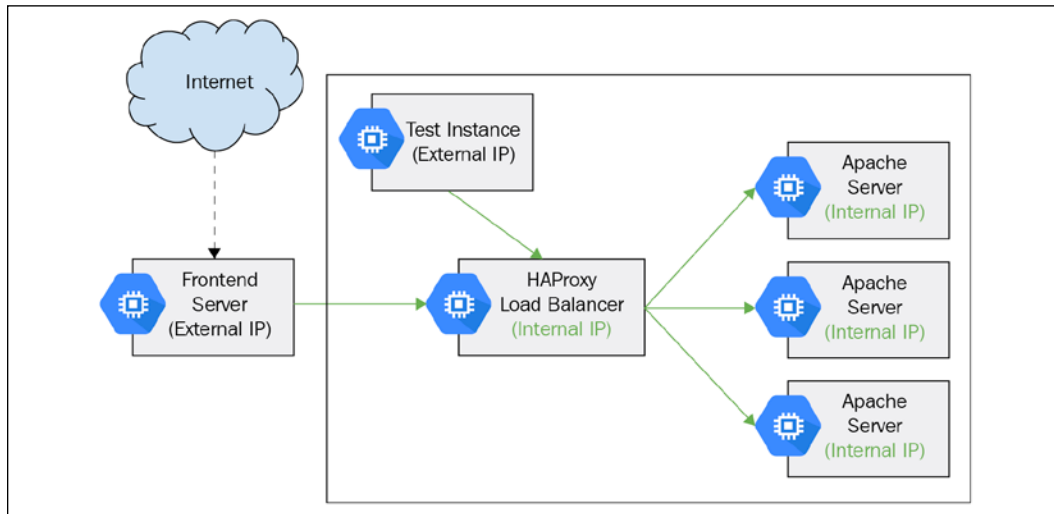


Figure 10.13: HAProxy load balancing

HAProxy also developed its own ingress controller, which is Kubernetes aware. This is arguably the most streamlined way to utilize HAProxy in your Kubernetes cluster. Here are some of the capabilities you acquire when using the HAProxy ingress controller:

- Streamlined integration with the HAProxy load balancer
- SSL termination
- Rate limiting
- IP whitelisting
- Multiple load balancing algorithms: round-robin, least connections, URL hash, and random
- A dashboard that shows the health of your pods, current request rates, response times, and so on
- Traffic overload protection

MetalLB

MetalLB also provides a load balancer solution for bare metal clusters. It is highly configurable and supports multiple modes such as L2 and BGP. I had success configuring it even for minikube.

For more details, check out the following link:

<https://metallb.universe.tf>

Keepalived VIP

Keepalived Virtual IP (Keepalived VIP) is not necessarily a load-balancing solution of its own.

It can be a complement to the Nginx ingress controller or the HAProxy-based service LoadBalancer. The main motivation is that pods move around in Kubernetes, including your load balancer(s). That creates a problem for clients outside the network that require a stable endpoint. DNS is often not good enough due to performance issues. Keepalived provides a high-performance virtual IP address that can serve as the address to the Nginx ingress controller or the HAProxy load balancer. Keepalived utilizes core Linux networking facilities, such as IPVS (IP virtual server) and implements high availability via **Virtual Redundancy Router Protocol (VRRP)**. Everything runs at layer 4 (TCP/UDP). It takes some effort and attention to detail to configure it. Luckily, there is a Kubernetes contrib project that can get you started: <https://github.com/kubernetes/contrib/tree/master/keepalived-vip>.

Traefic

Traefic is a modern HTTP reverse proxy and load balancer. It was designed to support microservices. It works with many backends, including Kubernetes, to manage its configuration automatically and dynamically. This is a game changer compared to traditional load balancers. It has an impressive list of features:

- It is fast
- Single Go executable
- Tiny official Docker image
- Rest API
- Hot-reloading of configuration; no need to restart the process

- Circuit breakers; retry
- Round-robin, rebalancer, load balancers
- Metrics (Rest, Prometheus, Datadog, Statsd, InfluxDB)
- Clean AngularJS web UI
- Websocket, HTTP/2, GRPC ready
- Access logs (JSON, CLF)
- Let's Encrypt support (automatic HTTPS with renewal)
- High availability with cluster mode

Load balancing on Kubernetes is an exciting area. It offers many options for both north-south and east-west load balancing. Now that we have covered load balancing in detail, let's dive deep into the CNI plugins and how they are implemented.

Writing your own CNI plugin

In this section, we will look at what it takes to actually write your own CNI plugin. First, we will look at the simplest plugin possible – the loopback plugin. Then, we will examine the plugin skeleton that implements most of the boilerplate associated with writing a CNI plugin. Finally, we will review the implementation of the bridge plugin. Before we dive in, here is a quick reminder of what a CNI plugin is:

- A CNI plugin is an executable
- It is responsible for connecting new containers to the network, assigning unique IP addresses to CNI containers, and taking care of routing
- A container is a network namespace (in Kubernetes, a pod is a CNI container)
- Network definitions are managed as JSON files, but stream to the plugin via standard input (no files are being read by the plugin)
- Auxiliary information can be provided via environment variables

First look at the loopback plugin

The loopback plugin simply adds the loopback interface. It is so simple that it doesn't require any network configuration information. Most CNI plugins are implemented in Golang and the loopback CNI plugin is no exception. The full source code is available here: <https://github.com/containernetworking/plugins/blob/master/plugins/main/loopback>.

There are multiple packages from the container networking project on GitHub that provide many of the building blocks necessary to implement CNI plugins and the netlink package for adding interfaces, removing interfaces, setting IP addresses, and setting routes. Let's look at the imports of the `loopback.go` file first:

```
package main

import (
    "encoding/json"
    "errors"
    "fmt"
    "net"

    "github.com/vishvananda/netlink"

    "github.com/container networking/cni/pkg/skel"
    "github.com/container networking/cni/pkg/types"
    "github.com/container networking/cni/pkg/types/current"
    "github.com/container networking/cni/pkg/version"

    "github.com/container networking/plugins/pkg/ns"
    bv "github.com/container networking/plugins/pkg/utils/buildversion")
```

Then, the plugin implements two commands, `cmdAdd` and `cmdDel`, which are called when a container is added to or removed from the network. Here is the `add` command, which does all the heavy lifting:

```
func cmdAdd(args *skel.CmdArgs) error {
    conf, err := parseNetConf(args.StdinData)
    if err != nil {
        return err
    }

    var v4Addr, v6Addr *net.IPNet

    args.IfName = "lo" // ignore config, this only works for Loopback
    err = ns.WithNetNSPath(args.Netns, func(_ ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
        if err != nil {
            return err // not tested
        }

        err = netlink.LinkSetUp(link)
```

```
    if err != nil {
        return err // not tested
    }
    v4Addrs, err := netlink.AddrList(link, netlink.FAMILY_V4)
    if err != nil {
        return err // not tested
    }
    if len(v4Addrs) != 0 {
        v4Addr = v4Addrs[0].IPNet
        // sanity check that this is a loopback address
        for _, addr := range v4Addrs {
            if !addr.IP.IsLoopback() {
                return fmt.Errorf("loopback interface found with
non-loopback address %q", addr.IP)
            }
        }
    }

    v6Addrs, err := netlink.AddrList(link, netlink.FAMILY_V6)
    if err != nil {
        return err // not tested
    }
    if len(v6Addrs) != 0 {
        v6Addr = v6Addrs[0].IPNet
        // sanity check that this is a loopback address
        for _, addr := range v6Addrs {
            if !addr.IP.IsLoopback() {
                return fmt.Errorf("loopback interface found with
non-loopback address %q", addr.IP)
            }
        }
    }

    return nil
})
if err != nil {
    return err // not tested
}

var result types.Result
if conf.PrevResult != nil {
    // If loopback has previous result which passes from previous
```

```

CNI plugin,
    // loopback should pass it transparently
    result = conf.PrevResult
} else {
    loopbackInterface := &current.Interface{Name: args.IfName, Mac:
"00:00:00:00:00:00", Sandbox: args.Netns}
    r := &current.Result{CNIVersion: conf.CNIVersion, Interfaces:
[]*current.Interface{loopbackInterface}}

    if v4Addr != nil {
        r.IPs = append(r.IPs, &current.IPConfig{
            Version: "4",
            Interface: current.Int(0),
            Address: *v4Addr,
        })
    }

    if v6Addr != nil {
        r.IPs = append(r.IPs, &current.IPConfig{
            Version: "6",
            Interface: current.Int(0),
            Address: *v6Addr,
        })
    }

    result = r
}

return types.PrintResult(result, conf.CNIVersion)
}

```

The core of this function is setting the interface name to `lo` (for loopback) and adding the link to the container's network namespace. It supports both IPv4 and IPv6.

The `del` command does the opposite and is much simpler:

```

func cmdDel(args *skel.CmdArgs) error {
    if args.Netns == "" {
        return nil
    }
    args.IfName = "lo" // ignore config, this only works for loopback
    err := ns.WithNetNSPath(args.Netns, func(ns.NetNS) error {
        link, err := netlink.LinkByName(args.IfName)
    })
    return err
}

```

```
        if err != nil {
            return err // not tested
        }

        err = netlink.LinkSetDown(link)
        if err != nil {
            return err // not tested
        }

        return nil
    })
    if err != nil {
        return err // not tested
    }

    return nil
}
```

The main function simply calls the `PluginMain()` function of the `skel` package, passing the command functions. The `skel` package will take care of running the CNI plugin executable and will invoke the `cmdAdd` and `cmdDel` functions at the right time:

```
func main() {
    skel.PluginMain(cmdAdd, cmdCheck, cmdDel, version.All,
        bv.BuildString("loopback"))
}
```

Building on the CNI plugin skeleton

Let's now explore the `skel` package and see what it does under the covers. The `PluginMain()` entry point is responsible for invoking `PluginMainWithError()`, catching errors, printing them to standard output, and exiting:

```
func PluginMain(cmdAdd, cmdCheck, cmdDel func(_ *CmdArgs) error,
    versionInfo version.PluginInfo, about string) {
    if e := PluginMainWithError(cmdAdd, cmdCheck, cmdDel, versionInfo,
        about); e != nil {
        if err := e.Print(); err != nil {
            log.Print("Error writing error JSON to stdout: ", err)
        }
        os.Exit(1)
    }
}
```

The `PluginErrorWithMain()` function instantiates a dispatcher, sets it up with all the I/O streams and the environment, and invokes its internal `pluginMain()` method:

```
func PluginMainWithError(cmdAdd, cmdCheck, cmdDel func(_ *CmdArgs)
error, versionInfo version.PluginInfo, about string) *types.Error {
    return (&dispatcher{
        Getenv: os.Getenv,
        Stdin:  os.Stdin,
        Stdout: os.Stdout,
        Stderr: os.Stderr,
    }).pluginMain(cmdAdd, cmdCheck, cmdDel, versionInfo, about)
}
```

Here is, finally, the main logic of the skeleton. It gets the `cmd` arguments from the environment (which includes the configuration from standard input), detects which `cmd` is invoked, and calls the appropriate plugin function (`cmdAdd` or `cmdDel`). It can also return version information:

```
func (t *dispatcher) pluginMain(cmdAdd, cmdCheck, cmdDel func(_
*CmdArgs) error, versionInfo version.PluginInfo, about string) *types.
Error {
    cmd, cmdArgs, err := t.getCmdArgsFromEnv()
    if err != nil {
        // Print the about string to stderr when no command is set
        if err.Code == types.ErrInvalidEnvironmentVariables &&
t.Getenv("CNI_COMMAND") == "" && about != "" {
            _, _ = fmt.Fprintln(t.Stderr, about)
            return nil
        }
        return err
    }

    if cmd != "VERSION" {
        if err = validateConfig(cmdArgs.StdinData); err != nil {
            return err
        }
        if err = utils.ValidateContainerID(cmdArgs.ContainerID); err !=
nil {
            return err
        }
        if err = utils.ValidateInterfaceName(cmdArgs.IfName); err !=
nil {
            return err
        }
    }
}
```

```
    }
}

switch cmd {
case "ADD":
    err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdAdd)
case "CHECK":
    configVersion, err := t.ConfVersionDecoder.Decode(cmdArgs.
StdinData)
    if err != nil {
        return types.NewError(types.ErrDecodingFailure, err.
Error(), "")
    }
    if gtet, err := version.GreaterThanOrEqualTo(configVersion,
"0.4.0"); err != nil {
        return types.NewError(types.ErrDecodingFailure, err.
Error(), "")
    } else if !gtet {
        return types.NewError(types.ErrIncompatibleCNIVersion,
"config version does not allow CHECK", "")
    }
    for _, pluginVersion := range versionInfo.SupportedVersions() {
        gtet, err := version.GreaterThanOrEqualTo(pluginVersion,
configVersion)
        if err != nil {
            return types.NewError(types.ErrDecodingFailure, err.
Error(), "")
        } else if gtet {
            if err := t.checkVersionAndCall(cmdArgs, versionInfo,
cmdCheck); err != nil {
                return err
            }
            return nil
        }
    }
    return types.NewError(types.ErrIncompatibleCNIVersion, "plugin
version does not allow CHECK", "")
case "DEL":
    err = t.checkVersionAndCall(cmdArgs, versionInfo, cmdDel)
case "VERSION":
    if err := versionInfo.Encode(t.Stdout); err != nil {
        return types.NewError(types.ErrIOFailure, err.Error(), "")
    }
}
```

```

    default:
        return types.NewError(types.ErrInvalidEnvironmentVariables,
            fmt.Sprintf("unknown CNI_COMMAND: %v", cmd), "")
    }

    if err != nil {
        return err
    }

    return nil
}

```

The loopback plugin is one of the simplest CNI plugins. Now, let's check out the bridge plugin.

Reviewing the bridge plugin

The bridge plugin is more substantial. Let's look at some of the key parts of its implementation. The full source code is available here:

<https://github.com/containernetworking/plugins/tree/master/plugins/main/bridge>

The plugin defines a network configuration struct with the following fields in the `bridge.go` file:

```

type NetConf struct {
    types.NetConf
    BrName      string 'json:"bridge"'
    IsGW        bool   'json:"isGateway"'
    IsDefaultGW bool   'json:"isDefaultGateway"'
    ForceAddress bool   'json:"forceAddress"'
    IPMasq      bool   'json:"ipMasq"'
    MTU         int    'json:"mtu"'
    HairpinMode bool   'json:"hairpinMode"'
    PromiscMode bool   'json:"promiscMode"'
    Vlan        int    'json:"vlan"'
}

```

We will not cover what each parameter does and how it interacts with the other parameters due to space limitations. The goal is to understand the flow and have a starting point if you want to implement your own CNI plugin. The configuration is loaded from JSON via the `loadNetConf()` function.

It is called at the beginning of the `cmdAdd()` and `cmdDel()` functions:

```
n, cniVersion, err := loadNetConf(args.StdinData)
```

Here is the core of the `cmdAdd()` function, which uses information from network configuration, sets up the bridge, and sets up a veth device:

```
br, brInterface, err := setupBridge(n)
if err != nil {
    return err
}

netns, err := ns.GetNS(args.Netns)
if err != nil {
    return fmt.Errorf("failed to open netns %q: %v", args.Netns,
err)
}
defer netns.Close()

hostInterface, containerInterface, err := setupVeth(netns, br,
args.IfName, n.MTU, n.HairpinMode, n.Vlan)
if err != nil {
    return err
}
```

Later, the function handles the L3 mode with its multiple cases:

```
// Assume L2 interface only
result := &current.Result{CNIVersion: cniVersion, Interfaces:
[]*current.Interface{brInterface, hostInterface, containerInterface}}
if isLayer3 {
    // run the IPAM plugin and get back the config to apply
    r, err := ipam.ExecAdd(n.IPAM.Type, args.StdinData)
    if err != nil {
        return err
    }

    // release IP in case of failure
    defer func() {
        if !success {
            ipam.ExecDel(n.IPAM.Type, args.StdinData)
        }
    }()
}
```

```

type      // Convert whatever the IPAM result was into the current Result

    ipamResult, err := current.NewResultFromResult(r)
    if err != nil {
        return err
    }

    result.IPs = ipamResult.IPs
    result.Routes = ipamResult.Routes

    if len(result.IPs) == 0 {
        return errors.New("IPAM plugin returned missing IP config")
    }

    // Gather gateway information for each IP family
    gwsV4, gwsV6, err := calcGateways(result, n)
    if err != nil {
        return err
    }

    // Configure the container hardware address and IP address(es)
    if err := netns.Do(func(_ ns.NetNS) error {
        ...
    })

    if n.IsGW {
        ...
    }

    if n.IPMasq {
        ...
    }
}

```

Finally, it updates the MAC address that may have changed and returns the results:

```

    // Refetch the bridge since its MAC address may change when the
    first
    // veth is added or after its IP address is set
    br, err = bridgeByName(n.BrName)
    if err != nil {
        return err
    }

```

```
}
brInterface.Mac = br.Attrs().HardwareAddr.String()

result.DNS = n.DNS

// Return an error requested by testcases, if any
if debugPostIPAMError != nil {
    return debugPostIPAMError
}

success = true

return types.PrintResult(result, cniVersion)
```

This is just part of the full implementation. There is also route setting and hardware IP allocation. If you plan to write your own CNI plugin, I encourage you to pursue the full source code, which is quite extensive, to get the full picture.

Summary

In this chapter, we covered a lot of ground. Networking is such a vast topic as there are so many combinations of hardware, software, operating environments, and user skills. It is a very complicated endeavor to come up with a comprehensive networking solution that is both robust, secure, performs well, and is easy to maintain. For Kubernetes clusters, the cloud providers mostly solve these issues. But if you run on-premises clusters or need a tailor-made solution, you get a lot of options to choose from. Kubernetes is a very flexible platform, designed for extension. Networking in particular is totally pluggable. The main topics we discussed were the Kubernetes networking model (flat address space where pods can reach other and share localhost between all containers inside a pod), how lookup and discovery work, the Kubernetes network plugins, various networking solutions at different levels of abstraction (a lot of interesting variations), using network policies effectively to control the traffic inside the cluster, and the spectrum of load-balancing solutions, and finally, we looked at how to write a CNI plugin by dissecting a real-world implementation.

At this point, you are probably overwhelmed, especially if you're not a subject matter expert. Nonetheless, you should have a solid grasp of the internals of Kubernetes networking, be aware of all the interlocking pieces required to implement a fully fledged solution, and be able to craft your own solution based on trade-offs that make sense for your system and your skill level.

In *Chapter 11, Running Kubernetes on Multiple Clouds and Cluster Federation*, we will go even bigger and look at running Kubernetes on multiple clusters, cloud providers, and federation. This is an important part of the Kubernetes story for geo-distributed deployments and ultimate scalability. Federated Kubernetes clusters can exceed local limitations, but they bring a whole slew of challenges as well.

11

Running Kubernetes on Multiple Clouds and Cluster Federation

In this chapter, we'll take it to the next level by running Kubernetes on multiple clouds, multiple clusters, and cluster federation. A Kubernetes cluster is a closely knit unit where all of the components run in relative proximity and are connected by a fast network (typically, a physical data center or cloud provider availability zone). This is great for many use cases, but there are several important use cases where systems need to scale beyond a single cluster. One approach to address this use case is with Kubernetes federation, which is a methodical way to combine multiple Kubernetes clusters and interact with them as a single entity. Another approach taken by the Gardener (<https://gardener.cloud/>) project is to provide an abstraction around managing multiple separate Kubernetes clusters.

We will cover the following topics:

- The history of cluster federation on Kubernetes
- Understanding cluster federation
- A deep dive into what cluster federation is all about
- How to prepare, configure, and manage a cluster federation
- How to run a federated workload across multiple clusters
- Introduction to the Gardener project

The history of cluster federation on Kubernetes

Before jumping into the details of cluster federation, let's get some historical context. It's funny to talk about the history of a project like Kubernetes that didn't even exist 5 years ago, but the pace of development and a large number of contributors took Kubernetes through accelerated evolution. This is especially relevant for Kubernetes federation.

In March 2015, the first revision of the Kubernetes Cluster Federation (<https://github.com/kubernetes/community/blob/master/contributors/design-proposals/multicloud/federation.md>) proposal was published. Back then, it was fondly nicknamed "Ubernetes." The basic idea was to reuse the existing Kubernetes APIs to manage multiple clusters. This proposal, now called Federation V1, went through several rounds of revision and implementation, but it never reached general availability and is considered deprecated at this point. The SIG cluster workgroup realized that the multi-cluster problem was more complicated than initially perceived. There are many ways to skin this particular cat, and there is no one-size-fits-all solution.

The new direction for cluster federation is dedicated APIs for federation. In the rest of the chapter, we will consider the Federation V2 design. Note that the current status is considered Alpha, so I don't recommend putting it to use in production without significant consideration.

Understanding cluster federation

Cluster federation is conceptually simple. You aggregate multiple Kubernetes clusters and treat them as a single logical cluster. There is a federation control plane that presents to clients a single unified view of the system.

The following diagram demonstrates the big picture of the Kubernetes Cluster Federation. The pink box is a host cluster that runs the federation APIs and the green boxes are member clusters:

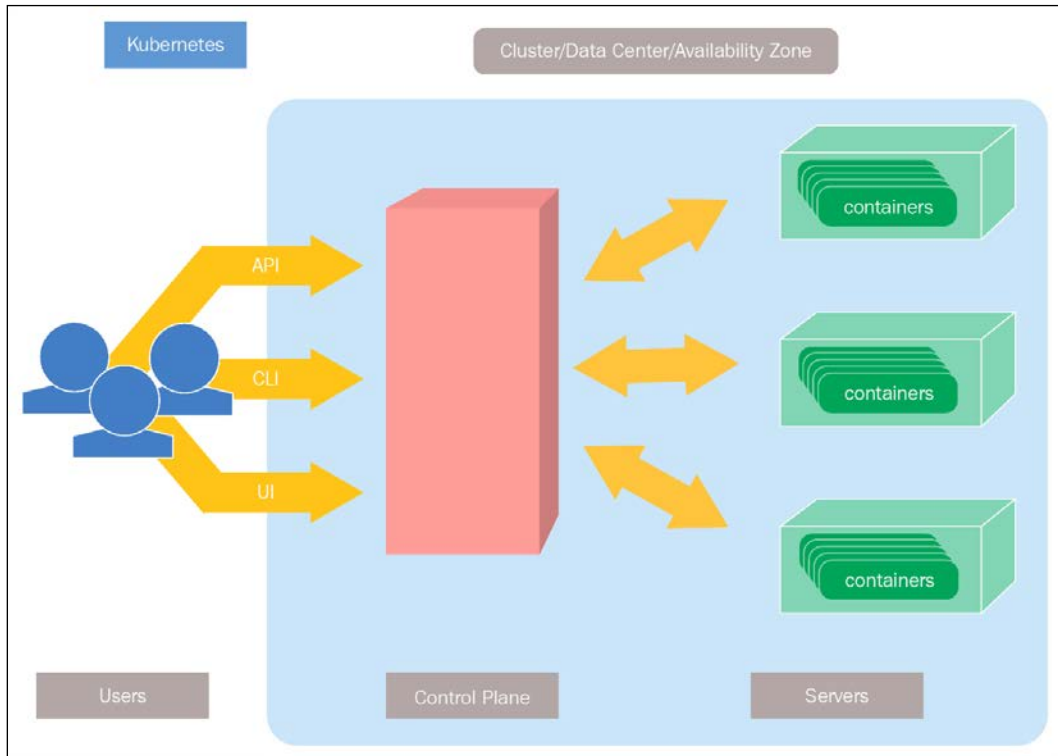


Figure 11.1: The Kubernetes Cluster Federation

The federation control plane consists of a federation API server and a federation controller manager that collaborate with each other. The federated API server forwards requests to all the clusters in the federation. In addition, the federated controller manager performs the duties of the controller manager across all of the clusters by routing requests to the individual federation cluster members' changes. In practice, cluster federation is not trivial and can't be totally abstracted away. Cross-pod communication and data transfer may suddenly incur a massive latency and cost overhead. Let's look at the use cases for cluster federation first, understand how the federated components and resources work, and then examine the hard parts: location affinity, cross-cluster scheduling, and federated data access.

Important use cases for cluster federation

There are four categories of use cases that benefit from cluster federation.

Capacity overflow

Public cloud platforms such as AWS, GCE, and Azure are great and provide many benefits, but they are not cheap. Many large organizations have invested a lot in their own data centers. Other organizations work with private service providers such as OVS, Rackspace, or Digital Ocean. If you have the operational capacity to manage and operate infrastructure on your own, it makes a lot of economic sense to run your Kubernetes cluster on your infrastructure rather than in the cloud. But what if some of your workloads fluctuate and, for a relatively short amount of time, require a lot more capacity?

For example, your system may be hit especially hard on the weekends or maybe during holidays. The traditional approach is to just provision extra capacity. But in many dynamic situations, this is not easy. With capacity overflow, you can run the bulk of your work in a Kubernetes cluster running on an on-premises data center or with a private service provider and have a secondary cloud-based Kubernetes cluster running on one of the big platform providers. Most of the time, the cloud-based cluster will be shut down (stopped instances), but when the need arises, you can elastically add capacity to your system by starting some stopped instances. Kubernetes Cluster Federation can make this configuration relatively straightforward. It eliminates a lot of headaches about capacity planning and paying for hardware that's not used most of the time.

This approach is sometimes called "cloud bursting."

Sensitive workloads

This is almost the opposite of capacity overflow. Maybe you've embraced the cloud-native lifestyle, and your entire system runs on the cloud, but some data or workloads deal with sensitive information. Regulatory compliance or your organization's security policies may dictate that those data and workloads must run in an environment that's fully controlled by you. Your sensitive data and workloads may be subject to external auditing. It may be critical to ensure that no information ever leaks from the private Kubernetes cluster to the cloud-based Kubernetes cluster. But it may be desirable to have visibility in the public cluster and be able to launch non-sensitive workloads from the private cluster to the cloud-based cluster. If the nature of a workload can change dynamically from non-sensitive to sensitive, then it needs to be addressed by coming up with a proper policy and process of implementation.

For example, you may prevent workloads from changing their nature. Alternatively, you may migrate a workload that has suddenly become sensitive and ensure that it doesn't run on the cloud-based cluster anymore. Another important instance is national compliance, where certain data is required by law to remain and be accessed only from a designated geographical region (typically, a country). In this case, a cluster must be created in that geographical region.

Avoiding vendor lock-in

Large organizations often prefer to have options and not be tied to a single provider. The risk is often too great, because the provider may shut down or be unable to provide the same level of service. Having multiple providers is often good for negotiating prices, too. Kubernetes is designed to be vendor-agnostic. You can run it on different cloud platforms, private service providers, and on-premises data centers.

However, this is not trivial. If you want to be sure that you are able to switch providers quickly or shift some workloads from one provider to the next, you should already be running your system on multiple providers. You can do it yourself, or there are some companies that provide the service of running Kubernetes transparently on multiple providers. Since different providers run different data centers, you automatically get some redundancy and protection from vendor-wide outages.

Geo-distributing high availability

High availability means that a service will remain available to users even when some parts of the system fail. In the context of a federated Kubernetes cluster, the scope of failure is an entire cluster, which is typically due to problems with the physical data center hosting the cluster, or perhaps a wider issue with the platform provider. The key to high availability is redundancy. Geo-distributed redundancy means having multiple clusters running in different locations. It may be different availability zones of the same cloud provider, different regions of the same cloud provider, or even different cloud providers altogether (refer to the *Avoiding vendor lock-in* section). There are many issues to address when it comes to running a cluster federation with redundancy. We'll discuss some of these issues later. Assuming that the technical and organizational issues have been resolved, high availability will allow you to switch traffic from a failed cluster to another cluster. This should be transparent to the users up to a point (if there is a delay during the switchover, some in-flight requests or tasks may disappear or fail). The system administrators may need to take extra steps to support the switchover and to deal with the original cluster failure.

Now that we understand why multi-cluster federation is such an important aspect of Kubernetes, it's time to dive in.

Learning the basics of Kubernetes federation

Kubernetes federation is a complex topic, and we will ease our way into it. In this section, we will first look at some basic concepts, then at the building blocks of the API, and finally at its supported features and capabilities.

Defining basic concepts

Let's start our journey into the Kubernetes federation with some basic concepts and terminology. The following table describes the most important concepts and terms:

Concept	Description
Federate	Create a common interface to a pool of clusters in order to deploy Kubernetes applications across those clusters.
KubeFed	The API and control plane of Kubernetes Cluster Federation.
Host Cluster	A cluster that exposes the KubeFed API and runs the KubeFed control plane.
Cluster Registration	The process of adding a new cluster to the federation.
Member Cluster	A cluster that is registered with the KubeFed API and that KubeFed controllers have authentication credentials for. The Host Cluster can be a Member Cluster too.
ServiceDNSRecord	A resource that associates one or more Kubernetes Service resources with a scheme to construct DNS resource records for the Service.
IngressDNSRecord	A resource that associates one or more Kubernetes Ingress resources with a scheme to construct DNS resource records for the Ingress.
Endpoint	A resource that represents a Domain Name System (DNS) resource record.
DNSEndpoint	A Custom Resource wrapper for the Endpoint resource.

In addition to these concepts, the architecture is based on three building blocks.

Federation building blocks

A federation is responsible for a given set of API types (Kubernetes resources) that it manages and distributes into a set of member clusters. For each API type, there are common dedicated resources that the federation control plane uses to keep their state:

- **FederatedTemplate:** This stores the basic specification of the managed resource
- **FederatedPlacement:** This type holds the specification of the clusters that the resource should be distributed to
- **FederatedOverrides:** This optional resource can specify how the Template resource should behave on specific clusters

These types are all associated by name. For example, for the `ReplicaSet` resource, there are `FederatedReplicaSetTemplate`, `FederatedReplicaSetPlacement`, and `FederatedReplicaSetOverrides`.

In addition to this, the following elements can be used by higher-level APIs to customize and control the behavior of the federation:

- **Status:** Collects the status of resources distributed by KubeFed across all federated clusters
- **Policy:** Determines which subset of clusters a resource is allowed to be distributed to
- **Scheduling:** Decides how workloads should be distributed across different clusters

The following diagram illustrates the full life cycle and interaction of all the elements of the Kubernetes federation:

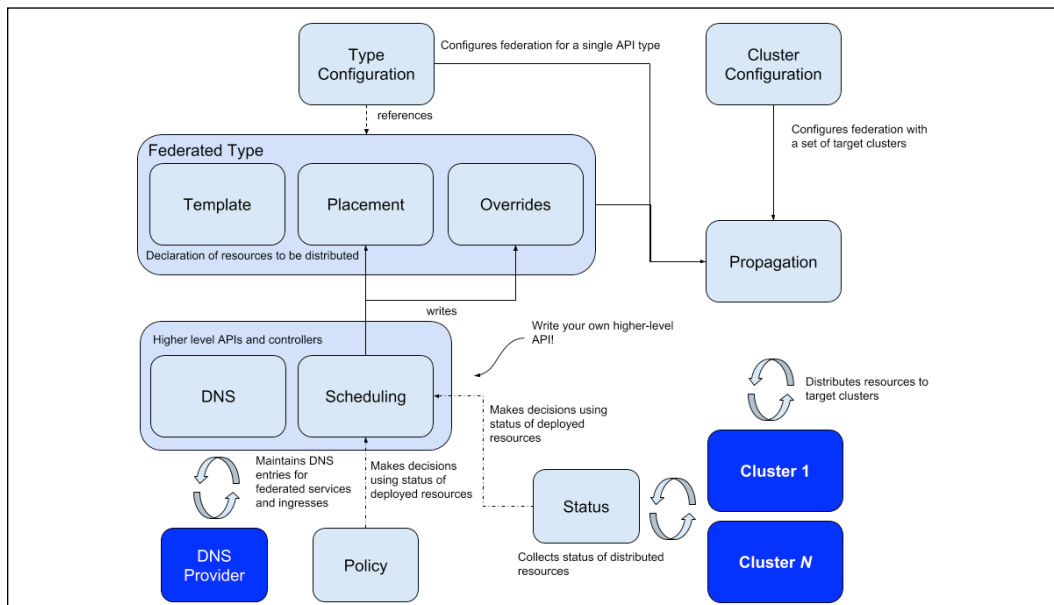


Figure 11.2: Life cycle of the Kubernetes federation

Federation features

These concepts and the foundational building blocks are used to implement the following features:

- Push propagation of arbitrary types to remote clusters
- A command-line interface program called `kubefedctl` to interact with the KubeFed API
- Generating KubeFed APIs without writing code
- Multi-cluster Service DNS via `external-dns`
- Multi-cluster Ingress DNS via `external-dns`
- Replica Scheduling Preferences

The KubeFed control plane

The KubeFed control plane requires Kubernetes 1.13 or later. It consists of two components that, together, enable a federation of Kubernetes clusters to appear and function as a single unified Kubernetes cluster.

The federation API server

The federation API server manages the Kubernetes clusters that, together, comprise the federation. It manages the federation state (which clusters are part of the federation) in an etcd database in the same way as a regular Kubernetes cluster, but the state it keeps is just those clusters that are members of the federation. The state of each cluster is stored in the etcd database of that cluster. The main purpose of the federation API server is to interact with the federation controller manager and route requests to the federation's member clusters. The federation members don't need to know that they are part of a federation: they just work the same.

The federation controller manager

The federation controller manager makes sure the federation's desired state matches the actual state. It forwards any necessary changes to the relevant cluster or clusters. The federated controller manager binary contains multiple controllers for all of the different federated resources that we'll cover later in the chapter. The control logic is similar, though: it observes changes and brings the cluster state to the desired state when they deviate. This is done for each member in the cluster federation.

The following diagram demonstrates this perpetual control loop:

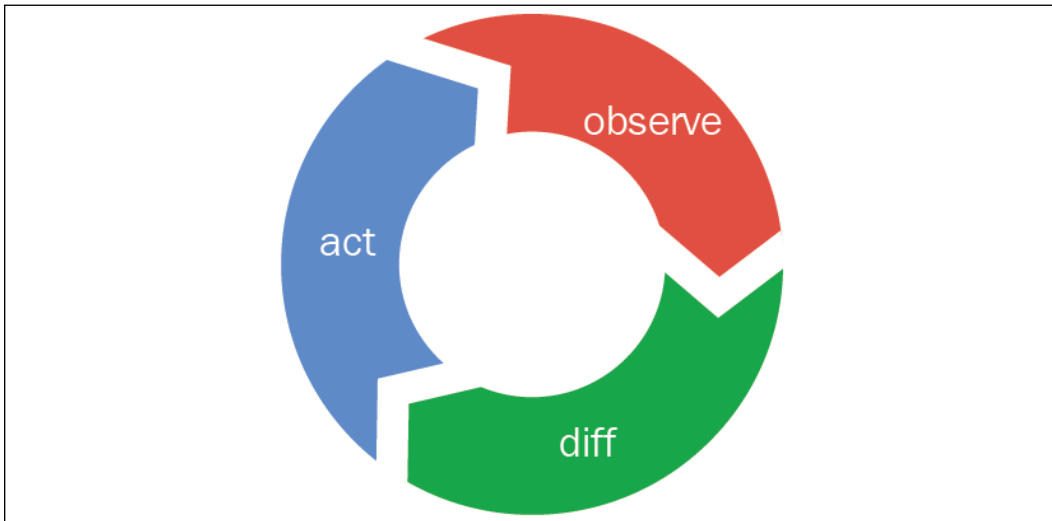


Figure 11.3: The federation controller manager's perpetual control loop

The hard parts

So far, the federation seems almost straightforward. KubeFed does the heavy lifting for you. You just group a bunch of clusters together, access them through the control plane, and everything just gets replicated to all of the clusters. But there are hard and difficult factors and basic concepts that complicate this simplified view. Much of the power of Kubernetes is derived from its ability to do a lot of work behind the scenes. Within a single cluster deployed fully in a single physical data center or availability zone where all the components are connected with a fast network, Kubernetes is very effective on its own. In a Kubernetes Cluster Federation, the situation is different. Latency, data transfer costs, and moving pods between clusters all have different trade-offs. Depending on the use case, making federation work may require extra attention, planning, and maintenance on the part of the system designers and operators. In addition, some of the federated resources are not as mature as their local counterparts, and that adds more uncertainty. Refer to <https://github.com/kubernetes-sigs/kubefed> for up-to-date information.

Federated unit of work

The unit of work in a Kubernetes cluster is the pod. You can't break a pod in Kubernetes. The entire pod will always be deployed together and be subject to the same life cycle treatment. Should the pod remain the unit of work for a cluster federation? Maybe it makes more sense to be able to associate a bigger unit, such as a whole ReplicaSet, deployment, or service with a specific cluster. If the cluster fails, the entire ReplicaSet, deployment, or service is scheduled to a different cluster. How about a collection of tightly coupled ReplicaSets? The answers to these questions are not always easy and may even change dynamically as the system evolves.

Location affinity

Location affinity is a major concern. When can pods be distributed across clusters? What are the relationships between those pods? Are there any requirements for affinity between pods or pods and other resources, such as storage? There are several major categories:

- Strictly coupled
- Loosely coupled
- Preferentially coupled
- Strictly decoupled
- Uniformly spread

When designing the system and how to allocate and schedule services and pods across the federation, it's important to make sure the location affinity requirements are always respected.

Strictly coupled

The strictly coupled requirement applies to applications where the pods must be in the same cluster. If you partition the pods, the application will fail (perhaps due to real-time requirements that can't be met when networking across clusters), or the cost may be too high (pods accessing a lot of local data). The only way to move such tightly coupled applications to another cluster is to start a complete copy (including data) on another cluster and then shut down the application on the current cluster. If the data is too large, the application may practically be immovable and sensitive to catastrophic failure. This is the most difficult situation to deal with, and, if possible, you should architect your system to avoid the strictly coupled requirement.

Loosely coupled

Loosely coupled applications are best when the workload is embarrassingly parallel, and each pod doesn't need to know about the other pods or access a lot of data. In these situations, pods can be scheduled to clusters just based on capacity and resource utilization across the federation. If necessary, pods can be moved from one cluster to another without problems. For example, consider a stateless validation service that performs a calculation and gets all of its input in the request itself and doesn't query or write any federation-wide data. It just validates its input and returns a valid/invalid verdict to the caller.

Preferentially coupled

Preferentially coupled applications perform better when all the pods are in the same cluster or the pods and the data are co-located, but this is not a hard requirement. For example, this could work with applications that require only eventual consistency, where some federation-wide application periodically synchronizes the application state across all clusters. In these cases, allocation is done explicitly to one cluster but leaves a safety hatch for running or migrating to other clusters under stress.

Strictly decoupled

Some services have fault isolation or high-availability requirements that force partitioning across clusters. There is no point running three replicas of a critical service if all replicas might end up scheduled to the same cluster, because that cluster just becomes an ad hoc SPOF.

Uniformly spread

Uniformly spread is when an instance of a service, ReplicaSet, or pod must run on each cluster. It is similar to DaemonSet, but instead of ensuring there is one instance on each node, it's one per cluster. A good example is a Redis cache backed up by some external persistent storage. The pods in each cluster should have their own cluster-local Redis cache to avoid accessing the central storage, which may be slower or become a bottleneck. On the other hand, there is no need for more than one Redis service per cluster (it could be distributed across several pods in the same cluster).

Cross-cluster scheduling

Cross-cluster scheduling goes hand in hand with location affinity. When a new pod is created or an existing pod fails and a replacement needs to be scheduled, where should it go? The current cluster federation doesn't handle all the scenarios and options for location affinity that we mentioned earlier.

At this point, cluster federation handles the loosely coupled (including weighted distribution) and strictly coupled (by making sure the number of replicas matches the number of clusters) categories well. Anything else will require that you don't use cluster federation. You'll have to add your own custom federation layer, which takes more specialized concerns into account and can accommodate more intricate scheduling use cases.

Federated data access

This is a tough problem. If you have a lot of data and pods running in multiple clusters (possibly on different continents) and need to access them quickly, then you have several unpleasant options:

- Replicate your data to each cluster (this is slow to replicate, expensive to transfer, expensive to store, and complicated to sync and deal with errors)
- Access the data remotely (this is slow to access, expensive on each access, and can be a SPOF)
- Use a sophisticated hybrid solution with per-cluster caching of some of the hottest data (this is complicated, results in stale data, and you still need to transfer a lot of data)

Federated auto-scaling

There is currently no support for federated auto-calling. There are two dimensions of scaling that can be utilized, as well as a combination of the two dimensions:

- Per-cluster scaling
- Adding/removing clusters from the federation
- Hybrid approach

Consider the relatively simple scenario of a loosely coupled application running on three clusters with five pods in each cluster. At some point, 15 pods can't handle the load anymore. We need to add more capacity. We can increase the number of pods per cluster, but if we do this at the federation level, then we will have six pods running in each cluster. We've increased the federation capacity by three pods when only one pod is needed. Of course, if you have more clusters, the problem gets worse. Another option is to pick a cluster and just change its capacity. This is possible with annotations, but now we're explicitly managing capacity across the federation. It can get complicated very quickly if we have lots of clusters running hundreds of services with dynamically changing requirements.

Adding a whole new cluster is even more complicated. Where should we add the new cluster? There is no requirement for extra availability that can guide the decision. It is just about extra capacity. Creating a new cluster also often requires a complicated first-time setup, where it may take days to approve various quotas on public cloud platforms. The hybrid approach increases the capacity of existing clusters in the federation until it reaches a threshold and then starts adding new clusters. The benefit of this approach is that when you're getting closer to the capacity limit of each cluster, you start preparing new clusters that will be ready to go when necessary. Other than that, it also requires a lot of effort, and you pay for the flexibility and scalability with increased complexity.

Managing a Kubernetes Cluster Federation

Managing a Kubernetes Cluster Federation involves many activities above and beyond managing a single cluster. You need to consider cascading resource deletion, load balancing across clusters, failover across clusters, federated service discovery, and federated discovery. Let's go over the various activities in detail. Note that due to the Alpha status of KubeFed, this should not be considered a step-by-step guide to follow. The goal here is to get a sense of what's involved in the management of multiple Kubernetes clusters as a federation.

Installing kubefedctl

The best way to interact with KubeFed is through the kubefedctl CLI. Here are the instructions to install the latest release of kubefedctl for macOS:

```
VERSION=0.3.0
OS=Darwin
ARCH=amd64
curl -LO https://github.com/kubernetes-sigs/kubefed/releases/download/
v${VERSION}/kubefedctl-${VERSION}-${OS}-${ARCH}.tgz
tar -zxvf kubefedctl-*.tgz
chmod u+x kubefedctl
sudo mv kubefedctl /usr/local/bin/
```

If typing kubefedctl is too much of a burden, then you can alias it like I did:

```
alias kf='kubefedctl'
```

To verify it's installed correctly, just run it and you will see the following:

```
$ kf
```

```
kubefedctl controls a Kubernetes Cluster Federation. Find more information
at https://sigs.k8s.io/kubefed.
```

Usage:

```
kubefedctl [flags]
kubefedctl [command]
```

Available Commands:

```
disable      Disables propagation of a Kubernetes API type
enable       Enables propagation of a Kubernetes API type
federate     Federate creates a federated resource from a
kubernetes resource
help         Help about any command
join         Register a cluster with a KubeFed control plane
orphaning-deletion Manage orphaning delete policy
unjoin       Remove the registration of a cluster from a KubeFed
control plane
version      Print the version info
```

Flags:

```
--alsologtostderr    log to standard error as well as
files
-h, --help           help for kubefedctl
--log-backtrace-at traceLocation when logging hits line file:N, emit
a stack trace (default :0)
--log-dir string     If non-empty, write log files in
this directory
--log-file string    If non-empty, use this log file
--log-flush-frequency duration Maximum number of seconds between
log flushes (default 5s)
--logtostderr        log to standard error instead of
files (default true)
--skip-headers       If true, avoid header prefixes in
the log messages
--stderrthreshold severity logs at or above this threshold go
to stderr
-v, --v Level        number for the log level verbosity
--vmodule moduleSpec comma-separated list of pattern=N
settings for file-filtered logging
```

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

The next step is to create some clusters that will form our federation.

Creating clusters

KubeFed officially supports four Kubernetes environments:

- KinD (Kubernetes in Docker)
- Minikube
- GKE
- IBM Cloud

The KinD environment is used by the KubeFed end-to-end tests. However, minikube is the easiest to set up for playing around with. Here are the instructions for creating two minikube clusters:

```
minikube start -p cluster-1
minikube start -p cluster-2
```

Then, in each cluster, verify all of the pods are running before moving forward.

Configuring the Host Cluster

OK. It's time to install the KubeFed control plane in your host cluster. The KubeFed project provides a convenient Helm chart for the task. Unfortunately, KubeFed doesn't support Helm 3 yet, because they use an outdated annotation (`crd-install` hook). You probably have Helm 3 installed, but you should install Helm 2 as well if you want to try using KubeFed. Since Helm 2 uses Tiller in the cluster, you need to create a service account for Tiller and give it administrator permissions so that it can install the KubeFed control plane securely:

```
$ kubectl config use-context cluster-1
Switched to context "cluster-1".
```

```
$ cat << EOF | kubectl apply -f -
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tiller
  namespace: kube-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
```

```
metadata:
  name: tiller
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
  - kind: ServiceAccount
    name: tiller
    namespace: kube-system
EOF
serviceaccount/tiller created
clusterrolebinding.rbac.authorization.k8s.io/tiller created

$ helm2 init --service-account tiller
```

Next, we need to add the KubeFed chart repository:

```
$ helm2 repo add kubefed-charts https://raw.githubusercontent.com/
kubernetes-sigs/kubefed/master/charts
"kubefed-charts" has been added to your repositories

$ helm2 repo list
NAME          URL
stable        https://kubernetes-charts.storage.googleapis.com
local         http://127.0.0.1:8879/charts
kubefed-charts https://raw.githubusercontent.com/kubernetes-sigs/kubefed/
master/charts
```

We can verify that the KubeFed chart is now available using this helm search command:

```
$ helm2 search kubefed
NAME                                CHART VERSION  APP VERSION DESCRIPTION
kubefed-charts/federation-v2       0.0.10
```

Update your repo:

```
$ helm2 repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "kubefed-charts" chart
repository
...Successfully got an update from the "stable" chart repository
Update Complete. ✨ Happy Helming! ✨
```

Then, the installation is as simple as:

```
$ helm2 install kubefed-charts/kubefed --name kubefed --version=$VERSION
--namespace kube-federation-system --devel
```

Registering clusters with the federation

Once the control plane is installed, make sure your `~/.kube/config` file has contexts for the host cluster and all of the member clusters. Then, you can use the `kubefedctl join` command to add clusters to the federation:

```
$ kf join cluster1 --cluster-context cluster-1 --host-cluster-context
cluster-1 --v=2
$ kf join cluster2 --cluster-context cluster-2 --host-cluster-context
cluster-1 --v=2
```

Note that `cluster-1`, here, is the host cluster but is also registered as a member cluster. This is totally fine. There is no need to have a dedicated host cluster.

To check the status of the federation, you can get the `kubefedclusters` CRDs:

```
$ kubectl -n kube-federation-system get kubefedclusters
```

NAME	READY	AGE
cluster-1	True	1m
cluster-2	True	1m



The preceding command works with Helm 2. In the future, we expect it to work with Helm 3. You can refer to <https://github.com/kubernetes-sigs/kubefed/blob/master/charts/kubefed/README.md#installing-the-chart> to check for this.

You use `kubectl` here and not `kubefedctl`.

If you want to unregister a cluster from the federation, use the `kubefedctl unjoin` command.

```
$ kf unjoin cluster-2 --cluster-context cluster-2 --host-cluster-context
cluster-1 --v=2
```

Working with federated API types

Kubernetes federation V1 supported only a limited number of Kubernetes API types. With KubeFed V2, any API type can be federated, including your own CRDs.

However, this doesn't happen automatically. You need to first enable any type that you want to federate:

```
$ kf enable <API Type>
```

The specification of the type to enable is pretty flexible. It can be the kind of type, the plural name, the group-qualified plural name, or the short name. For example, for deployments, it can be `Deployment` (kind), `deploy` (short name), `deployments` (plural), or `deployment.apps` (group-qualified plural name).

When you enable a type, `kubefedctl` generates a `Federated<Type>` CRD (for example, `FederatedDeployment`) and a `Federated<Type>Config` association to the original type (`Deployment`).

Note that the target type must be installed on all member clusters. Ideally, all clusters should run the same version of Kubernetes and be upgraded in tandem to avoid versioning issues. Even if all of the clusters run the same version of Kubernetes, they might not have the same CRDs installed. Remember, you can federate CRDs too, but only as long as they are installed on all clusters.

Suppose you have a CRD called `awesome` in the API group `example.com`. You can verify it is installed in `cluster-1` and `cluster-2` by running this little script:

```
CLUSTER_CONTEXTS=(cluster-1 cluster-2)
for c in ${CLUSTER_CONTEXTS}; do
    echo ----- ${c} -----
    kubectl --context=${c} api-resources --api-group=example.com
done
```

The result should be:

```
----- cluster1 -----
NAME      SHORTNAMES  APIGROUP      NAMESPACED  KIND
awesome                                example.com   true         Awesome
----- cluster2 -----
NAME      SHORTNAMES  APIGROUP      NAMESPACED  KIND
awesome                                example.com   true         Awesome
```

Federating resources

Enabling API types for your federation doesn't actually distribute any resources across the clusters. When you are ready to propagate resources across your federation, you can use the `federate` command:

```
kubefedctl federate <TYPE NAME> <RESOURCE NAME> [flags]
```

Let's look at the output of federating a pod without actually federating it (similar to a dry run). Here is the command:

```
$ kf federate pod trouble --output yaml
```

Here are some selected parts of the generated output. First, the API version is `types.kubefed.io/v1beta1`, and the kind is `FederatedPod`:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedPod
metadata:
  name: trouble
  namespace: default
```

Then comes the spec that contains the placement with its `clusterSelector`, in case you want to only federate to clusters that match some criteria:

```
spec:
  placement:
    clusterSelector:
      matchLabels: {}
```

The rest of the spec is a standard pod template:

```
template:
  metadata:
    labels:
      run: trouble
  spec:
    containers:
    - args:
      - bash
      image: g1g1/py-kube:0.2
      imagePullPolicy: IfNotPresent
      name: trouble
      resources: {}
      ...
```

By default, the federated resource will be created in the same namespace as the target type. Of course, the API type must be enabled and installed in all the federation clusters.

Federating an entire namespace

KubeFed supports whole namespace federation too. This is very useful because namespaces are convenient for organizing groups of resources together, and it often makes sense to federate all of the resources in a namespace in one fell swoop. Conceptually, a namespace is also a Kubernetes resource, so you can think of it as just federating a single resource, which is the namespace. However, in practice, namespaces are different from other resources because they are the only resources that contain other resources. The key to federated namespaces is the `--contents` flag that is required. You can also exclude some resources from the federation using the `--skip-api-resources` flag with a comma-separated list of resources:

```
kubefedctl federate namespace awesome-namespace --contents --skip-api-resources "secrets,apps"
```

Checking the status of federated resources

The top-level status field of a federated resource contains information about the propagation of the resource across the federation's member clusters. Here is an example:

```
apiVersion: types.kubefed.io/v1beta1
kind: FederatedNamespace
metadata:
  name: awesome-namespace
  namespace: awesome-namespace
spec:
  placement:      clusterSelector: {}
status:
  # The status True of the condition of type Propagation
  # indicates that the state of all member clusters is as
  # intended as of the last probe time.
  conditions:
  - type: Propagation
    status: True
    lastTransitionTime: "2019-12-08T14:33:45Z"
    lastUpdateTime: "2019-12-08T14:33:45Z"
  # The namespace 'awesome-namespace' has been verified to exist in the
  # following clusters as of the lastUpdateTime recorded
  # in the 'Propagation' condition. Since that time, no
  # change has been detected to this resource or the
  # resources it manages.
  clusters:
  - name: cluster-1
  - name: cluster-2
```

Using overrides

In the real world, not all clusters are the same. You may need to make various cluster-specific changes. The overrides field of the `FederatedDeployment` allows you to do exactly that. You specify the overrides field using the `jsonpatch` (<http://jsonpatch.com/>) syntax, similar to Kustomize.

For each override, you specify a path (for example, `/spec/replicas`), and then you either provide a value to set (for example, `value: 5`) or an operation (for example, `add` or `remove`) and a value to apply. Here is an example:

```
kind: FederatedDeployment
...
spec:
  ...
  overrides:
    # Apply overrides to cluster1
    - clusterName: cluster1
      clusterOverrides:
        # Set the replicas field to 5
        - path: "/spec/replicas"
          value: 5
        # Set the image of the first container
        - path: "/spec/template/spec/containers/0/image"
          value: "nginx:1.17.0-alpine"
        # Ensure the annotation "foo: bar" exists
        - path: "/metadata/annotations"
          op: "add"
          value:
            foo: bar
        # Ensure an annotation with key "baz" does not exist
        - path: "/metadata/annotations/baz"
          op: "remove"
```

Using placement to control federation

The placement field of the federated resources controls which member cluster the resource will be federated to. There two ways to place federated resources:

- A direct list of cluster names
- A label-based cluster selector

Here is a direct list placement:

```
spec:
  placement:
    clusters:
      - name: cluster-1
      - name: cluster-2
```

And here is a label-based cluster selector that deploys the resource to clusters that have a label `federate: True`:

```
spec:
  placement:
    clusterSelector:
      matchLabels:
        federate: True
```

So far, so good; however, there's more. If there is no placement field, or if there is a placement with an empty `clusterSelector`, then the resource will be placed in all of the member clusters:

```
spec:
  placement:
    clusterSelector: {}
```

However, if an empty list of clusters is specified, then the resource will not be deployed to ANY cluster!

```
spec:
  placement:
    clusters: []
```

In general, a list of clusters, if specified, always overrides the cluster selector. In this case, the resource will not be deployed to any cluster because of the empty list of clusters, not even to clusters that have a matching label:

```
spec:
  placement:
    clusters: []
    clusterSelector:
      matchLabels:
        federate: True
```

Debugging propagation failures

You can use `kubectl` and the same techniques you use to debug general Kubernetes issues. The `kubectl describe` command will show you events related to a federated resource:

```
$ kubectl describe <federated CRD> <CR name> -n awesome-namespace
```

If that doesn't help, then you can check the federation controller logs:

```
$ kubectl logs deployment/kubefed-controller-manager -n kube-federation-system
```

Now that you have a good sense of how to work with federated resources, let's look at some of the higher abstractions that are built on top of the basics.

Employing higher-order behavior

There several high-level multi-cluster considerations and patterns that KubeFed supports out of the box. These behaviors are built on top of the foundation building blocks of templates, overrides, and placement. Let's review them.

Utilizing multi-cluster Ingress DNS

Ingress in a single cluster is done at the edge of the cluster and forwards traffic into the cluster. Ingress literally means entrance. However, in the multi-cluster world, the situation is different. The cluster may be needed to send requests from the receiving cluster to a different cluster. Finding the correct destination relies on an external DNS, which is used in addition to the in-cluster CoreDNS. The primary idea is that the endpoints from all of the clusters are managed by a DNS endpoint controller and an **Ingress DNS controller**. They watch all the clusters and update the multi-cluster **IngressDNSRecord** and domain names. An **External DNS Controller** interacts with the external **DNS Provider** to assign external names that are valid across all the clusters and allow you to locate endpoints across clusters.

The following diagram illustrates the flow of information and the control loops:

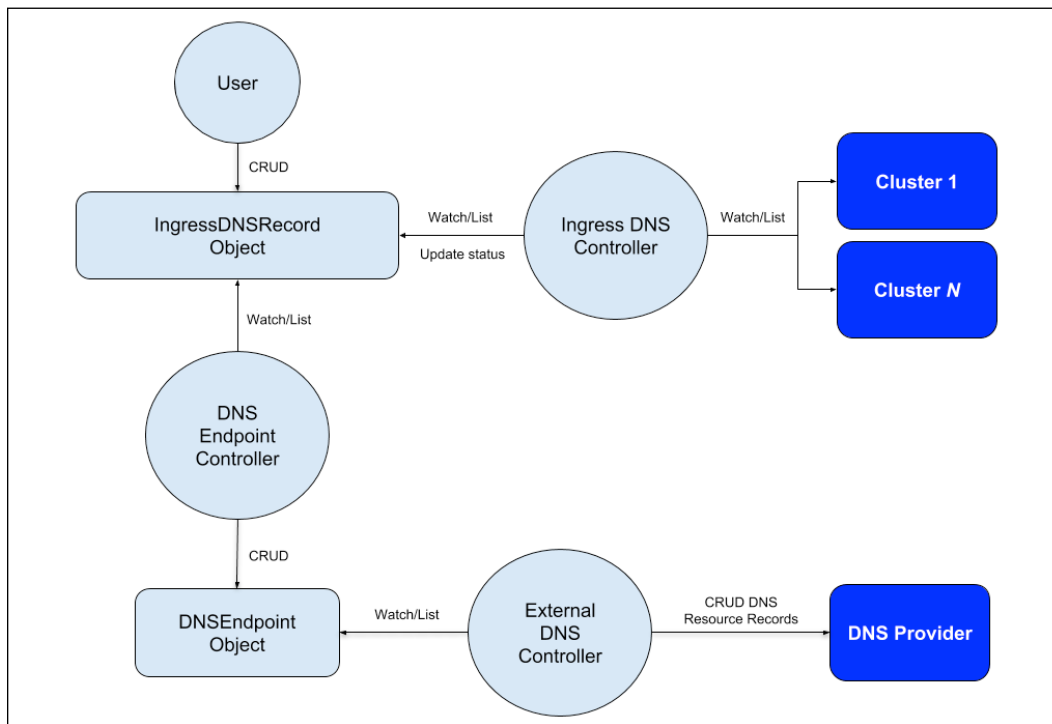


Figure 11.4: Ingress DNS

Utilizing multi-cluster Service DNS

In a Kubernetes federation, services need to be federated, which means their backing pods may be federated across multiple clusters. In order to access those pods and their endpoints, federated services require a mechanism that is very similar to the multi-cluster ingress DNS.

The typical workflow is:

1. Create `FederatedDeployment` and `FederatedService` objects.
2. Create a `Domain` object that associates a DNS zone and an authoritative nameserver for the KubeFed control plane.
3. Create a `ServiceDNSRecord` object that identifies the intended domain name of a multi-cluster `Service` object.

4. The DNS Endpoint controller will create a DNSEndpoint object associated with the ServiceDNSRecord. It contains three A records:
 - `<service>.<namespace>.<federation>.svc.<federation-domain>`
 - `<service>.<namespace>.<federation>.svc.<region>.<federation-domain>`
 - `<service>.<namespace>.<federation>.svc.<availability-zone>.<region>.<federation-domain>`
5. An external DNS system watches and lists DNSEndpoint objects and creates DNS resource records in external DNS providers.

The following diagram illustrates this process:

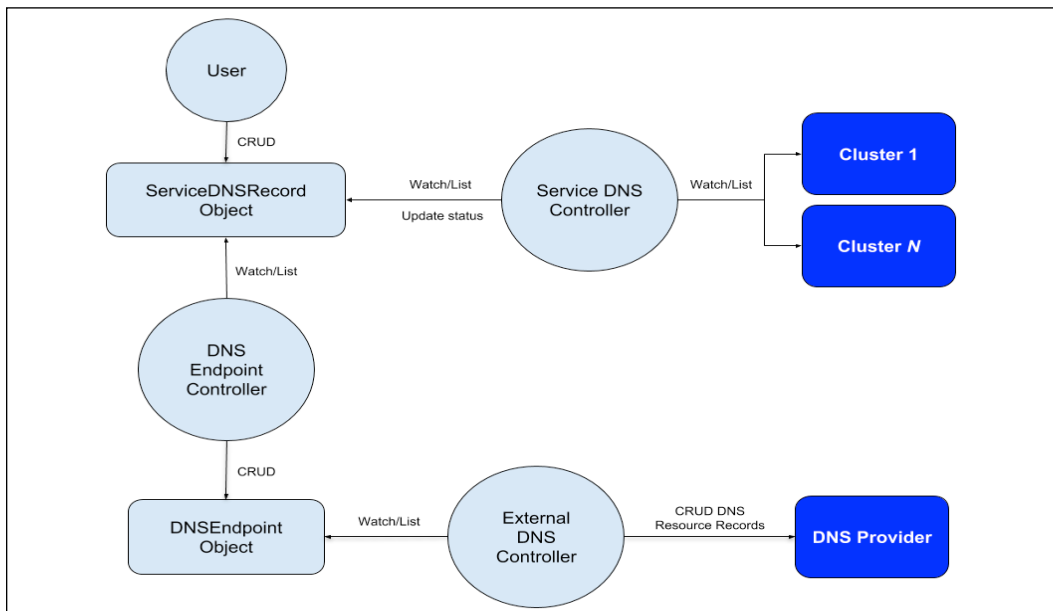


Figure 11.5: Service DNS

Next, let's look at the critical topic of multi-cluster scheduling.

Utilizing multi-cluster scheduling

Consider what it means to do multi-cluster scheduling. You need to specify for each deployment the total number of replicas just like on a single cluster, but you also need to provide some constraints and guidance for distributing the pods across the different clusters. Enter the `ReplicaSchedulingPreference`. This resource allows you to specify all of your preferences and accomplish a healthy distribution of your workloads across all the federation clusters.

Here is a simple example that specifies the total number of 15 replicas. KubeFed will try to distribute the 15 replicas evenly across all of the member clusters:

```
apiVersion: scheduling.kubefed.io/v1alpha1
kind: ReplicaSchedulingPreference
metadata:
  name: awesome-deployment
  namespace: awesome-ns
spec:
  targetKind: FederatedDeployment
  totalReplicas: 15
```

If there are three clusters in the federation, then 5 replicas will run in each cluster.

The following example is a little more elaborate and uses weighted distribution. The weights are 2:3 for cluster-1 and cluster-2. This means that the 15 replicas will be distributed in a ratio of 2 to 3, which results in 6 replicas running in cluster-1 and 9 replicas running in cluster-2:

```
apiVersion: scheduling.kubefed.io/v1alpha1
kind: ReplicaSchedulingPreference
metadata:
  name: awesome-deployment
  namespace: awesome-ns
spec:
  targetKind: FederatedDeployment
  totalReplicas: 15
  clusters:
    cluster-1:
      weight: 2
    cluster-2:
      weight: 3
```

Weighted distribution is nice, but it can lead to undesirable edge cases, especially when one of the member clusters becomes unreachable or is otherwise unavailable. To maintain some boundaries, you can specify for each cluster a range of the minimum and maximum number of replicas that are allowed to run in the cluster. KubeFed will do its best to maintain the weighted distribution without violating the constraints of the minimum and maximum number of replicas. In particular, the maximum number is a hard limit that KubeFed will always respect. The minimum number might be impossible to uphold under certain circumstances.

In the following example, the same 2:3 ratio of 15 replicas is maintained. However, cluster-1 has a `maxReplicas` limit of 5, so it will run just 5 replicas and not 6 as before. On the other hand, cluster-2 has a `maxReplicas` limit of 12, so it can pick up the slack and run 10 replicas, which is one more replica than before. The end result is that all 15 replicas are scheduled, that is, cluster-1 runs 5 replicas and cluster-2 runs 10 replicas, which is a ratio of 1:2 and not 2:3. That's the best KubeFed can do under this particular set of constraints while still scheduling all 15 replicas:

```
apiVersion: scheduling.kubefed.io/v1alpha1
kind: ReplicaSchedulingPreference
metadata:
  name: awesome-deployment
  namespace: awesome-ns
spec:
  targetKind: FederatedDeployment
  totalReplicas: 15
  clusters:
    cluster-1:
      weight: 2
      minReplicas: 4
      maxReplicas: 5
    cluster-2:
      weight: 3
      minReplicas: 4
      maxReplicas: 12
```

You can also do a uniform distribution with exceptions. For example, you can distribute evenly across all clusters except one particular cluster that has some constraints. Here, 100 replicas will be distributed evenly to all clusters, except for cluster-3, which must have at least 5 replicas:

```
apiVersion: scheduling.kubefed.io/v1alpha1
kind: ReplicaSchedulingPreference
metadata:
  name: awesome-deployment
  namespace: awesome-ns
spec:
  targetKind: FederatedDeployment
  totalReplicas: 100
  clusters:
    "*":
      weight: 1
    cluster-3:
      minReplicas: 5
      weight: 1
```


Cluster federation shines when you want to treat your multi-cluster system like one big cluster. However, in many cases, the correct level of abstraction is a collection of separate clusters. This is where the Gardener project comes in.

Introducing the Gardener project

The **Gardener** (<https://gardener.cloud/>) project is an open source project developed by SAP. It lets you manage thousands (yes, thousands!) of Kubernetes clusters efficiently and economically. Gardener solves a very complex problem, and the solution is elegant but not simple. In this section, we will cover the terminology of Gardener, its conceptual model, dive deep into its architecture, and learn about its features of extensibility. The primary theme of Gardener is to use Kubernetes to manage Kubernetes clusters. A good way to think about Gardener is as Kubernetes-control-plane-as-a-service.

Understanding the terminology of Gardener

The Gardener project, as you may have guessed, uses botanical terminology to describe the world. There is a garden, which is a Kubernetes cluster that is responsible for managing seed clusters. A seed is a Kubernetes cluster that is responsible for managing a set of shoot clusters. A shoot cluster is a Kubernetes cluster that runs actual workloads. The cool idea behind Gardener is that the shoot clusters contain only the worker nodes. The control planes of all the shoot clusters run as Kubernetes pods and services in the seed cluster.

The following diagram describes, in detail, the structure of Gardener and the relationships between its components:

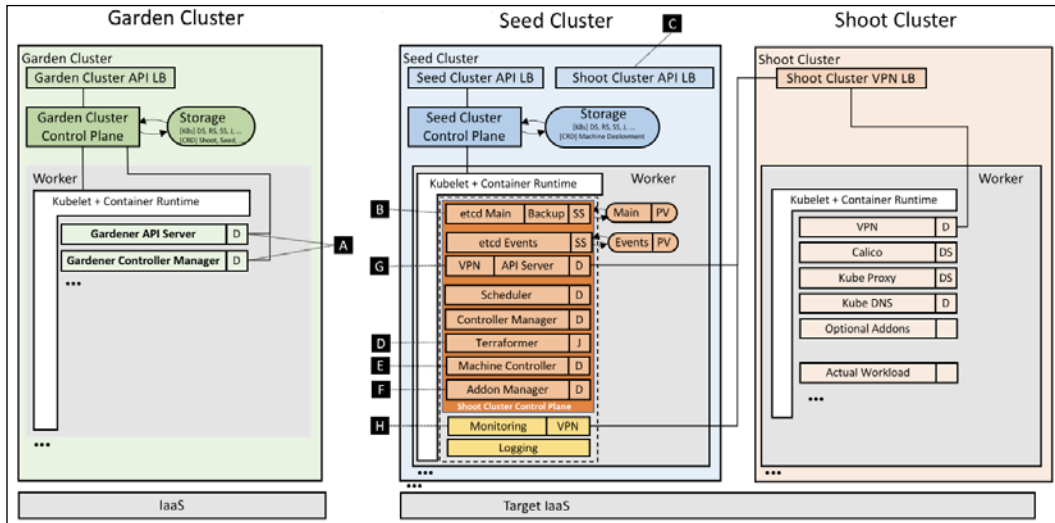


Figure 11.6: The structure of Gardener and its components

Don't panic! Underlying all of this complexity is a crystal-clear conceptual model.

Understanding the conceptual model of Gardener

The architecture diagram of Gardener can be overwhelming. Let's unpack it slowly and consider the underlying principles. Gardener really embraces the spirit of Kubernetes and offloads a lot of the complexity of managing a large set of Kubernetes clusters to Kubernetes itself. At its heart, Gardener is an aggregated API server (an extended Kubernetes API server) that manages a set of custom resources using various controllers. It embraces and takes full advantage of Kubernetes extensibility. This approach is common in the Kubernetes community. Define a set of custom resources and let Kubernetes manage them for you. The novelty of Gardener is that it takes this approach to the extreme and abstracts away parts of the Kubernetes infrastructure itself.

In a "normal" Kubernetes cluster, the control plane runs in the same cluster as the worker nodes. Typically, in large clusters, control plane components like the Kubernetes API server and etcd run on dedicated nodes and don't mix with the worker nodes. Gardener thinks in terms of many clusters, and it takes all the control planes of all the shoot clusters and uses a seed cluster to manage them. So the Kubernetes control planes of the shoot clusters are managed in the seed cluster as regular Kubernetes Deployments, which are automatically provided with replication, monitoring, self-healing, and rolling updates by Kubernetes.

So the control plane of a Kubernetes shoot cluster is analogous to a Deployment. The seed cluster, on the other hand, maps to a Kubernetes node. It manages multiple shoot clusters. We recommend that you have a seed cluster per cloud provider. The Gardener developers actually work on a Gardenlet controller for seed clusters that is similar to the kubelet on nodes.

If the seed clusters are like Kubernetes nodes, then the garden cluster that manages those seed clusters is like a Kubernetes cluster that manages its worker nodes.

By pushing the Kubernetes model this far, the Gardener project leverages the strengths of Kubernetes to achieve a robustness and a performance that would be very difficult to build from scratch.

Let's dive into the architecture.

Diving into the Gardener architecture

Gardener creates a Kubernetes namespace in the seed cluster for each shoot cluster. It manages the certificates of the shoot cluster as Kubernetes secrets in the seed cluster.

Managing cluster state

The etcd data store for each cluster is deployed as a StatefulSet with one replica. In addition to this, events are stored in a separate etcd instance. The etcd data is periodically snapshotted and stored in remote storage for backup and restore purposes. This enables the very fast recovery of clusters that have lost their control plane (for example, when an entire seed cluster becomes unreachable). Note that when a seed cluster goes down, the shoot cluster continues to run as usual.

Managing the control plane

As mentioned before, the control plane of a shoot cluster X runs in a separate seed cluster, while the worker nodes run in a shoot cluster. This means that pods in the shoot cluster can use an internal DNS to locate each other, but communication to the Kubernetes API server running in the seed cluster must be done through an external DNS. This means the Kubernetes API server runs as a service of type LoadBalancer.

Preparing the infrastructure

When creating a new shoot cluster, it's important to provide the necessary infrastructure. Gardener uses **Terraform** for this task. It generates a Terraform script based on the shoot cluster specification and stores it as a ConfigMap in the seed cluster. A dedicated Terraformer component runs as a Job, performs all the provisioning, and then writes the state into a separate ConfigMap.

Using the Machine controller manager

To provision nodes in a provider-agnostic way that can work for private clouds too, Gardener has several custom resources such as `MachineDeployment`, `MachineClass`, `MachineSet`, and `Machine`. They work with the Kubernetes Cluster Lifecycle group to unify their abstractions because there is a lot of overlap. In addition, Gardener takes advantage of the cluster auto-scaler to offload the complexity of scaling node pools up and down.

Networking across clusters

The seed cluster and shoot clusters can run on different cloud providers. The worker nodes in the shoot clusters are often deployed in private networks. Since the control plane needs to interact closely with the worker nodes (mostly the kubelet), the Gardener creates a VPN for direct communication.

Monitoring clusters

Observability is a big part of operating complex distributed systems. Gardener provides a lot of monitoring out of the box using the best of class open source projects like a central **Prometheus** (<https://github.com/prometheus/prometheus>) server, which is deployed in the garden cluster that collects information about all seed clusters.

In addition, each shoot cluster gets its own Prometheus instance in the seed cluster. To collect metrics, Gardener deploys two **kube-state-metrics** (<https://github.com/kubernetes/kube-state-metrics>) instances for each cluster (one for the control plane in the seed and one for the worker nodes in the shoot). The **node-exporter** (https://github.com/prometheus/node_exporter) is deployed too, to provide additional information about the nodes. The Prometheus **AlertManager** (<https://prometheus.io/docs/alerting/alertmanager/>) is used to notify the operator when something goes wrong. **Grafana** (<https://github.com/grafana/grafana>) is used to display dashboards with relevant data about the state of the system.

The gardenctl CLI

You can manage Gardener only using kubectl, but you will have to switch profiles and contexts a lot as you explore different clusters. Gardener provides the `gardenctl` command-line tool, which offers higher-level abstractions and can operate on multiple clusters at the same time. Here is an example:

```
$ gardenctl ls shoots
```

```
projects:
```

```
- project: team-a
```

```
  shoots:
```

```
    - dev-eu1
```

```
    - prod-eu1
```

```
$ gardenctl target shoot prod-eu1
```

```
[prod-eu1]
```

```
$ gardenctl show Prometheus
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
prometheus-0	3/3	Running	0	106d	10.241.241.42	ip-10-240-7-72.eu-central-1.compute.internal

```
URL: https://user:password@p.prod-eu1.team-a.seed.aws-eu1.example.com
```

One of the most prominent features of Gardener is its extensibility. It has a large surface area and supports many environments. Let's look at how extensibility is built into its design.

Extending Gardener

Gardener supports the following environments:

- AWS
- GCP
- Azure
- AliCloud
- Packet
- OpenStack

It started like Kubernetes itself with a lot of provider-specific support in the primary Gardener repository. Over time, it followed the Kubernetes example, which externalized cloud providers and migrated the providers to a separate Gardener extension. Providers can be specified using a `CloudProfile` CRD, such as:

```
apiVersion: gardener.cloud/v1alpha1
kind: CloudProfile
metadata:
  name: aws
spec:
  type: aws
# caBundle: |
# -----BEGIN CERTIFICATE-----
# ...
# -----END CERTIFICATE-----
dnsProviders:
- type: aws-route53
- type: unmanaged
kubernetes:
  versions:
  - 1.12.1
  - 1.11.0
  - 1.10.5
machineTypes:
- name: m4.large
  cpu: "2"
  gpu: "0"
  memory: 8Gi
# storage: 20Gi # optional (not needed in every environment, may
# only be specified if no volumeTypes have been specified)
```

```
...
volumeTypes:      # optional (not needed in every environment, may
                  # only be specified if no machineType has a 'storage' field)
- name: gp2
  class: standard
- name: io1
  class: premium
providerConfig:
  apiVersion: aws.cloud.gardener.cloud/v1alpha1
  kind: CloudProfileConfig
  constraints:
    minimumVolumeSize: 20Gi
    machineImages:
      - name: coreos
        regions:
          - name: eu-west-1
            ami: ami-32d1474b
          - name: us-east-1
            ami: ami-e582d29f
    zones:
      - region: eu-west-1
        zones:
          - name: eu-west-1a
            unavailableMachineTypes: # List of machine types defined above
that are not available in this zone
              - name: m4.large
            unavailableVolumeTypes: # List of volume types defined above
that are not available in this zone
              - name: gp2
              - name: eu-west-1b
              - name: eu-west-1c
```

Then, a shoot cluster will choose a provider and configure it with the necessary information:

```
apiVersion: gardener.cloud/v1alpha1
kind: Shoot
metadata:
  name: johndoe-aws
  namespace: garden-dev
spec:
  cloudProfileName: aws
  secretBindingName: core-aws
```

```

cloud:
  type: aws
  region: eu-west-1
  providerConfig:
    apiVersion: aws.cloud.gardener.cloud/v1alpha1
    kind: InfrastructureConfig
  networks:
    vpc: # specify either 'id' or 'cidr'
      # id: vpc-123456
      cidr: 10.250.0.0/16
    internal:
      - 10.250.112.0/22
    public:
      - 10.250.96.0/22
    workers:
      - 10.250.0.0/19
  zones:
    - eu-west-1a
  workerPools:
    - name: pool-01
      # Taints, labels, and annotations are not yet implemented. This
      # requires interaction with the machine-controller-manager, see
      # https://github.com/gardener/machine-controller-manager/
      # issues/174. It is only mentioned here as future proposal.
      # taints:
      # - key: foo
      #   value: bar
      #   effect: PreferNoSchedule
      # labels:
      # - key: bar
      #   value: baz
      # annotations:
      # - key: foo
      #   value: hugo
      machineType: m4.large
      volume: # optional, not needed in every environment, may only be
        # specified if the referenced CloudProfile contains the volumeTypes field
      type: gp2
      size: 20Gi
      providerConfig:
        apiVersion: aws.cloud.gardener.cloud/v1alpha1
        kind: WorkerPoolConfig

```



```
    machineImage:
      name: coreos
      ami: ami-d0dcef3
    zones:
      - eu-west-1a
    minimum: 2
    maximum: 2
    maxSurge: 1
    maxUnavailable: 0
  kubernetes:
    version: 1.11.0
    ...
  dns:
    provider: aws-route53
    domain: johndoe-aws.garden-dev.example.com
  maintenance:
    timeWindow:
      begin: 220000+0100
      end: 230000+0100
    autoUpdate:
      kubernetesVersion: true
  backup:
    schedule: "*/5 * * * *"
    maximum: 7
  addons:
    kube2iam:
      enabled: false
    kubernetes-dashboard:
      enabled: true
    cluster-autoscaler:
      enabled: true
    nginx-ingress:
      enabled: true
      loadBalancerSourceRanges: []
    kube-lego:
      enabled: true
      email: john.doe@example.com
```

However, the extensibility goals of Gardener go far beyond just being provider-agnostic. The overall process of standing up a Kubernetes cluster involves many steps. The Gardener project aims to let the operator customize each and every step by defining custom resources and Webhooks. Here is a general flow diagram with the CRDs, mutating/validating admission controllers, and Webhooks associated with each step:

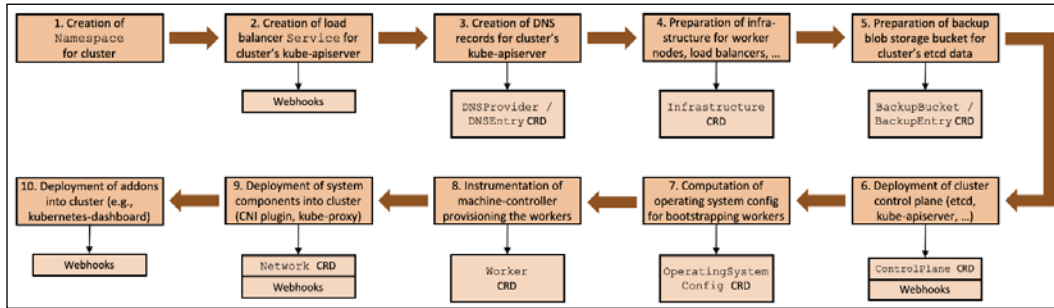


Figure 11.7: General flow diagram with the CRDs, mutating/validating admission controllers, and Webhooks

Here are the CRD categories that comprise the extensibility space of Gardener:

- DNS providers (for example, Route53 and CloudDNS)
- Blob storage providers (for example, S3, GCS, and ABS)
- Infrastructure providers (for example, AWS, GCP, and Azure)
- Operating systems (for example, CoreOS Container Linux, Ubuntu, and FlatCar Linux)
- Network plugins (for example, Calico, Flannel, and Cilium)
- Non-essential extensions (for example, the Let's Encrypt certificate service)

Gardener ring

Another novel idea is to create a cluster ring of at least three clusters, where shoot clusters serve as seed clusters for the next cluster in the ring. Together with the ability to migrate control planes to other clusters, the ring provides a robust solution that can self-heal if any cluster becomes unavailable. This is especially powerful if clusters are deployed on different cloud providers or at least in different regions. It has the potential to protect the garden from severe situations like a total region outage or even a complete cloud provider outage.

Here is how the ring is organized:

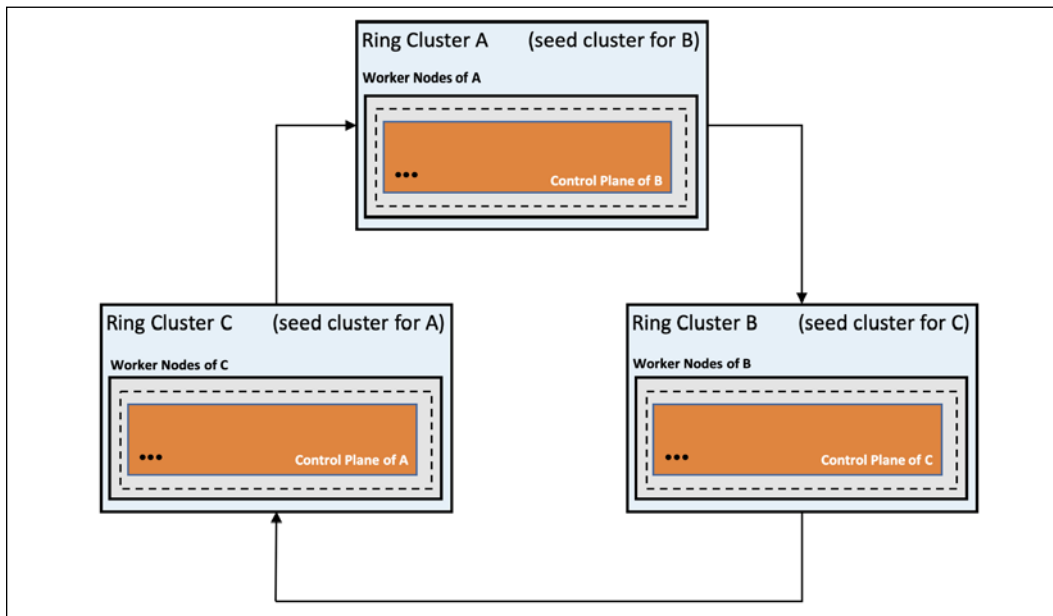


Figure 11.8: The Gardener Ring

Summary

In this chapter, we've covered the important aspects of a Kubernetes Cluster Federation as well as the management of many Kubernetes clusters using the Gardener project. Cluster federation is still in beta and is a little raw, but it is already usable. There aren't a lot of deployments, and the officially supported target platforms are currently AWS and GCE/GKE, but there is a lot of momentum behind cloud federation. It is a very important piece for building massively scalable systems on Kubernetes. We've discussed the motivation and use cases for the Kubernetes Cluster Federation, the federation control plane components, and the federated Kubernetes objects. We also looked at the less supported aspects of a federation, such as custom scheduling, federated data access, and auto-scaling. We then looked at how to run multiple Kubernetes clusters, which includes setting up a Kubernetes Cluster Federation, adding and removing clusters to the federation along with load balancing, federated failover when something goes wrong, service discovery, and migration. Then, we dived into running federated workloads across multiple clusters with federated services and the various challenges associated with this scenario.

The Gardener project has a very interesting approach and architecture. It tackles the problem of multiple clusters from a different angle and focuses on the large-scale management of clusters. It is relatively new, but it is used at scale by SAP and other partners.

At this point, you should have a clear understanding of the current state of a federation, what it takes to utilize the existing capabilities provided by Kubernetes, and what pieces you'll have to implement yourself to augment incomplete or immature features. Depending on your use case, you may decide that it's still too early or that you want to take the plunge. The developers who are working on the Kubernetes federation are moving fast, so it's very likely that it will be much more mature and battle-tested by the time you need to make your decision.

If you're in a position where you need to manage more than a few Kubernetes clusters, the Gardener project may be for you.

In the next chapter, we will explore the exciting world of serverless computing on Kubernetes. Serverless can mean two different things: when you don't have to manage servers for your long-running workloads and when running functions on a service. Both forms of serverless are available for Kubernetes and both are extremely useful.

12

Serverless Computing on Kubernetes

In this chapter, we will explore the fascinating world of serverless computing in the cloud. The term "serverless" is getting a lot of attention, but it is a misnomer used to describe two different paradigms. A true serverless application runs as a web application in the user's browser or a mobile app and only interacts with external services. The types of serverless systems we build on Kubernetes are different. We will explain exactly what serverless means on Kubernetes and how it relates to other serverless solutions. We will cover serverless cloud solutions, introduce Knative – the Kubernetes foundation for functions as a service – and dive into Kubernetes **Functions as a Service (FaaS)** frameworks.

Let's start by clarifying what serverless is all about.

Understanding serverless computing

OK. Let's get it out of the way. The servers are still there. The term "serverless" means that you don't have to provision, configure, and manage the servers yourself. Public cloud platforms were a real paradigm shift by eliminating the need for dealing with physical hardware, data centers, and networking. But, even on the cloud it takes a lot of work and knowhow to create and provision machine images, provision instances, configure them, upgrade and patch operating systems, define network policies, and manage certificates and access control. With serverless computing, large chunks of this important but tedious work go away.

The allure of serverless is multi-pronged:

- A whole category of problems dealing with provisioning goes away
- Capacity planning is a non-issue
- You pay only for what you use

You lose some control because you have to live with the choices made by the cloud provider. But, there is a lot of customization you can take advantage of for critical parts of the system. Of course, where you need total control you can still manage your own infrastructure.

The bottom line is that the serverless approach is not just hype, but provides real benefits. Let's examine the two flavors of serverless.

Running long-running services on "serverless" infrastructure

Long-running services are the bread and butter of microservice based distributed systems. These services must be always available and waiting to service requests, and can be scaled up and down to match the volume. In the traditional cloud, you had to provision enough capacity to handle spikes and changing volumes, which often led to over-provisioning or increased delays in processing when requests were waiting for under-provisioned services.

Serverless services address this issue with zero effort from developers and relatively little effort from operators. The idea is that you just mark your service to run on the serverless infrastructure and configure it with some parameters such as the expected CPU, memory, and any limits for the scaling. The service appears to other services and clients just like a traditional service you deployed on infrastructure you provisioned yourself.

Services that fall into this category have the following characteristics:

- They're always running (they never scale down to zero)
- They expose multiple endpoints (such as HTTP and gRPC)
- They require that you implement the request handling and routing yourself
- They can listen to events instead or in addition to exposing endpoints
- Service instances can maintain in-memory caches, long-term connections, and sessions
- In Kubernetes, microservices are represented directly by the Service resource

Now, let's look at FaaS.

Running FaaS on "serverless" infrastructure

Even in the largest distributed systems, we don't have every workload handling multiple requests per second. There are always tasks that need to run in response to relatively infrequent events, be it on schedule or invoked in an ad hoc manner. It's possible to have a long-running service just sitting there twiddling its virtual thumbs and processing a request every now and then. But that's wasteful. You can try to hitch such tasks to other long-running services, but that creates very undesirable coupling, which goes against the philosophy of microservices.

A much better approach is to treat such tasks separately and provide different abstractions and tooling to address them. Kubernetes has the concepts of a Job and a CronJob object. They address some of issues that FaaS tackles, but not completely.

A FaaS solution is often much simpler to get up and running compared to a traditional service. The developers may only need to write the code for the function. The FaaS solution will take care of the rest:

- Building and packaging
- Exposing as an endpoint
- Triggers based on events
- Automatic provisioning and scaling
- Monitoring and providing logs and metrics

Here are some of the characteristics of a FaaS solution:

- Runs on demand (that is, it can scale down to zero)
- Exposes a single endpoint (usually HTTP)
- Can be triggered by events or get an automatic endpoint
- Often has severe limitations on resource usage and maximum runtime
- Sometimes, it might have a cold start (that is, when scaling up from zero)

Serverless Kubernetes in the cloud

All the major cloud providers now supports serverless long-running services for Kubernetes. Surprisingly, Microsoft Azure was the first to offer this. Kubernetes interacts with nodes via the kubelet. The basic idea of serverless infrastructure is that instead of provisioning actual nodes (be they physical or on virtual machines (VMs)), a virtual node is created in some fashion. Different cloud providers use different solutions to accomplish this goal.

Don't forget the cluster autoscaler

Before jumping into cloud provider-specific solutions, make sure to check out the Kubernetes-native option of the cluster autoscaler. The cluster autoscaler scales the nodes in your cluster and doesn't suffer from the limitations of some of the other solutions. All the Kubernetes scheduling and control mechanisms work out of the box with the cluster autoscaler because it just automates adding and removing regular nodes from your cluster. No exotic or provider-specific capabilities are used.

But, you may have good reasons to prefer a more provider-integrated solution. For example, Fargate runs inside Firecracker, which is a lightweight VM with strong security boundaries (as a side note, Lambda functions run on Firecracker too). Similarly, Google Cloud Run runs in gVisor.

Azure AKS and Azure Container Instances

Azure has supported Azure Container Instances (ACI) for a long time. ACI is not Kubernetes-specific. It allows the running of on-demand containers on Azure in a managed environment. It is similar in some regards to Kubernetes, but is Azure-specific. It even has the concept of a container group, which is similar to a pod. All containers in a container group will be scheduled to run on the same host machine:

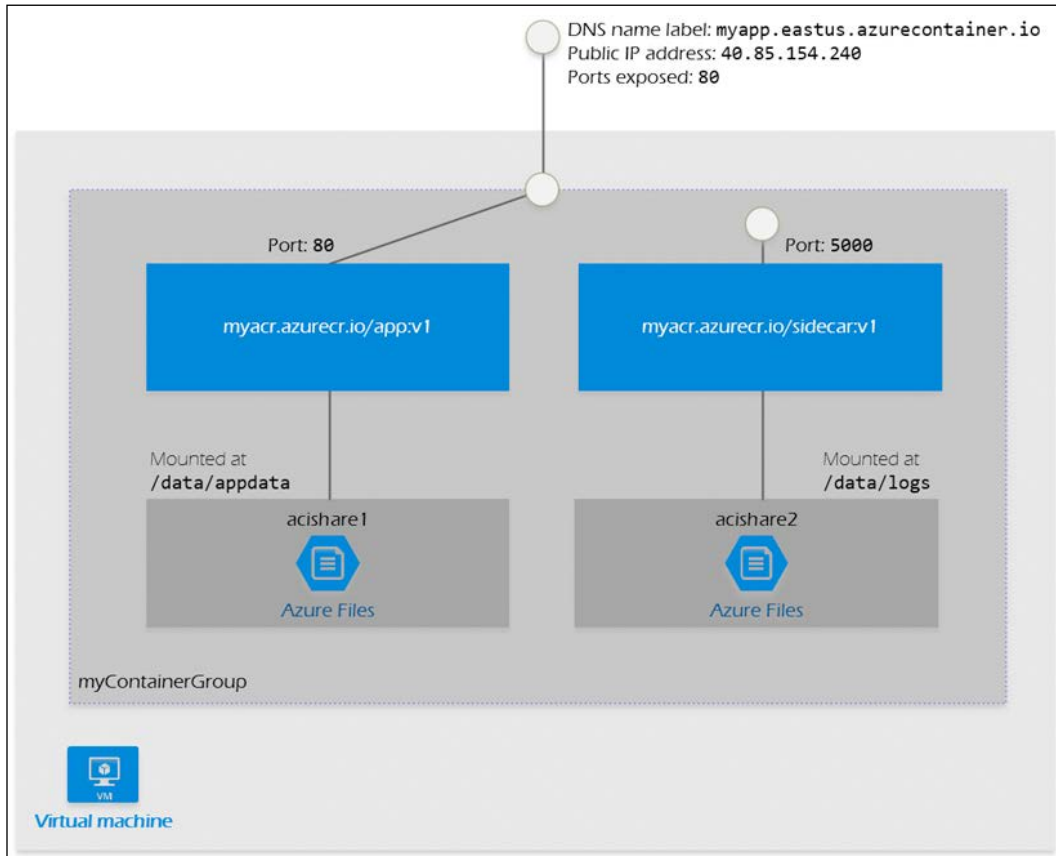


Figure 12.1: ACI container group

The integration with Kubernetes/AKS is modeled as bursting from AKS to ACI. The guiding principle here is that for your known workloads, you should provision your own nodes, but if there are spikes then the extra load will burst dynamically to ACI. This approach is considered more economical because running on ACI is more expansive than provisioning your own nodes. AKS uses the virtual kubelet (<https://virtual-kubelet.io/>) CNCF project to integrate your Kubernetes cluster with the infinite capacity of ACI. It works by adding a virtual node to your cluster backed by ACI that appears on the Kubernetes side as a single node with infinite resources:

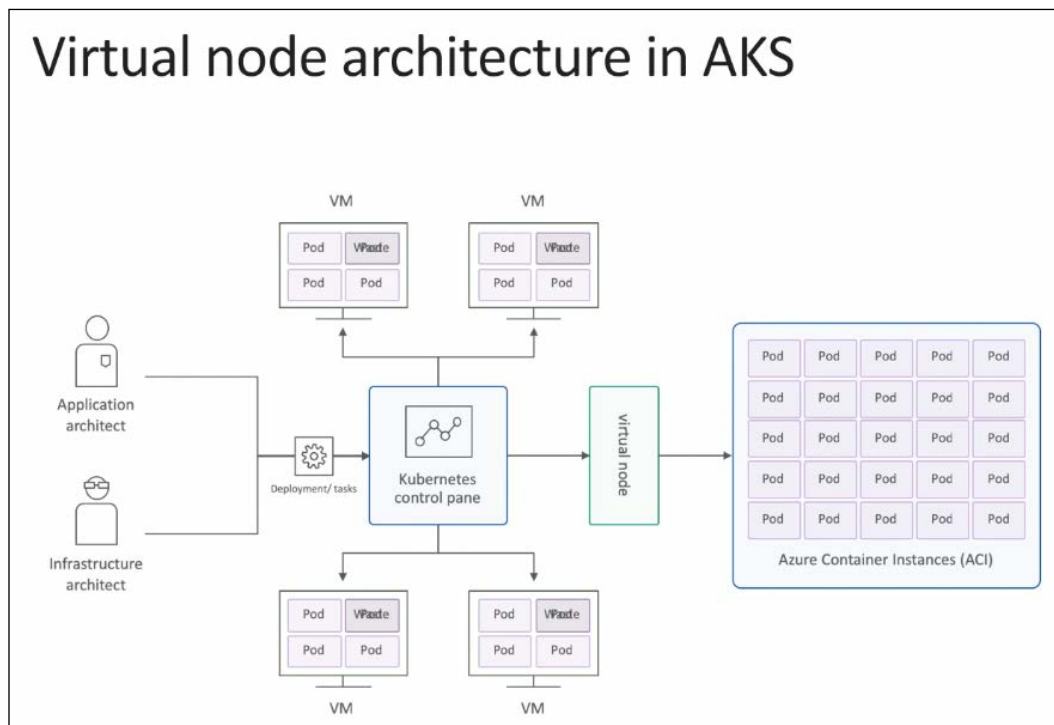


Figure 12.2: Virtual node architecture in AKS

Let's see how AWS does it with EKS and Fargate.

AWS EKS and Fargate

AWS released **Fargate** (<https://aws.amazon.com/fargate/>) in 2018, which is similar to Azure ACI and lets you run containers in a managed environment. Originally, you could use Fargate on EC2 or ECS (AWS proprietary container orchestration). At the big AWS conference, *re:Invent 2019*, Fargate became generally available on EKS too. That means that you now have a fully managed Kubernetes solution that is truly serverless.

EKS takes care of control plane and Fargate takes care of worker nodes for you:

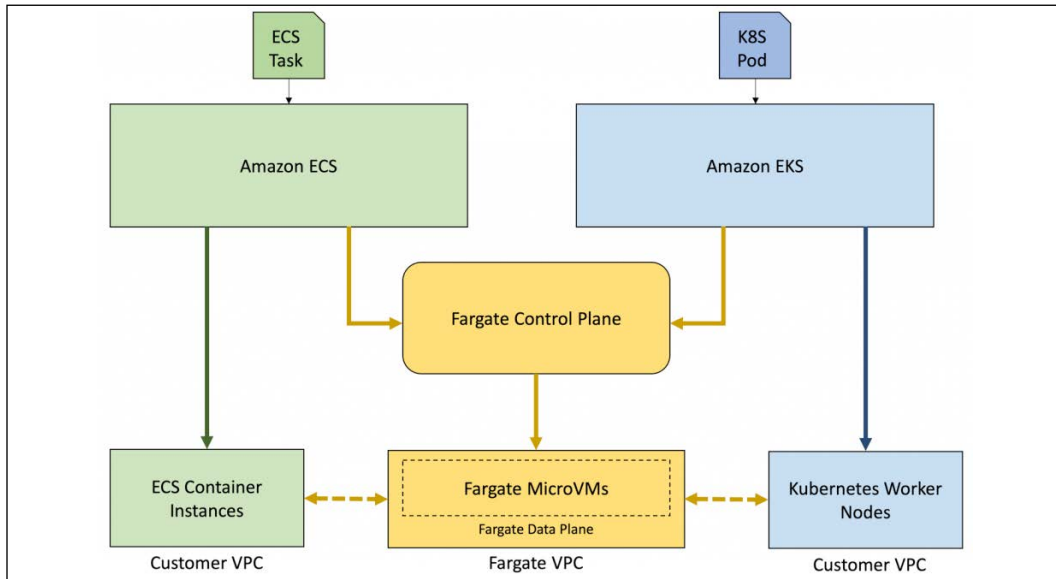


Figure 12.3: EKS and Fargate in practice

Use of EKS and Fargate models the interaction between your Kubernetes cluster and Fargate differently than AKS and ACI. While on AKS, a single infinite virtual node represents the entire capacity of ACI, on EKS each pod gets its own virtual node. But of course, those nodes are not real nodes. Fargate has its own control plane and data plane that support EC2 and ECS, as well as EKS. The EKS-Fargate integration is done via a set of custom Kubernetes controllers that watch for pods that need to be deployed to a particular namespace or have specific labels, and forwards those pods to be scheduled by Fargate. The following diagram illustrates the integration between EKS and Fargate:

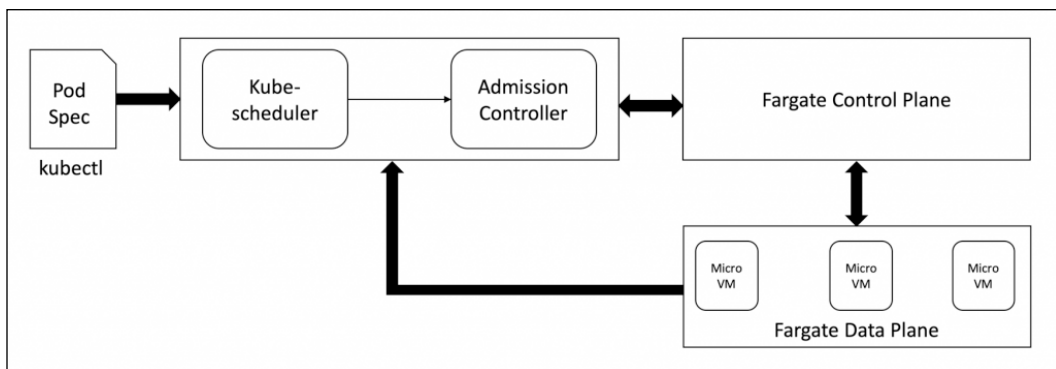


Figure 12.4: Integration between EKS and Fargate

When working with Fargate, there are several limitations you should be aware of:

- A maximum of 4 vCPUs and 30 GB memory per pod
- No support for stateful workloads that require persistent volumes or filesystems
- No DaemonSets, privileged pods, or pods that use HostNetwork or HostPort
- You can only use the application load balancer

If those limitations are too severe for you, you can try a more direct approach and utilize the virtual kubelet project to integrate Fargate into your cluster.

What about Google, the father of Kubernetes?

Google Cloud Run

It may come as a surprise, but Google is the Johnny-come-lately of serverless Kubernetes. Cloud Run is Google's serverless offering. It is based on Knative, which we will dissect in depth in the next section. The basic premise is that there are two flavors of Cloud Run. Plain Cloud Run is similar to ACI and Fargate. It lets you run containers in an environment fully managed by Google. Cloud for Anthos supports GKE and On-Prem lets you run containerized workloads in your GKE cluster.

Cloud for Anthos is currently the only serverless platform to allow running on custom machine types (including GPUs). Anthos Cloud Run services participate in the Istio service mesh and provide a streamlined Kubernetes-native experience.

Note that while managed Cloud Run instances use gVisor isolation, Anthos Cloud Run uses standard Kubernetes isolation (container-based).

It's time to learn more about Knative.

Knative

Kubernetes doesn't have built-in support for FaaS. As a result, many solutions were developed by the community for the ecosystem. The goal of Knative is to provide building blocks that multiple FaaS solutions can utilize without reinventing the wheel.

But that's not all! Knative also offers the unique capability of scaling long-running services all the way down to zero. This is a big deal. There are many use cases where you may prefer to have a long-running service that can handle a lot of requests coming its way in rapid succession. In those situations, it is not the best approach to fire a new function instance per request. But, when there is no traffic coming in, it's great to scale the service to zero instances, pay nothing, and leave more capacity for other services that may need more resources at that time. Knative supports other important use cases including load balancing based on percentages, load balancing based on metrics, blue-green deployments, canary deployments, and advanced routing. It can even optionally do automatic TLS certificates as well as HTTP monitoring. Finally, Knative works with both HTTP and gRPC.

There are currently two Knative components: **Knative Serving** and **Knative Eventing**. There used to also be a Knative build component, but it was factored out to form the foundation of **Tekton** (<https://github.com/tektoncd/pipeline>), a Kubernetes-native CD project.

Let's start with Knative Serving.

Knative Serving

The domain of Knative Serving is running versions services on Kubernetes and routing traffic to those services. This is above and beyond standard Kubernetes services. A Knative service defines several CRDs to model its domain: **Service**, **Route**, **Configuration**, and **Revision**. The Service manages a **Route** and a **Configuration**. A **Configuration** can have multiple **Revisions**.

The **Route** can route service traffic to a particular revision. Here is a diagram that illustrates the relationship between the different objects:

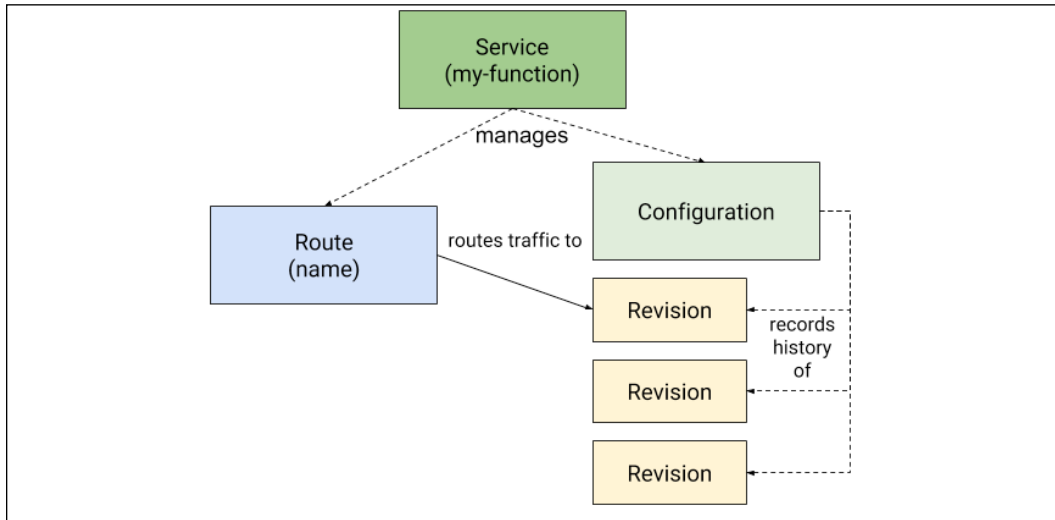


Figure 12.5: Relationships between Knative objects

The Knative Service object

The **Knative Service** object combines the Kubernetes Deployment and Service into a single object. That makes a lot of sense because, except for the special case of headless services (<https://kubernetes.io/docs/concepts/services-networking/service/#headless-services>), there is always a deployment behind every service.

The Service automatically manages the entire life cycle of its workload. It is responsible for creating the route and configuration and a new revision whenever the service is updated. This is very convenient because the user just needs to deal with the Service object.

Here is the metadata for the helloworld-go Knative service:

```
$ kubectl get ksvc helloworld-go -o json | jq .metadata
{
  "annotations": {
    "serving.knative.dev/creator": "minikube-user",
    "serving.knative.dev/lastModifier": "minikube-user"
  },
  "creationTimestamp": "2019-12-25T18:44:34Z",
  "generation": 1,
  "name": "helloworld-go",
```

```

"namespace": "default",
"resourceVersion": "43258",
"selfLink": "/apis/serving.knative.dev/v1/namespaces/default/services/
helloworld-go",
"uid": "d1979430-464e-49d6-bf68-bb384d1ef0b3"
}

```

And here is the spec:

```

$ kubectl get ksvc helloworld-go -o json | jq .spec
{
  "template": {
    "metadata": {
      "creationTimestamp": null
    },
    "spec": {
      "containerConcurrency": 0,
      "containers": [
        {
          "env": [
            {
              "name": "TARGET",
              "value": "Yeah, it works!!!"
            }
          ],
          "image": "gcr.io/knative-samples/helloworld-go",
          "name": "user-container",
          "readinessProbe": {
            "successThreshold": 1,
            "tcpSocket": {
              "port": 0
            }
          },
          "resources": {}
        }
      ],
      "timeoutSeconds": 300
    }
  },
  "traffic": [
    {
      "latestRevision": true,
      "percent": 100
    }
  ]
}

```


Note the `traffic` section of the spec that directs 100% of requests to the latest revision. This is what determines the Route CRD.

The Knative Route object

The **Knative Route** object allows the directing of a percentage of incoming requests to particular revisions. The default is 100% to the latest revision, but you can change it. This allows advanced deployment scenarios such as blue-green deployments as well as canary deployments.

For example, this is how to switch from blue to green when deploying a new version. Start with 100% going to the current revision and 0% going to the new revision (tagged v2):

```
apiVersion: serving.knative.dev/v1
kind: Route
metadata:
  name: blue-green-demo # Route name is unchanged, since we're updating
                        an existing Route
  namespace: default
spec:
  traffic:
    - revisionName: blue-green-demo-lcfrd
      percent: 100 # All traffic still going to the first revision
    - revisionName: blue-green-demo-m9548
      percent: 0 # 0% of traffic routed to the second revision
      tag: v2 # A named route
```

Then, to switch all traffic to the new version, apply the following change to the route:

```
apiVersion: serving.knative.dev/v1
kind: Route
metadata:
  name: blue-green-demo # Updating our existing route
  namespace: default
spec:
  traffic:
    - revisionName: blue-green-demo-lcfrd
      percent: 0
      tag: v1 # Adding a new named route for v1
    - revisionName: blue-green-demo-m9548
      percent: 100
      # Named route for v2 has been removed, since we don't need it
      # anymore
```

If you want more gradual shifting of the load, you can do different percentages as long as they add up to 100%.

The Knative Configuration object

The Configuration CRD contains the latest version of the service and the number of generations. For example, if we update the service to version 2:

```
apiVersion: serving.knative.dev/v1 # Current version of Knative
kind: Service
metadata:
  name: helloworld-go # The name of the app
  namespace: default # The namespace the app will use
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go # The URL to the
          image of the app
          env:
            - name: TARGET # The environment variable printed out by
              the sample app
              value: "Yeah, it still works - version 2 !!!"
```

Then the configuration will contain this new version, but mark it as generation 2:

```
$ kubectl get configurations helloworld-go -o yaml
apiVersion: serving.knative.dev/v1
kind: Configuration
metadata:
  annotations:
    serving.knative.dev/creator: minikube-user
    serving.knative.dev/lastModifier: minikube-user
  creationTimestamp: "2019-12-25T18:44:34Z"
  generation: 2
  labels:
    serving.knative.dev/route: helloworld-go
    serving.knative.dev/service: helloworld-go
  name: helloworld-go
  namespace: default
  ownerReferences:
    - apiVersion: serving.knative.dev/v1alpha1
      blockOwnerDeletion: true
```

```
    controller: true
    kind: Service
    name: helloworld-go
    uid: d1979430-464e-49d6-bf68-bb384d1ef0b3
    resourceVersion: "75459"
    selfLink: /apis/serving.knative.dev/v1/namespaces/default/configurations/
helloworld-go
    uid: c1ce42e0-e6ec-412f-9e07-4c41370e024c
spec:
  template:
    metadata:
      creationTimestamp: null
    spec:
      containerConcurrency: 0
      containers:
      - env:
        - name: TARGET
          value: Yeah, it still works - version 2 !!!
        image: gcr.io/knative-samples/helloworld-go
        name: user-container
        readinessProbe:
          successThreshold: 1
          tcpSocket:
            port: 0
          resources: {}
          timeoutSeconds: 300
status:
  conditions:
  - lastTransitionTime: "2019-12-26T03:21:45Z"
    status: "True"
    type: Ready
  latestCreatedRevisionName: helloworld-go-l58sn
  latestReadyRevisionName: helloworld-go-l58sn
  observedGeneration: 2
```

But note that the Route will still point to version 1:

```
$ kubectl get route helloworld-go -o yaml
apiVersion: serving.knative.dev/v1
kind: Route
metadata:
  annotations:
    serving.knative.dev/creator: minikube-user
```

```

    serving.knative.dev/lastModifier: minikube-user
creationTimestamp: "2019-12-25T18:44:35Z"
generation: 1
labels:
  serving.knative.dev/service: helloworld-go
name: helloworld-go
namespace: default
ownerReferences:
- apiVersion: serving.knative.dev/v1alpha1
  blockOwnerDeletion: true
  controller: true
  kind: Service
  name: helloworld-go
  uid: d1979430-464e-49d6-bf68-bb384d1ef0b3
resourceVersion: "75500"
selfLink: /apis/serving.knative.dev/v1/namespaces/default/routes/
helloworld-go
uid: 5a22217f-7090-46d2-b009-61ca0d3b6561
spec:
  traffic:
  - configurationName: helloworld-go
    latestRevision: true
    percent: 100
status:
  address:
    url: http://helloworld-go.default.svc.cluster.local
  conditions:
  - lastTransitionTime: "2019-12-25T18:45:25Z"
    status: "True"
    type: AllTrafficAssigned
  - lastTransitionTime: "2019-12-26T03:21:51Z"
    status: "True"
    type: IngressReady
  - lastTransitionTime: "2019-12-26T03:21:51Z"
    status: "True"
    type: Ready
  observedGeneration: 1
  traffic:
  - latestRevision: true
    percent: 100
    revisionName: helloworld-go-l58sn
  url: http://helloworld-go.default.example.com

```

The Knative Revision object

However, both the current and new versions will be captured as separate revisions:

```
$ kubectl get revisions
```

NAME	CONFIG NAME	K8S SERVICE NAME	GENERATION	
READY	REASON			
helloworld-go-fltxb	helloworld-go	helloworld-go-fltxb	1	True
helloworld-go-l58sn	helloworld-go	helloworld-go-l58sn	2	True

As you can see, both generations are present and that allows routing to either one of them or dividing the traffic between them using a Route, as we saw earlier.

To summarize, Knative Serving provides better deployment and networking for Kubernetes for long-running services and functions. Let's see what Knative Eventing brings to the table.

Knative Eventing

Traditional services on Kubernetes or other systems expose API endpoints that consumers can hit (often over HTTP) to send a request for processing. This pattern of request-response is very useful and hence is so popular. However, this is not the only pattern for invoking services or functions. Most distributed systems have some form of loosely coupled interactions where events are published. It is desirable to invoke some code when events occur.

Before Knative, you had to build this capability yourself or use some third-party library that binds events to code. Knative Eventing aims to provide a standard way to accomplish this task. It is compatible with the CNCF's CloudEvents specification (<https://github.com/cloudevents/spec/blob/master/spec.md#design-goals>).

Getting familiar with Knative Eventing terminology

Before diving into the architecture, let's define some terms and concepts we will use later.

Event consumer

There are two types of event consumers: Addressable and Callable. Addressable consumers can receive events over HTTP through their `status.address.url` field. The Kubernetes Service object doesn't have such a field, but it is also considered a special case of an Addressable consumer.

Callable consumers receive an event delivered over HTTP and may return another event in the response that will be consumed just like an external event. Callable consumers provide an effective way to transform events.

Event source

This is the originator of an event. Knative supports many common sources and you can write your own custom event source too. Here is a list of supported event sources (many of them are still in the early development phase):

- AWS SQS
- Apache Camel
- Apache CouchDB
- Apache Kafka
- BitBucket
- Cron Job
- GCP Pub/Sub
- GitHub
- GitLab
- Google Cloud Scheduler
- Kubernetes (Kubernetes Events)

There are also a few meta controllers that assist in implementing event sources, such as the following:

- ContainerSource: https://github.com/knative/eventing/blob/master/pkg/apis/sources/v1alpha1/container_source_types.go
- AutoContainerSource: <https://github.com/Harwayne/auto-container-source>
- Same Source: <https://github.com/grantr/sample-source>

Broker and Trigger

A broker mediates events identified by particular attributes and matches them with consumers via triggers. The trigger includes a filter of event attributes and an addressable consumer. When the event arrives at the broker, it forwards it to consumers that have triggers with matching filters to the event attributes. The following diagram illustrates this workflow:

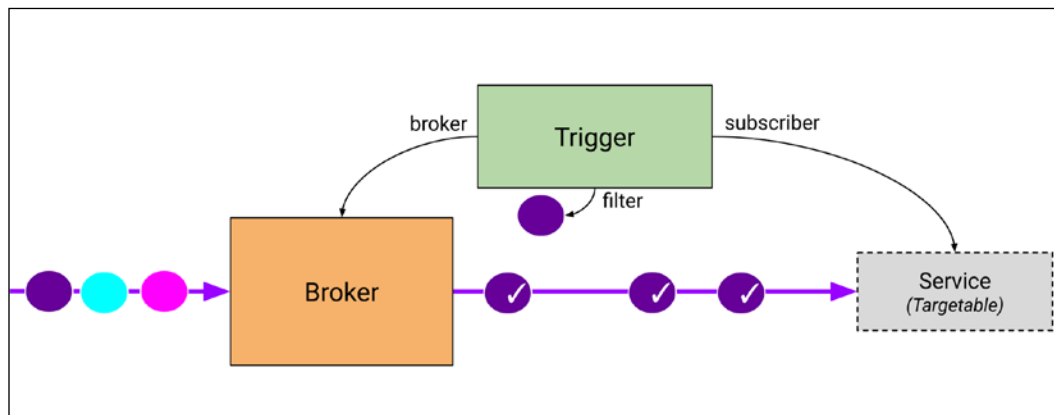


Figure 12.6: Broker and trigger workflow

Event types and the Event Registry

Events can have a type, which is modeled as the EventType CRD. The Event Registry stores all the event types. Triggers can use the event type as one of their filter criteria.

Channels and subscriptions

A channel is an optional persistence layer. Different event types may be routed to different channels with different backing stores. Some channels may store an event in memory, while other channels may persist to disk via NATS streaming, Kafka, or something similar. Subscribers (consumers) eventually receive and handle the events.

The architecture of Knative Eventing

The current architecture supports two modes of event delivery:

- Simple delivery
- Fan-out delivery

The simple delivery is just 1:1 source to consumer. The consumer can be a core Kubernetes service or a Knative service. If the consumer is unreachable, the source is responsible for handling the fact that the event can't be delivered. The source can retry, log an error, or take any other appropriate action.

The following diagram illustrates this simple concept:

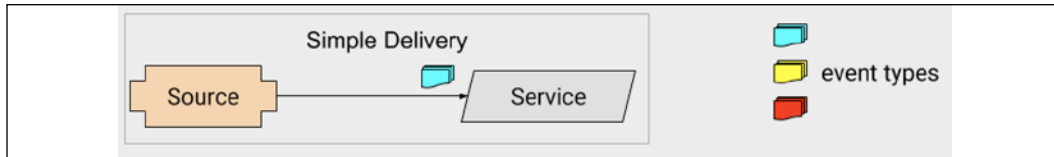


Figure 12.7: Simple delivery

Fan-out delivery supports arbitrarily complex processing, where multiple consumers subscribe to the same event on a channel. Once an event is received by the channel, the source is not responsible for the event anymore. This allows more dynamic subscriptions of consumers because the source doesn't even know who the consumers are.

The following diagram illustrates the complex processing and subscription patterns that can arise when using channels:

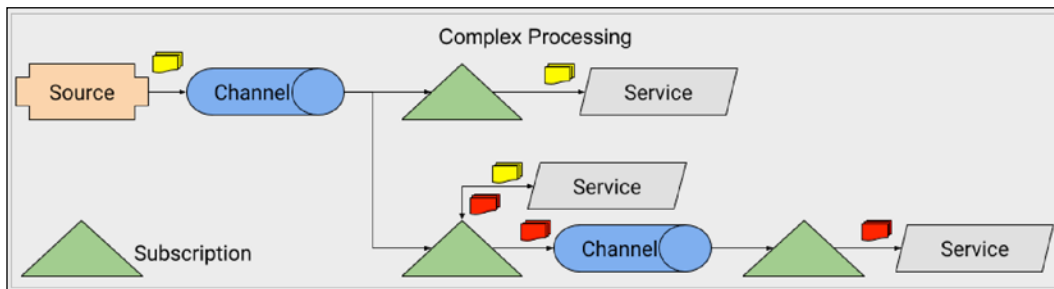


Figure 12.8: Delivery with channels and subscriptions

At this point, you should have a decent understanding of the scope of Knative and how it establishes a solid serverless foundation for Kubernetes. Let's play around a little with Knative and see what it feels like.

Taking Knative for a ride

Knative is not a small piece of software. It has many moving parts, it supports many modes of operation, and it can integrate with many systems. We will just explore a small part of it using a Minikube cluster.

We will perform the following:

- Create a compatible Minikube cluster
- Install Knative using Gloo as its ingress gateway
- Deploy a Knative service
- Invoke the Knative service
- Verify it can scale to zero

Here we go...

Installing Knative

There are many ways to install Knative. See <https://knative.dev/docs/install>. We will use the minikube installation. First, let's create a minikube cluster with the specific parameters recommended by Knative:

```
$ minikube start --memory=8192 --cpus=4 \
    --vm-driver=hyperkit \
    --disk-size=30g \
    --extra-config=apiserver.enable-admission-plugins="LimitRan
ger,NamespaceExists,NamespaceLifecycle,ResourceQuota,ServiceAccount,Default
StorageClass,MutatingAdmissionWebhook"
```

😊 minikube v1.10.1 on Darwin 10.15.5

🌟
[] Selecting 'hyperkit' driver from user configuration (alternates: [])

📁 Downloading driver docker-machine-driver-hyperkit:
> docker-machine-driver-hyperkit.sha256: 65 B / 65 B [---] 100.00% ?
p/s 0s
> docker-machine-driver-hyperkit: 10.81 MiB / 10.81 MiB 100.00% 8.84
MiB p

🔑 The 'hyperkit' driver requires elevated permissions. The following
commands will be executed:

```
$ sudo chown root:wheel /Users/gigi.sayfan/.minikube/bin/docker-
machine-driver-hyperkit
```

```
$ sudo chmod u+s /Users/gigi.sayfan/.minikube/bin/docker-machine-
driver-hyperkit
```

Password:

🕒 Downloading VM boot image ...

```

> minikube-v1.10.1.iso.sha256: 65 B / 65 B [-----] 100.00% ?
p/s 0s
> minikube-v1.10.1.iso: 150.93 MiB / 150.93 MiB [] 100.00% 10.27 MiB
p/s 14s
🔥 Creating hyperkit VM (CPUs=4, Memory=8192MB, Disk=30000MB) ...
🐳 Preparing Kubernetes v1.18.2 on Docker '19.03.8' ...
  ▪ apiserver.enable-admission-plugins=LimitRanger,NamespaceExists,Namesp
  aceLifecycle,ResourceQuota,ServiceAccount,DefaultStorageClass,MutatingAdmis
  sionWebhook
📄 Downloading kubeadm v1.18.0
📄 Downloading kubelet v1.18.0
🚚 Pulling images ...
🚀 Launching Kubernetes ...
⌚ Waiting for cluster to come online ...
🎉 Done! kubectl is now configured to use "minikube"

```

Knative requires an ingress gateway. The current options are Istio, Ambassador, and Gloo. Let's use Gloo as it is very lightweight:

```

$ curl -sL https://run.solo.io/gloo/install | sh
Attempting to download gloctl version v1.3.29
Downloading gloctl-darwin-amd64...
Download complete!, validating checksum...
Checksum valid.
Gloo was successfully installed 🎉

```

Add the gloo CLI to your path with:

```
export PATH=$HOME/.gloo/bin:$PATH
```

Now run:

```

gloctl install gateway      # install gloo's function gateway
                             functionality into the 'gloo-system' namespace
gloctl install ingress      # install very basic Kubernetes Ingress
                             support with Gloo into namespace gloo-system
gloctl install knative      # install Knative serving with Gloo configured
                             as the default cluster ingress
Please see visit the Gloo Installation guides for more: https://gloo.solo.
io/installation/

```

Make sure gloctl is on your path (I copied \$HOME/.gloo/bin/gloctl to /usr/local/bin) then run the following command to install both Gloo and Knative:

```

$ gloctl install knative -g --install-knative-version="0.15.0"
installing Knative CRDs...

```

installing Knative...

Knative successfully installed!

```
$ glooctl install knative --install-knative=false
```

Creating namespace gloo-system... Done.

Starting Gloo installation...

Gloo was successfully installed!

Deploying a Knative service

At this point, we can deploy Knative services. Here is a simple hello-world app that returns a "Hello Yeah, it works!!!" message. Save the following YAML to `service.yaml`:

```
apiVersion: serving.knative.dev/v1 # Current version of Knative
kind: Service
metadata:
  name: helloworld-go # The name of the app
  namespace: default # The namespace the app will use
spec:
  template:
    spec:
      containers:
        - image: gcr.io/knative-samples/helloworld-go # The URL to the
          image of the app
          env:
            - name: TARGET # The environment variable printed out by
              the sample app
              value: "Yeah, it works!!!"
```

Then deploy it:

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

service.serving.knative.dev/helloworld-go created

Invoking a Knative service

You can view the deployed Knative service by getting the `kservice` (or `ksvc`) CRD:

```
$ kubectl get kservice
```

NAME	URL	LATESTCREATED
helloworld-go	http://helloworld-go.default.example.com	helloworld-go-ftxb
helloworld-go-ftxb	helloworld-go-ftxb	True

The LATESTCREATED and LATESTREADY columns correspond to a standard Kubernetes service of type ClusterIP that the Knative service delegates the actual work to:

```
$ kubectl get svc helloworld-go-fltzb
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
helloworld-go-fltzb	ClusterIP	10.96.124.21	<none>	80/TCP	67m

Normally, the Gloo external proxy LoadBalancer service receives the incoming requests and routes them to the Knative service. But minikube LoadBalancer services have no external IP:

```
$ kubectl get svc knative-external-proxy -n gloo-system
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
knative-external-proxy	LoadBalancer	10.96.72.13	<pending>	80:30150/TCP,443:30146/TCP	4h53m

To work around this, we need to get the URL of the proxy using `glooctl proxy url --name knative-external-proxy` and use that URL to access the service while passing the original URL as a Host header:

```
$ curl -H "Host: helloworld-go.default.example.com" $(glooctl proxy url
--name knative-external-proxy)
Hello Yeah, it works!!!!
```

Checking the scale-to-zero option in Knative

Knative is configured by default to scale to zero with a grace period of 30 seconds. That means that after 30 seconds of inactivity (no request coming in), all the pods will be terminated until a new request comes in. To verify this, we can wait 30 seconds and check the pods in the default namespace:

```
$ kubectl get po
```

```
No resources found in default namespace.
```

Then, we can invoke the service and immediately check the pods:

```
$ curl -H "Host: helloworld-go.default.example.com" $(glooctl proxy url
--name knative-external-proxy)
Hello Yeah, it works!!!!
```

```
$ kubectl get po
```

NAME	READY	STATUS
RESTARTS AGE		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	2/2	Running 0
6s		

Let's watch when the pods disappear by using the `-w` flag. Apparently, the pods start terminating after a minute:

```
$ kubectl get po -w
```

NAME	READY	STATUS
RESTARTS AGE		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	2/2	Running 0
49s		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	2/2	Terminating 0
62s		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	1/2	Terminating 0
83s		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	0/2	Terminating 0
84s		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	0/2	Terminating 0
96s		
helloworld-go-flt x b-deployment-74b5dc8665-2j7hw	0/2	Terminating 0
96s		

At this point, we've had a little fun with Knative and can now move on to discussing FaaS solutions on Kubernetes.

Kubernetes FaaS frameworks

Let's acknowledge the elephant in the room – FaaS. The Kubernetes Job and CronJob are great, and having cluster autoscaling and cloud providers managing the infrastructure is awesome. Knative, with its scale-to-zero and traffic routing functionalities, is super cool. But what about actual FaaS? Fear not – Kubernetes has many options here. Maybe too many options. There are more than ten FaaS frameworks for Kubernetes:

- Fission
- Kubeless
- FaaS

- OpenWhisk
- Riff (built on top of Knative)
- Nuclio
- Funktion
- BlueNimble
- Fn
- Gestalt
- Rainbond
- IronFunctions

We will look into a few of the more popular options.

Fission

Fission (<https://fission.io/>) is a mature and well-documented framework. It models the FaaS world as environments, functions, and triggers. Environments are needed to build and run your function code for the specific languages. Each language environment contains an HTTP server and often a dynamic loader (for dynamic languages). Functions are the objects that represent the serverless functions and triggers determine how the functions deployed in the cluster can be invoked. There are four kinds of triggers:

- **HTTP trigger:** Invokes a function via an HTTP endpoint
- **Timer trigger:** Invokes a function at a certain time
- **Message queue trigger:** Invokes a function when an event is pulled from message queue (this supports Kafka, NATS, and Azure queues)
- **Kubernetes watch trigger:** Invokes a function in response to a Kubernetes event in your cluster

It's interesting that the message queue triggers are not just *fire and forget*. They support optional response and error queues. Here is a diagram that shows the flow:

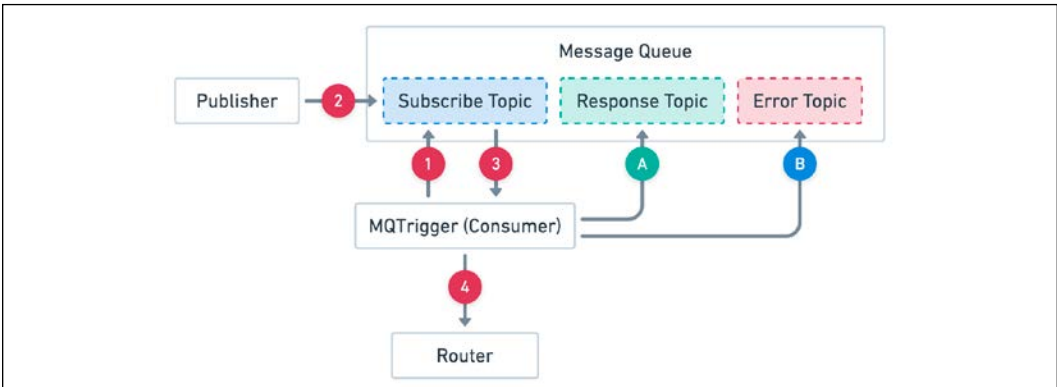


Figure 12.9: Message queue trigger flow

Fission is proud of its 100-millisecond cold-start. It achieves this by keeping a pool of "warm" containers with a small dynamic loader. When a function is first called, there is a running container ready to go and the code is sent to this container for execution. In a sense, Fission cheats because it never starts cold. The bottom line is that Fission doesn't scale to zero, but is very fast for first-time calls.

Fission Workflows

Fission has one other claim to fame – Fission Workflows. This feature allows you to build sophisticated workflows made of chains of Fission functions. Here is a diagram that describes the architecture of Fission Workflows:

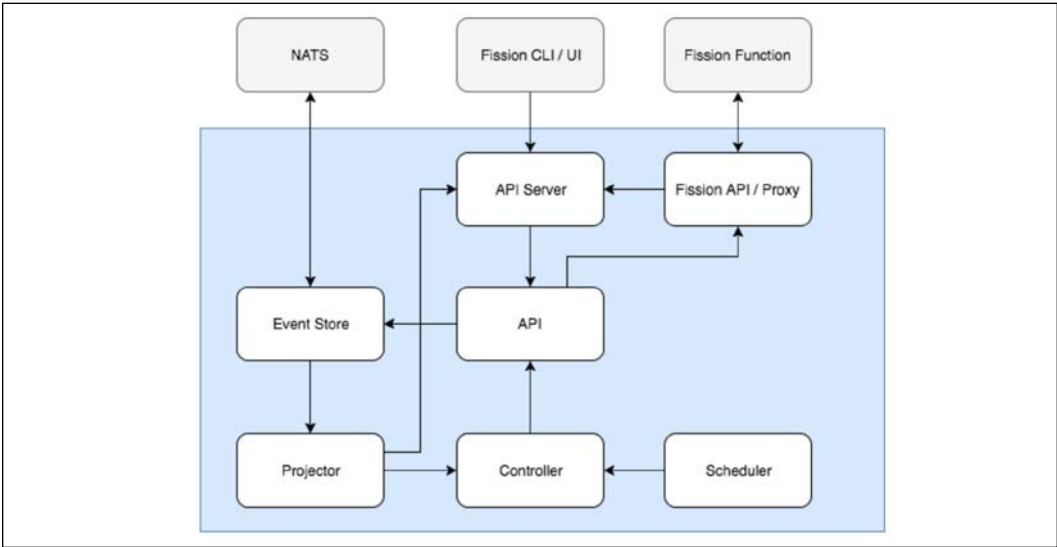


Figure 12.10: Fission workflow architecture

You define workflows in YAML that specify tasks (often Fission functions), inputs, outputs, conditions, and delays:

```

apiVersion: 1
description: Send a message to a slack channel when the temperature
exceeds a certain threshold
output: CreateResult
# Input: 'San Fransisco, CA'
tasks:
  # Fetch weather for input
  FetchWeather:
    run: wunderground-conditions
    inputs:
      default:
        apiKey: <API_KEY>
        state: "${$.Invocation.Inputs.default.substring($.Invocation.
Inputs.default.indexOf(',') + 1).trim()}"
        city: "${$.Invocation.Inputs.default.substring(0, $.Invocation.
Inputs.default.indexOf(',')).trim()}"

    ToCelsius:
      run: tempconv
      inputs:
        default:
          temperature: "${$.Tasks.FetchWeather.Output.current_observation.
temp_f}"
          format: F
          target: C
      requires:
        - FetchWeather

  # Send a slack message if the temperature threshold has been exceeded
  CheckTemperatureThreshold:
    run: if
    inputs:
      if: "${$.Tasks.ToCelsius.Output.temperature > 25}"
      then:
        run: slack-post-message
        inputs:
          default:
            message: "'It is ' + $.Tasks.ToCelsius.Output.temperature
+ 'C in ' + $.Invocation.Inputs.default + ' :fire:'}"
            path: <HOOK_URL>

```



```
requires:
- ToCelsius

# Besides the potential Slack message, compose the response of this
workflow {location, celsius, fahrenheit}
CreateResult:
run: compose
inputs:
  celsius: "${$.Tasks.ToCelsius.Output.temperature}"
  fahrenheit: "${$.Tasks.FetchWeather.Output.current_observation.
temp_f}"
  location: "${$.Invocation.Inputs.default}"
  sentSlackMsg: "${$.Tasks.CheckTemperatureThreshold.Output}"
requires:
- ToCelsius
- CheckTemperatureThreshold
```

Let's give Fission a try.

Experimenting with Fission

First, let's install Fission using Helm (Helm 3):

```
$ kubectl create ns fission
$ helm install fission --namespace fission \
  --set serviceType=NodePort,routerServiceType=NodePort \
  https://github.com/fission/fission/releases/download/1.9.0/fission-all-
1.9.0.tgz
```

Here are all the CRDs it created:

```
$ kubectl get crd -o name | grep fission
customresourcedefinition.apiextensions.k8s.io/canaryconfigs.fission.io
customresourcedefinition.apiextensions.k8s.io/environments.fission.io
customresourcedefinition.apiextensions.k8s.io/functions.fission.io
customresourcedefinition.apiextensions.k8s.io/httptriggers.fission.io
customresourcedefinition.apiextensions.k8s.io/kuberneteswatchtriggers.
fission.io
customresourcedefinition.apiextensions.k8s.io/messagequeuetriggers.fission.io
customresourcedefinition.apiextensions.k8s.io/packages.fission.io
customresourcedefinition.apiextensions.k8s.io/timetriggers.fission.io
```

The Fission CLI will come in handy too:

```
$ curl -Lo fission https://github.com/fission/fission/releases/download/1.9.0/fission-cli-osx && chmod +x fission && sudo mv fission /usr/local/bin/
```

We need to create an environment to be able to build our function. Let's go with a Python environment:

```
$ fission environment create --name python --image fission/python-env
environment 'python' created
```

With a Python environment in place, we can create a serverless function. First, save this code to `yeah.py`:

```
def main():
    return 'Yeah, it works!!!'
```

Then, we create the Fission function called `yeah`:

```
$ fission function create --name yeah --env python --code yeah.py
Package 'yeah-b9d5d944-9c6e-4e67-81fb-96e047625b74' created
function 'yeah' created
```

We can test the function though the Fission CLI:

```
$ fission function test --name yeah
Yeah, it works!!!
```

The real deal is invoking it though an HTTP endpoint. We need to create a route for that:

```
$ fission route create --method GET --url /yeah --function yeah
```

With the route in place, we still need to export the `FISSION_ROUTER` environment variable:

```
$ export FISSION_ROUTER=$(minikube ip):$(kubectl -n fission get svc router
-o jsonpath='{...nodePort}')
```

With all the preliminaries out of the way, let's invoke our function via httpie:

```
$ http http://${FISSION_ROUTER}/yeah
HTTP/1.1 200 OK
Content-Length: 17
Content-Type: text/html; charset=utf-8
Date: Wed, 10 Jun 2020 01:16:51 GMT
```

Yeah, it works!!!

Kubeless

Kubeless is another successful Kubernetes FaaS framework. It uses Kubernetes for autoscaling, routing, monitoring, and so on. Its claim to fame is bringing the most Kubernetes-native FaaS framework, along with its great UI. Kubeless models the world using similar concepts to Fission. Let's explore its architecture.

Kubeless architecture

Kubeless maintains a Kubernetes deployment and service for each function. It doesn't scale to zero, but as a result has a very fast response time. It is based on three pillars: runtimes, functions, and triggers. Let's examine them.

Kubeless runtimes

A Kubeless runtime is basically an image for each supported language that the Kubeless controller manager launches when a new function is created. The controller is watching the function CRD and if it changes, it dynamically reloads the code.

Kubeless can tell us exactly what the supported runtimes are:

```
$ kubeless get-server-config
INFO[0000] Current Server Config:
INFO[0000] Supported Runtimes are: ballerina0.981.0, dotnetcore2.0,
dotnetcore2.1, go1.10, go1.11, go1.12, java1.8, java11, nodejs6, nodejs8,
nodejs10, nodejs12, php7.2, php7.3, python2.7, python3.4, python3.6,
python3.7, ruby2.3, ruby2.4, ruby2.5, ruby2.6, jvm1.8, nodejs_distroless8,
nodejsCE8, vertx1.8
```

Kubeless functions

The Kubeless function CRD actually contains the source code for dynamic languages. When the Kubeless controller manager detects that a new function has been created, it will create a deployment and service for the function. It will also update the deployment if the function ever changes. The function can then be triggered via HTTP or events.

It is also possible to pre-build function images. This offers some performance benefits when redeploying the same function multiple times.

Kubeless triggers

Kubeless functions can be triggered (invoked) in multiple ways. You can directly invoke them from the CLI or the UI during development, which is very nice. But the real deal is triggering the functions in production. Similar to other frameworks, you can invoke functions via HTTP endpoints (after all, they are deployed as Kubernetes services). You will need to expose the service to the outside world yourself though.

Kubeless also supports triggering based on event sources. Current event sources include Kafka, NATS, and AWS Kinesis.

It's time to get hands-on with Kubeless.

Playing with Kubeless

Let's install the CLI first via brew:

```
$ brew install kubeless
```

The Helm charts for Kubeless are broken at the moment. It may be fixed by the time you read this. We will install it directly:

```
$ export RELEASE=$(curl -s https://api.github.com/repos/kubeless/kubeless/
releases/latest | grep tag_name | cut -d '"' -f 4)
$ kubectl create ns kubeless
$ kubectl create -f https://github.com/kubeless/kubeless/releases/
download/$RELEASE/kubeless-$RELEASE.yaml
configmap/kubeless-config created
deployment.apps/kubeless-controller-manager created
serviceaccount/controller-acct created
clusterrole.rbac.authorization.k8s.io/kubeless-controller-deployer created
clusterrolebinding.rbac.authorization.k8s.io/kubeless-controller-deployer
created
customresourcedefinition.apiextensions.k8s.io/functions.kubeless.io created
customresourcedefinition.apiextensions.k8s.io/httptriggers.kubeless.io
created
customresourcedefinition.apiextensions.k8s.io/cronjobtriggers.kubeless.io
created
```

We can verify whether Kubeless was installed properly by checking the kubeless-controller-manager (which should be ready):

```
$ kubectl get deploy kubeless-controller-manager -n kubeless
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
kubeless-controller-manager	1/1	1	1	11h

Alright, let's create a function. Here is our test Python function. Note the signature that includes an event object and a context object. The request body is always in the data field of the event object regardless of how the function was invoked and which programming language was used. In Python, the type of event['data'] is bytes and not a string. I learned it the hard way when I tried to concatenate the string 'Yeah, ' with event['data']. I changed it to a bytes type b'Yeah, ' and all was well:

```
def yeah(event, context):
    print(event)
    print(context)
    return b'Yeah, ' + event['data']
```

We can deploy it to the cluster with the kubeless function deploy command:

```
$ kubeless function deploy yeah --runtime python3.7 \
                                --from-file yeah.py \
                                --handler yeah.yeah
```

```
INFO[0000] Deploying function...
INFO[0000] Function yeah submitted for deployment
INFO[0000] Check the deployment status executing 'kubeless function ls
yeah'
```

After a while, the function will be ready:

```
$ kubeless function ls
```

NAME	NAMESPACE	HANDLER	RUNTIME	DEPENDENCIES	STATUS
yeah	default	yeah.yeah	python3.7		1/1 READY

Now, we can invoke it from the kubeless CLI:

```
$ kubeless function call yeah --data 'it works!!!'
Yeah, it works!!!
```

We can check the logs too and see the entire event and context that we print inside the function:

```
$ kubeless function logs yeah
{'data': 'it works!!!', 'event-id': 'cUo0tQDb5bt5V88', 'event-type':
'application/x-www-form-urlencoded', 'event-time': '2019-12-27T17:05:16Z',
'event-namespace': 'cli.kubeless.io', 'extensions': {'request':
<LocalRequest: POST http://192.168.64.3:8443/>}}
{'function-name': <function yeah at 0x7f07b3a5a7b8>, 'timeout': 180.0,
'runtime': 'python3.7', 'memory-limit': '0'}
```

Using the Kubeless UI

Let's check out the famous Kubeless web UI:

```
$ kubectl create -f https://raw.githubusercontent.com/kubeless/kubeless-ui/
master/k8s.yaml
serviceaccount/ui-acct created
clusterrole.rbac.authorization.k8s.io/kubeless-ui created
clusterrolebinding.rbac.authorization.k8s.io/kubeless-ui created
deployment.apps/ui created
service/ui created
```

The service was deployed in the kubeless namespace. We can use port-forward to expose it:

```
$ kubectl get svc -n kubeless
NAME      TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
ui        NodePort    10.96.248.15    <none>           3000:32079/TCP   93s
$ k port-forward -n kubeless svc/ui 3000
Forwarding from 127.0.0.1:3000 -> 3000
Forwarding from [::1]:3000 -> 3000
```

It is indeed a very user-friendly and convenient UI:

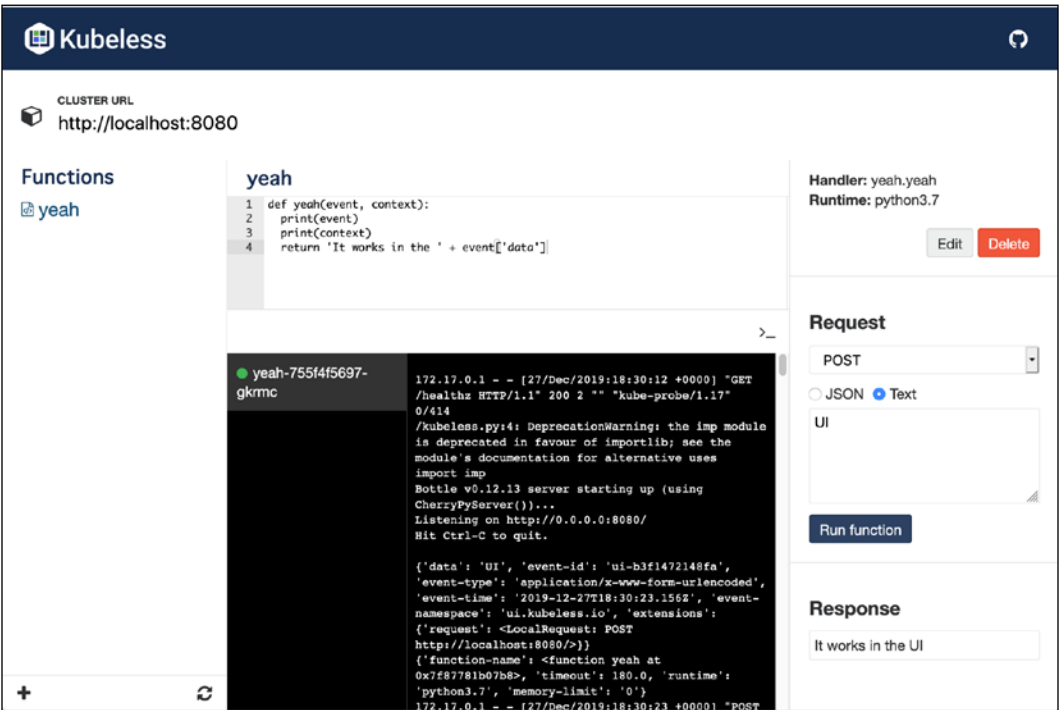


Figure 12.11: Kubeless web UI

You can see all your functions, run them with different parameters, and see the responses and the logs. You can even edit and save your changes all from the comfort of the web UI. I discovered that when passing event data through the event UI, you must use POST and not the default GET, and also that the data arrives as a string and not as bytes, like it did when invoking the function through the CLI. This is inconsistent and annoying. It means that the function can't assume the type of the event data and must handle both cases. Overall, however, the Kubeless UI definitely lives up to its reputation.

Kubeless with the serverless framework

You can also work with Kubeless using the serverless framework. For more details, check out the guide at <https://serverless.com/framework/docs/providers/kubeless/guide/intro/>.

Knative and riff

Riff is an open source project from Pivotal for running functions on Kubernetes. Its recursive name stands for "riff is for functions". Riff is built on top of Knative and enjoys all its benefits. It adds a CLI and the ability to build the function containers for you via another open source project from Heroku and Pivotal Cloud-native Buildpacks (<https://buildpacks.io/>).

Riff is used to power PKS – Pivotal's enterprise Kubernetes offering. That should give you a lot of confidence that it is robust and battle tested.

Understanding riff runtimes

Riff supports different runtimes. A riff runtime is unlike a Kubeless runtime or a Fission environment. It is more about the underlying implementation that riff uses to implement its FaaS capabilities. There are three different runtimes:

- Core
- Knative
- Streaming

The core runtime uses vanilla Kubernetes objects to create a Deployment and a Service. No ingress or autoscaling is provided. It doesn't provide much value, really.

The Knative runtime uses Knative (obviously) and it also depends on Istio (so, no Knative and Gloo if you're using riff).

Installing riff with Helm 2

At the moment the riff Helm chart is not compatible with Helm 3. Let's use Helm 2 to install riff. First, we'll add the project riff charts to our repo list:

```
$ helm2 repo add projectriff https://projectriff.storage.googleapis.com/
charts/releases
$ helm2 repo update
```

Next, we'll install Istio (specifically, a version provided by project riff itself to ensure compatibility):

```
$ helm2 install projectriff/istio --name istio -n istio-system --set
gateways.istio-ingressgateway.type=NodePort --wait --devel
```


Finally, let's install riff itself (make sure to first delete the knative-serving namespace if it exists):

```
$ helm2 install projectriff/riff --name riff \
  --set tags.core-runtime=true \
  --set tags.knative-runtime=true \
  --set tags.streaming-runtime=false \
  --wait --devel
```

Now that riff is installed in our cluster, we can install the riff CLI. On macOS, we can use homebrew:

```
$ brew install riff
```

You can find instructions for other operating systems at <https://github.com/projectriff/cli/#installation-of-the-latest-release>.

We can verify the status of the riff installation with the `riff doctor` command:

```
$ riff doctor
```

```
NAMESPACE    STATUS
riff-system   ok
```

RESOURCE	READ	WRITE
configmaps	allowed	allowed
secrets	allowed	allowed
Pods	allowed	n/a
Pods/log	allowed	n/a
applications.build.projectriff.io	allowed	allowed
containers.build.projectriff.io	allowed	allowed
functions.build.projectriff.io	allowed	allowed
deployers.core.projectriff.io	allowed	allowed
processors.streaming.projectriff.io	missing	missing
streams.streaming.projectriff.io	missing	missing
adapters.knative.projectriff.io	allowed	allowed
deployers.knative.projectriff.io	allowed	allowed

The status is OK and all the components are installed, except the two streaming components we didn't install on purpose (as they require Kafka).

Functions will be packaged as images and stored in a container registry. So, we need to give riff credentials:

```
$ export DOCKER_ID=g1g1
$ riff credential apply dockerhub-creds --docker-hub $DOCKER_ID --set-
default-image-prefix
```

Be sure to use your own DockerHub ID (not mine obviously). Riff will ask you for your password.

Moving on, it's time to create the function. Riff requires a little more ceremony than other FaaS frameworks. The function code must be in a Git repository. Let's use a simple Node.js square function from the riff project:

```
module.exports = x => {
  const xx = x ** 2;
  console.log('the square of ${x} is ${xx}');
  return xx;
}
```

Here is how to create a function with riff:

```
$ riff function create square \
  --git-repo https://github.com/projectriff-samples/node-square \
  --artifact square.js \
```

Created function "square"

The next step is to create a deployer:

```
$ riff knative deployer create knative-square --function-ref square -tail
Created deployer "knative-square"
```

To invoke the function, we need to get the proper IP address and node port of the Istio ingress gateway from Minikube. In a cluster that supports public IP addresses, it will be available through a proper DNS name:

```
MINIKUBE_IP=$(minikube ip)
INGRESS_PORT=$(kubectl get svc istio-ingressgateway --namespace istio-
system --output 'jsonpath={.spec.ports[?(@.port==80)].nodePort}')
```

Now we can invoke the function:

```
$ curl http://$MINIKUBE_IP:$INGRESS_PORT/ -w '\n' \
-H 'Host: knative-square.default.example.com' \
-H 'Content-Type: application/json' \
-d 12
```

144

The bottom line is that riff feels a little rough. It provides a CLI and a way to build images from code, but you still have to configure a lot of things and go through a GitHub repo and container registry. It doesn't feel very agile and there is definitely no UI. For event-based triggering, you need to install the streaming runtime separately and it supports only Kafka, which is not everyone's cup of tea.

Summary

In this chapter, we covered the hot topic of serverless computing. We explained the two meanings of serverless – eliminating the need to manage servers, and deploying and running FaaS. We explored in depth the aspects of serverless infrastructure in the cloud, especially in the context of Kubernetes. We compared the built-in cluster autoscaler as a Kubernetes-native serverless solution to the offerings of cloud providers such as AWS EKS+Fargate, Azure AKS+ACI, and Google Cloud Run. We switched gears and dove into the exciting and promising Knative project with its scale-to-zero capabilities and advanced deployment options. Then, we moved to the wild world of FaaS on Kubernetes. We mentioned the plethora of solutions out there and examined them in detail with hands-on experiments for some of the prominent solutions out there: Fission, Kubeless, and riff. The bottom line is that both flavors of serverless computing bring real benefits as far as operations and cost management are concerned. It's going to be fascinating to watch the evolution and consolidation of these technologies in the cloud and Kubernetes.

In the next chapter, our focus will be on monitoring and observability. Complex systems such as large Kubernetes clusters, with lots of different workloads and continuous-delivery pipeline and configuration changes, must have excellent monitoring in place in order to keep all the balls up in the air. Kubernetes has some great options that we should take advantage of.

13

Monitoring Kubernetes Clusters

In the previous chapter, we looked at serverless computing and its manifestations on Kubernetes. A lot of innovation happens in this space and it's both super useful and fascinating to follow the evolution.

In this chapter, we're going to talk about how to make sure your systems are up and running, performing correctly, and how to respond to them when they aren't. In *Chapter 3, High Availability and Reliability*, we discussed related topics. The focus here is about knowing what's going on in your system and what practices and tools you can use.

There are many aspects to monitoring, such as logging, metrics, distributed tracing, error reporting, and alerting. Practices like auto-scaling and self-healing depend on monitoring to detect that there is a need to scale or to heal.

The topics we will cover in this chapter include:

- Understanding observability
- Logging with Kubernetes
- Recording metrics with Kubernetes
- Distributed tracing with Jaeger
- Troubleshooting problems

The Kubernetes community recognizes the importance of monitoring and has put a lot of effort into making sure Kubernetes has a solid monitoring story. The **Cloud Native Computing Foundation (CNCF)** is the de facto curator of cloud native infrastructure projects. It's graduated eight projects so far (early 2020). Kubernetes was the first project to graduate and out of the other seven, three projects focus on monitoring: Prometheus, Fluentd, and Jaeger. Before we dive into the ins and outs of Kubernetes monitoring and specific projects and tools, we should get a better understanding of what monitoring is all about. A good framework for thinking about monitoring is how observable your system is. Indeed, observability is another term that people flaunt about these days.

Understanding observability

Observability is a big word. What does it mean in practice? There are different definitions out there and big debates regarding how monitoring and observability are similar and different. I take the stance that observability is the property of the system that defines what we can tell about the state and behavior of the system, right now and historically. In particular, we are interested in the health of the system and its components. Monitoring is the collection of tolls, processes, and techniques we use to increase the observability of the system.

There are different facets of information that we need to collect, record, and aggregate in order to get a good sense of what our system is doing. Those facets include logs, metrics, distributed traces, and errors. The monitoring or observability data is multi-dimensional and crosses many levels. Just collecting it doesn't help much. We need to be able to query it, visualize it, and alert other systems when things go wrong. Let's review the various components of observability.

Logging

Logging is a key monitoring tool. Every self-respecting long-running software must have logs. Logs capture timestamped events. They are critical for many applications, like business intelligence, security, compliance, audits, debugging, and troubleshooting. It's important to understand that a complicated distributed system will have different logs for different components, and extracting insights from logs is not a trivial undertaking.

There are several key attributes to logs: format, storage, and aggregation.

Log format

Logs may come in various formats. Plain text is very common and human-readable but requires a lot of work to parse and merge with other logs. Structured logs are more suitable for large systems because they can be processed at scale. Binary logs make sense for systems that generate a lot of logs as they are more space efficient, but they require custom tools and processing to extract their information.

Log storage

Logs can be stored in memory, on the filesystem, in a database, in cloud storage, sent to remote logging, or any combination of those options. In the cloud-native world, where software runs in containers, it's important to know where logs are stored and how to fetch them when necessary.

Questions as to durability come to mind when containers come and go. On Kubernetes, the standard output and error of containers is automatically logged and available, even when the pod terminates. However, issues such as having enough space for logs and log rotation are always relevant.

Log aggregation

In the end, the best practice when sending local logs to a centralized logging service that is designed to handle various log formats is to persist them, as necessary, and aggregate many types of logs in a way that they can be queried and reasoned about.

Metrics

Metrics measure the same aspect of the system over time. Metrics are time series of numerical values (typically, floating-point numbers). Each metric has a name and often a set of labels that help later in slicing and dicing. For example, the CPU utilization of a node or the error rate of a service are metrics.

Metrics are much more economical than logs. They require a fixed amount of space that doesn't ebb and flow with incoming traffic like logs.

Also, since metrics are numerical in nature, they don't need parsing and transformations and can be easily combined, analyzed using statistical methods, and used to serve as triggers for events and alerts.

A lot of metrics at different levels (node, container, process, networks, disk) are often collected for you automatically by the OS, cloud provider, or Kubernetes.

However, you can also create custom metrics that map to high-level concerns of your system and can be configured with application-level policies.

Distributed tracing

Modern distributed systems often use a microservice-based architecture where an incoming request is bounced between multiple microservices, waits in queues, and triggers serverless functions. When you try to analyze errors, failures, data integrity issues, or performance issues, it is critical to be able to follow the path of a request. This is where distributed tracing comes in.

A distributed trace is a collection of spans and references. You can think of a trace as a **directed acyclic graph (DAG)** that represents a request traversal through the components of a distributed system. Each span records the time the request spent in a given component, while references are the edges of the graph that connect one span to the following spans.

Here is an example:

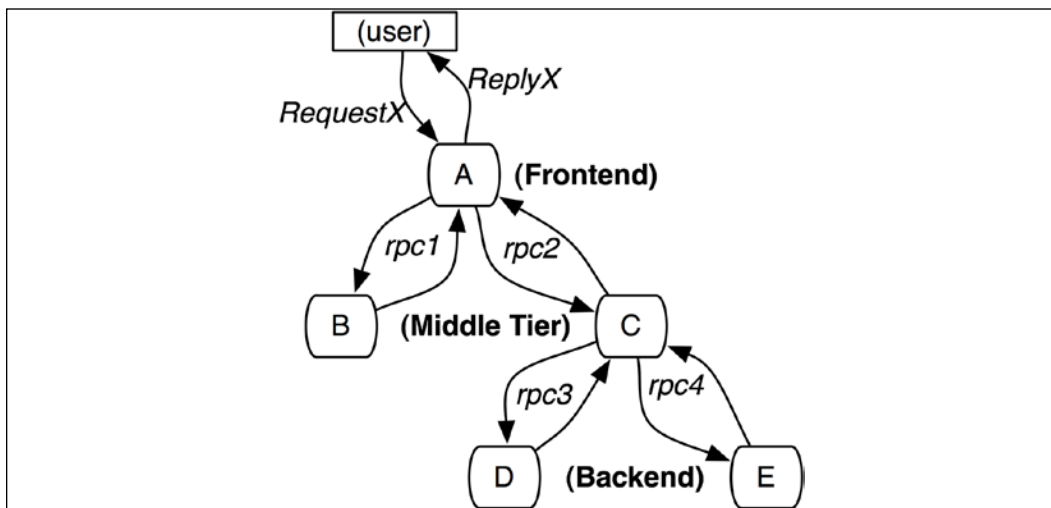


Figure 13.1: A Directed Acyclic Graph (DAG)

Distributed tracing is indispensable for understanding complex distributed systems.

Application error reporting

Error and exception reporting are sometimes done as part of logging. You definitely want to log errors and looking at logs when things go wrong is a time-honored tradition. However, there are levels for capturing error information that go beyond logging. When an error occurs in one of your applications, it is useful to capture an error message, the location of the error in the code, and the stack trace. This is pretty standard, and most programming languages can provide all this information, although stack traces are multi-line and don't fit well with line-based logs. A very useful additional piece of information is capturing the local state in each level of the stack trace.

A central error reporting service such as Sentry or Rollbar provides a lot of value specific to errors beyond logging, such as rich error information and context and user information.

Dashboards and visualization

OK. You've done a great job of collecting logs, defining metrics, tracing your requests, and reporting rich errors. Now, you want to figure out what your system or parts of it are doing. What is the baseline? How does traffic fluctuate throughout the day, week, and on holidays? When the system is under stress, what parts are the most vulnerable?

In a complicated system that involves hundreds and thousands of services, data stores, and integrates with external systems, you can just look at the raw log files, metrics, and traces.

You need to be able combine a lot of information and build system health dashboards, visualize your infrastructure, and create business-level reports and diagrams.

You may get some of it (especially for infrastructure) automatically if you're using cloud platforms. However, you should expect to do some serious work around visualization and dashboards.

Alerting

Dashboards are great for humans that want to get a broad view of the system and be able to drill down and understand how it behaves. Alerting is all about detecting abnormal situations and triggering some action. Ideally, your system should be self-healing and be able to recover on its own from most situations. However, you should at least report it so humans can review what happened at their leisure and decide if further action is needed.

Alerting can be integrated with emails, chat rooms, and on call systems. It is often linked to metrics and when certain conditions apply, an alert is raised.

Now that we've covered the different elements involved in monitoring complex systems, let's see how to do it with Kubernetes.

Logging with Kubernetes

We need to carefully consider our logging strategy with Kubernetes. There are several types of logs that are relevant for monitoring purposes. Our workloads run in containers, of course, and we care about these logs, but we also care about the logs of Kubernetes components such as kubelets and the container runtime. In addition, chasing logs across multiple nodes and containers is a non-starter. The best practice is to use central logging (also known as log aggregation). There are several options here that we will explore soon.

Container logs

Kubernetes stores the standard output and standard error of every container. They are made available through the `kubectl logs` command.

Here is a pod manifest that prints the current date and time every 10 seconds:

```
apiVersion: v1
kind: Pod
metadata:
  name: now
spec:
  containers:
  - name: now
    image: g1g1/py-kube:0.2
    command: ["/bin/bash", "-c", "while true; do sleep 10; date;
done"]
```

We can save it to a file called `now-pod.yaml` and create it:

```
$ kubectl apply -f now-pod.yaml
pod/now created
```

Wait until the pod is ready. To check out the logs, we use the `kubectl logs` command:

```
$ kubectl logs now
Thu Jun 11 00:32:38 UTC 2020
Thu Jun 11 00:32:48 UTC 2020
Thu Jun 11 00:32:58 UTC 2020
Thu Jun 11 00:33:08 UTC 2020
Thu Jun 11 00:33:18 UTC 2020
```

A few points about container logs. `kubectl logs` expects a pod name. If the pod has multiple containers, you need to specify the container name too:

```
$ kubectl logs <pod name> -c <container name>
```

Also, if a deployment or replica set creates multiple copies of the same pod, you still have to query each pod independently for its logs. There is no way to get the logs of multiple pods in a single call.

If a container crashes for some reason, you can use the `kubectl logs -p` command to look at logs from the container.

Kubernetes component logs

If you run Kubernetes in a managed environment like GKE, EKS, or AKS, you won't be able to access Kubernetes component logs, but this is expected. You're not responsible for the Kubernetes control plane. There are Kubernetes components that run on master nodes and there are components that run on each worker node:

Here are the master components and their log locations:

- **API server:** `/var/log/kube-apiserver.log`
- **Scheduler:** `/var/log/kube-scheduler.log`
- **Controller manager:** `/var/log/kube-controller-manager.log`

The worker node components and their log locations are:

- **Kubelet:** `/var/log/kubelet.log`
- **Kube proxy:** `/var/log/kube-proxy.log`

Note that on systemd-based systems, you'll need to use `journalctl` to view the logs.

Centralized logging

Reading container logs is fine for quick and dirty troubleshooting problems in a single pod. To diagnose and debug system-wide issues, we need centralized logging (also known as log aggregation). All the logs from our containers should be sent to a central repository and made accessible for slicing and dicing using filters and queries.

When deciding on your central logging approach, there are several important decisions: How do we collect the logs? Where do we store the logs? And how do we handle sensitive log information?

Choosing a log collection strategy

Logs are typically collected by agents that are running in close proximity to the process generating the logs. They make sure to deliver them to the central logging service.

Here are the common approaches.

Direct logging

In this approach, there is no log agent. It is the responsibility of each application container to send logs to the remote logging service. This is typically done through a client library. It is a high-touch approach and applications need to be aware of the logging target, as well as being configured with proper credentials. If you ever want to change your log collection strategy, you will need to make changes to each and every application (at the least, bumping to a new version of the library):

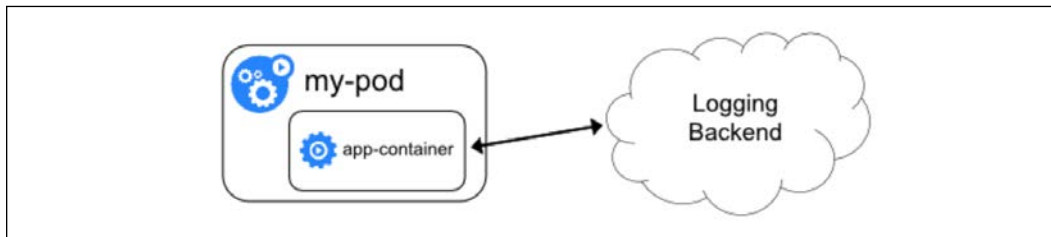


Figure 13.2: Direct logging

Node agent

The node agent approach is best when you control the worker nodes and you want to abstract away the act of log aggregation from your applications. Each application container can simply write to standard output and standard error, and the agent running on each node will intercept the logs and deliver them to the remote logging service.

Typically, you deploy the node agent as a DaemonSet, so as nodes are added or removed from the cluster, the log agent will always be present, without any additional work being needed:

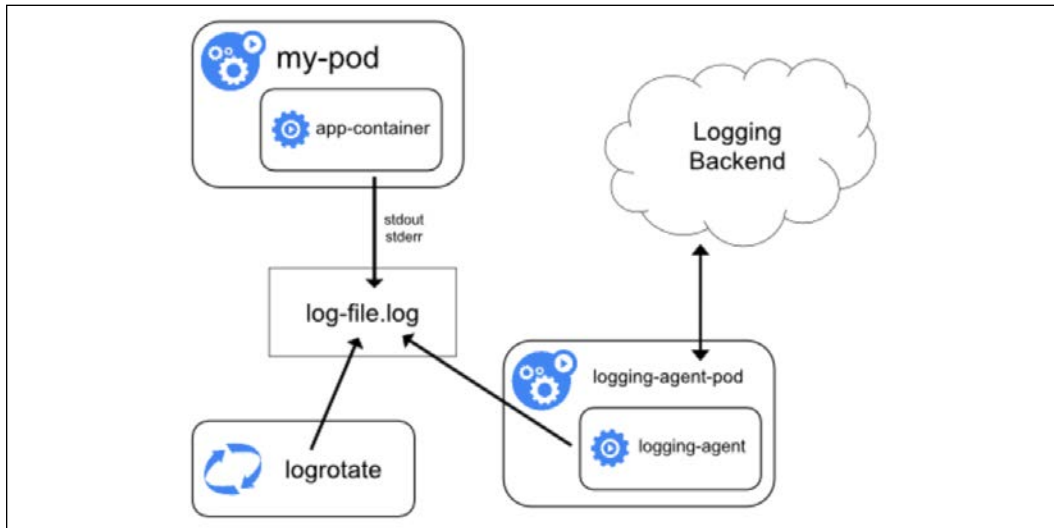


Figure 13.3: Logging with a node agent

Sidecar container

The sidecar container is best when you don't have control over your cluster nodes or if you use some serverless computing infrastructure to deploy containers, but you don't want to use the direct logging approach. The node agent approach is out of the question, but you can attach a sidecar container that will collect the logs and deliver them to the central logging service. It is not as efficient as the node agent approach because each container will need its own logging sidecar container, but it can be done at the deployment stage without requiring code changes and application knowledge:

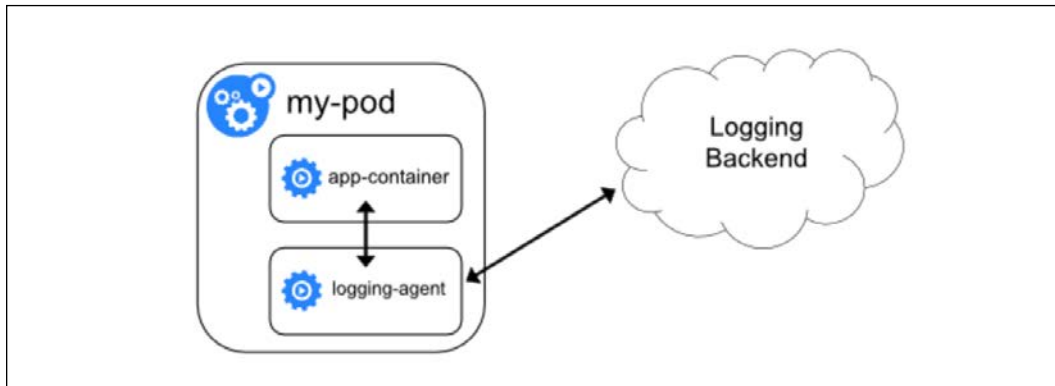


Figure 13.4: Logging with a sidecar container

Now that we've covered the topic of log collection, let's consider how to store and manage those logs centrally.

Cluster-level central logging

If your entire system is running in a single Kubernetes cluster, then cluster-level logging may be a great choice. You can install a central logging service like Grafana Loki, Elasticsearch, or Graylog in your cluster and enjoy a cohesive log aggregation experience without sending your log data elsewhere.

However, for in-cluster central logging, this is not always possible or desirable.

Remote central logging

There are use cases where in-cluster central logging doesn't cut it for various reasons:

- Logs are used for audit purposes, so it may be necessary to log to a separate and controlled location (for example, on AWS, it is common to log to a separate account)
- Your system runs on multiple clusters and logging in each cluster is not really central
- You run on a cloud provider and prefer to log into the cloud platform logging service (for example, StackDriver on GCP or CloudWatch on AWS)
- You already work with a remote central logging service like SumoLogic or Splunk and you prefer to continue using them
- You just don't want the hassle of collecting and storing log data

Dealing with sensitive log information

OK. We can collect the logs and send them to a central logging service. If the central logging service is remote, you might need to be selective about which information you log.

For example, **personally identifiable information (PII)** and **protected health information (PHI)** are two categories of information that you probably shouldn't log without making sure access to the log is properly controlled.

At Helix, for example, we redact PII like usernames and emails.

Using Fluentd for log collection

Fluentd (<https://www.fluentd.org/>) is an open source CNCF graduated project. It is considered best in class on Kubernetes and it can integrate with pretty much every logging backend you want. If you want to set up your own centralized logging solution, I recommend using Fluentd. The following diagram shows how Fluentd can be deployed as a DaemonSet in a Kubernetes cluster:

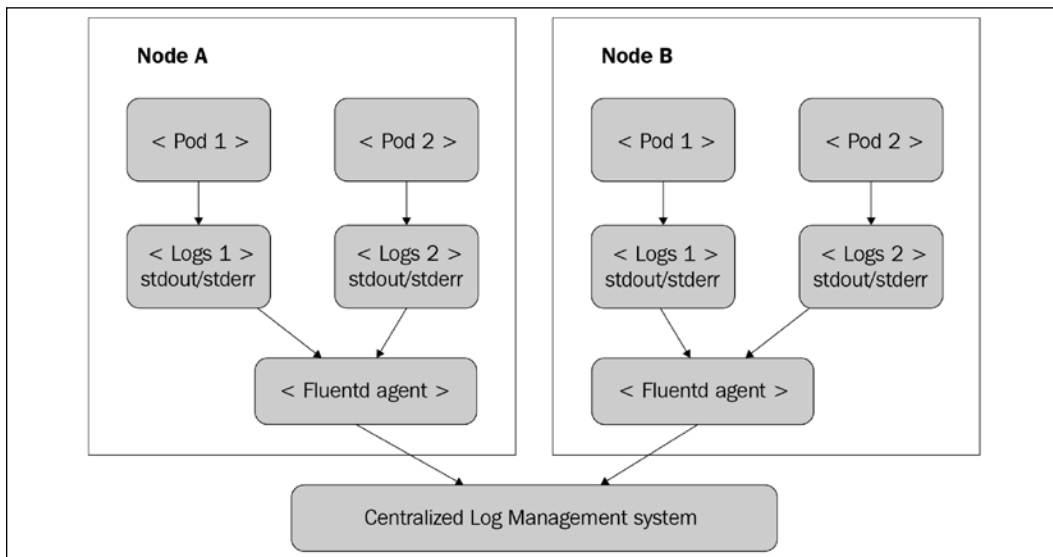


Figure 13.5: Fluentd as a DaemonSet in a Kubernetes cluster

One of the most popular DIY centralized logging solutions is ELK, where E stands for ElasticSearch, L stands for LogStash, and K stands for Kibana. On Kubernetes EFK, where Fluentd replaces LogStash, this is very common, and there are Helm charts and a lot of examples available for deploying and operating it on Kubernetes.

Fluentd has a plugin-based architecture, so don't feel limited to EFK. Fluentd doesn't require a lot of resources, but if you really need a high-performance solution, **Fluentbit** (<https://fluentbit.io/>) is a pure forwarder that uses barely 450 KB of memory.

Collecting metrics with Kubernetes

If you have some experience with Kubernetes, you may be familiar with **cAdvisor** and **Heapster**. cAdvisor was integrated into the kube-proxy until Kubernetes 1.12 and then it was removed. Heapster was removed in Kubernetes 1.13. If you wish, you can install them, but they are not recommended anymore as there are much better solutions now.

One caveat is that the Kubernetes dashboard v1 still depends on Heapster. The Kubernetes dashboard v2 is still in Beta at the time of writing. Hopefully, it will be generally available by the time you read this.

Kubernetes now has a Metrics API. It supports node and pod metrics out of the box. You can also define your own custom metrics.

A metric contains a timestamp, a usage field, and the time range the metric was collected (many metrics are accumulated over a time period). Here is the API definition for node metrics:

```
type NodeMetrics struct {
    metav1.TypeMeta
    metav1.ObjectMeta

    Timestamp metav1.Time
    Window    metav1.Duration

    Usage corev1.ResourceList
}

// NodeMetricsList is a list of NodeMetrics.
type NodeMetricsList struct {
    metav1.TypeMeta
    // Standard list metadata.
    // More info: https://git.k8s.io/community/contributors/devel/sig-
architecture/api-conventions.md#types-kinds
    metav1.ListMeta

    // List of node metrics.
    Items []NodeMetrics
}
```

The usage field type is `ResourceList`, but it's actually a map of a resource name to a quantity:

```
// ResourceList is a set of (resource name, quantity) pairs.
type ResourceList map[ResourceName]resource.Quantity
```

`Quantity` (<https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/apimachinery/pkg/api/resource/quantity.go#L88>) is a fixed-point representation of a number. It provides convenient marshaling/unmarshaling in JSON and YAML, as well as `String()` and `Int64()` accessors:

```
type Quantity struct {
    // i is the quantity in int64 scaled form, if d.Dec == nil
    i int64Amount

    // d is the quantity in inf.Dec form if d.Dec != nil
    d infDecAmount

    // s is the generated value of this quantity to avoid recalculation
    s string

    // Change Format at will. See the comment for Canonicalize for more
    // details.
    Format
}
```

Monitoring with the metrics server

The Kubernetes `metrics-server` implements the Kubernetes Metrics API.

You can deploy it with Helm 3:

```
helm install metrics-server bitnami/metrics-server --version 4.2.0 -n kube-system
```

On minikube, you enable it as an add-on:

```
$ minikube addons enable metrics-server
✔ metrics-server was successfully enabled
```

After waiting a few minutes to let the metrics server collect some data, you can query it using these commands for node metrics:

```
$ kubectl get --raw "/apis/metrics.k8s.io/v1beta1/nodes" | jq .
```



```
{
  "kind": "NodeMetricsList",
  "apiVersion": "metrics.k8s.io/v1beta1",
  "metadata": {
    "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes"
  },
  "items": [
    {
      "metadata": {
        "name": "ip-192-168-13-100.ec2.internal",
        "selfLink": "/apis/metrics.k8s.io/v1beta1/nodes/ip-192-168-13-100.
ec2.internal",
        "creationTimestamp": "2020-01-07T20:05:29Z"
      },
      "timestamp": "2020-01-07T20:04:54Z",
      "window": "30s",
      "usage": {
        "cpu": "85887417n",
        "memory": "885828Ki"
      }
    }
  ]
}
```

In addition, the `kubectl top` command gets its information from the metrics server:

```
$ kubectl top nodes
```

NAME	CPU(cores)	CPU%	MEMORY(bytes)	MEMORY%
ip-192-168-13-100.ec2.internal	85m	4%	863Mi	11%

```
$ kubectl top pods
```

NAME	CPU(cores)	MEMORY(bytes)
api-gateway-795f7dcbdb-ml2tm	1m	23Mi
link-db-7445d6cbf7-2zs2m	1m	32Mi
link-manager-54968ff8cf-q94pj	0m	4Mi
nats-cluster-1	1m	3Mi
nats-operator-55dfdc6868-fj5j2	2m	11Mi
news-manager-7f447f5c9f-c4pc4	0m	1Mi
news-manager-redis-0	1m	1Mi
social-graph-db-7565b59467-dmdlw	1m	31Mi

social-graph-manager-64cdf589c7-4bjcn	0m	1Mi
user-db-0	1m	32Mi
user-manager-699458447-6lwjq	1m	1Mi

The metrics server is also the source for performance information in the Kubernetes dashboard.

Exploring your cluster with the Kubernetes dashboard

The Kubernetes dashboard is a web application that you can install and then use to drill down to your cluster through a nice user interface. Depending on your Kubernetes distribution, it may or may not be installed. On minikube, you install it as an add-on:

```
$ minikube addons enable dashboard
✓ dashboard was successfully enabled
```

On other distributions, you can install it yourself:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.0.0-beta8/aio/deploy/recommended.yaml
```

I'm a big fan of the dashboard because it gives a very condensed view of your entire cluster, as well as the ability to drill down by namespace, resource type, or labels, and even perform a general search:

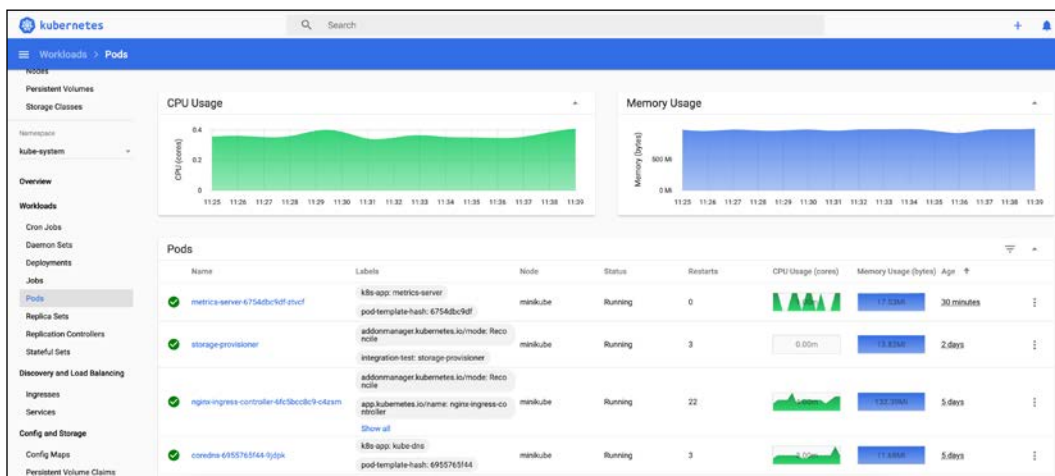


Figure 13.6: the Kubernetes dashboard

The rise of Prometheus

Prometheus (<https://prometheus.io/>) is yet another graduated CNCF open source project. It focuses on metrics collection and alert management. It has a simple yet powerful data model for managing time-series data and a sophisticated query language. It is considered best in class in the Kubernetes world. Prometheus lets you define recording rules that are fired at regular intervals and collect data from targets. In addition, you can define alerting rules that evaluate a condition and trigger alerts if the condition is satisfied.

It has several unique features compared to other monitoring solutions:

- The collection system is pull over HTTP. Nobody has to push metrics to Prometheus (but push is supported via a gateway).
- A multi-dimensional data model (each metric is a named time series with a set of key/value pairs attached to each data point).
- PromQL, a powerful and flexible query language to slice and dice your metrics.
- Prometheus server nodes are independent and don't rely on shared storage.
- Target discovery can be dynamic or done via static configuration.
- Built-in time series storage, but supports other backends if necessary.
- Built-in alert manager and ability to define alerting rules.

The following diagram illustrates the entire system:

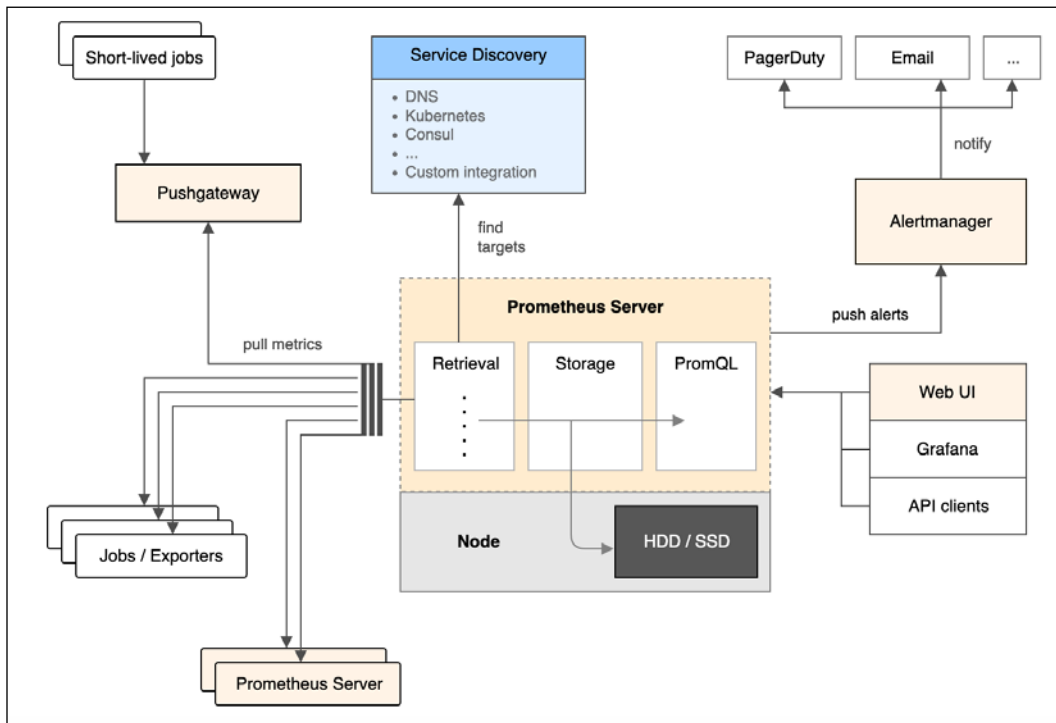


Figure 13.7: the Prometheus system

Installing Prometheus

Prometheus is a complex beast, as you can see. The best way to install it is using the **Prometheus operator** (<https://github.com/coreos/prometheus-operator>).

However, before you install it, make sure to delete the `knative-monitoring` namespace if you're using the same cluster that you installed knative on. Knative quietly installs its own Prometheus-based monitoring system into your cluster.

On minikube, it takes some extra steps and configuration to get ready for Prometheus (they should probably make it an add-on).

To prepare minikube for Prometheus, we need to start it with some extra arguments:

```
$ minikube start --memory=4096 \  
    --bootstrapper=kubeadm \  
    --extra-config=scheduler.address=0.0.0.0 \  
    --extra-config=controller-manager.address=0.0.0.0
```

The following article dives into the details: <https://medium.com/faun/trying-prometheus-operator-with-helm-minikube-b617a2dccfa3>.

There is a Helm chart that's managed by the community for the Prometheus operator, but it is incompatible with Helm 3 (it uses the dreaded `crd-install` hook). We will install it with Helm 2, which requires, as you may recall, installing Tiller too. If you have Tiller installed already from previous chapters, you can skip this step:

```
$ kubectl create serviceaccount tiller --namespace kube-system  
$ kubectl create clusterrolebinding tiller-role-binding \  
    --clusterrole cluster-admin --serviceaccount=kube-system:tiller  
$ helm2 init --service-account tiller
```

Now, we can install the Prometheus operator. This may take a few minutes, so don't be alarmed if it appears to just hang there. The `minikube_values.yaml` file can be found in the `prometheus` sub-directory of the code directory:

```
$ helm2 install stable/prometheus-operator \  
    --version=8.5.4 \  
    --name monitoring \  
    --namespace monitoring \  
    --values=minikube_values.yaml
```

The Helm chart installs a comprehensive metric-based monitoring stack with quite a few components:

- prometheus-operator
- prometheus
- alertmanager
- node-exporter
- kube-state-metrics
- grafana

Check out the pods installed in the monitoring namespace. It should look something like:

```
$ kubectl get po -n monitoring
```

NAME	READY	STATUS
alertmanager-monitoring-prometheus-oper-alertmanager-0	2/2	Running
0 15m		
monitoring-grafana-697fd7b5cc-2rgmq	2/2	Running
0 15m		
monitoring-kube-state-metrics-574ccf8cd6-ng2mq	1/1	Running
0 15m		
monitoring-prometheus-node-exporter-pgnj8	1/1	Running
0 15m		
monitoring-prometheus-oper-operator-74d96f6ffb-r5zt7	2/2	Running
15m 0		
prometheus-monitoring-prometheus-oper-prometheus-0	3/3	Running
1 15m		

The Prometheus operator manages Prometheus and its Alertmanager through four CRDs:

- Prometheus - ServiceMonitor - PrometheusRule - AlertManager

If you want a more complete and opinionated installation experience, check out **kube-prometheus** (<https://github.com/coreos/kube-prometheus>). It installs Prometheus and the AlertManager using a high-availability configuration, as well as additional tools and default rules and a dashboard. It even has its own Metrics API server, so you don't need to enable the metrics-server add-on in minikube.

Let's examine Prometheus and the other components.

Interacting with Prometheus

Prometheus has a basic web UI that you can use to explore its metrics. Let's do port forwarding to localhost:

```
$ POD_NAME=$(kubectl get pods -n monitoring -l "app=prometheus" \
-o jsonpath="{.items[0].metadata.name}")
$ kubectl port-forward -n monitoring $POD_NAME 9090
```

Then, you can browse to <http://localhost:9090>, where you can select different metrics and view raw data or graphs:



Figure 13.8: Prometheus UI

Prometheus records an outstanding number of metrics (990, in my current setup). The most relevant metrics on Kubernetes are the metrics exposed by kube-state-metrics and node exporters.

Incorporating kube-state-metrics

The Prometheus operator already installs kube-state-metrics. It is a service that listens to Kubernetes events and exposes them through a `/metrics` HTTP endpoint in the format that Prometheus expects. So, it is a Prometheus exporter.

This is very different from the Kubernetes metrics server, which is the standard way Kubernetes exposes metrics for nodes and pods and allows you to expose your own custom metrics too. The Kubernetes metrics server is a service that periodically queries Kubernetes for data and stores it in memory. It exposes its data through the Kubernetes Metrics API.

The metrics exposed by kube-state-metrics are vast. Here is the list of groups of metrics, which is pretty massive on its own. Each group corresponds to a Kubernetes API object and contains multiple metrics:

- CertificateSigningRequest metrics
- ConfigMap metrics
- CronJob metrics
- DaemonSet metrics
- Deployment metrics
- Endpoint metrics
- Horizontal Pod Autoscaler metrics
- Ingress metrics
- Job metrics
- LimitRange metrics
- MutatingWebhookConfiguration metrics
- Namespace metrics
- NetworkPolicy metrics
- Node metrics
- PersistentVolume metrics
- PersistentVolumeClaim metrics
- Pod Disruption Budget metrics
- Pod metrics
- ReplicaSet metrics
- ReplicationController metrics
- ResourceQuota metrics

- Secret metrics
- Service metrics
- StatefulSet metrics
- StorageClass metrics
- ValidatingWebhookConfiguration metrics
- VerticalPodAutoscaler metrics
- VolumeAttachment metrics

For example, here are the metrics collected for Kubernetes services:

- kube_service_info
- kube_service_labels
- kube_service_created
- kube_service_spec_type
- kube_service_spec_external_ip
- kube_service_status_load_balancer_ingress

Utilizing the node exporter

kube-state-metrics collects node information from the Kubernetes API server, but this information is pretty limited. Prometheus comes with its own node exporter, which collects tons of low-level information about the nodes. Remember that Prometheus may be the de facto standard metrics platform on Kubernetes, but it is not Kubernetes-specific. For other systems that use Prometheus, the node exporter is super important. On Kubernetes, if you manage your own nodes, this information can be invaluable too.

Here is a small subset of the metrics exposed by the node exporter:

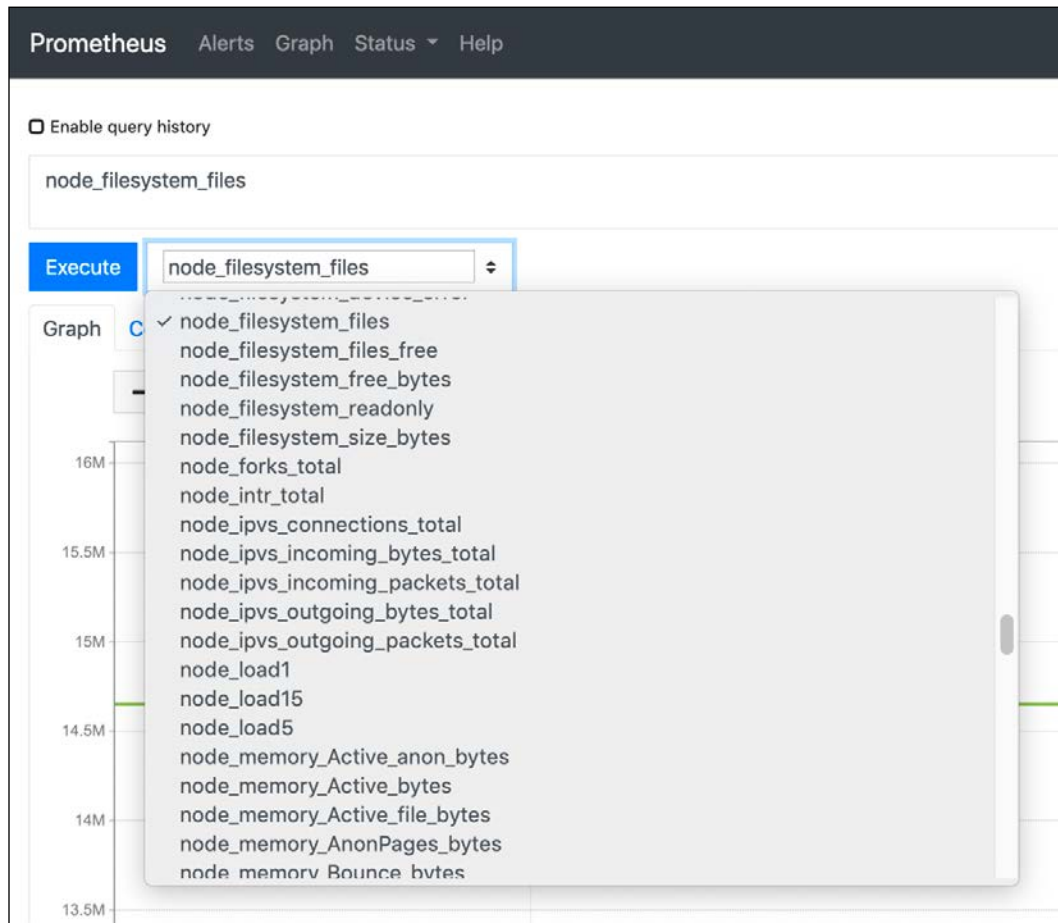


Figure 13.9: Metrics exposed by the node exporter

Incorporating custom metrics

The built-in metrics, node metrics, and Kubernetes metrics are great, but very often, the most interesting metrics are domain-specific and need to be captured as custom-metrics. There are two ways to do this:

- Write your own exporter and tell Prometheus to scrape it
- Use the push gateway, which allows you to push metrics into Prometheus
- In my book, *Hands-On Microservices with Kubernetes*, I provide a full-fledged example of how to implement your own exporter from a Go service

The push gateway is more appropriate if you already have a push-based metrics collector in place and you just want to have Prometheus record those metrics. It provides a convenient migration path from other metrics collection systems to Prometheus.

Alerting with Alertmanager

Collecting metrics is great, but when things go south, or ideally BEFORE things go south, you want to get notified. In Prometheus, this is the job of the Alertmanager. You can define rules as expressions-based metrics and when those expressions become true, they trigger an alert.

Alerts can serve multiple purposes. They can be handled automatically by a controller that is responsible for mitigating specific problems, they can wake up a poor on-call engineer at 3 A.M, they can result in an email or a group chat message, or any combination of those options.

The Alertmanager lets you group similar alerts into a single notification, inhibiting notifications if other alerts are already firing and silencing alerts. All those features are useful when a large-scale system is in trouble. The stakeholders are aware of the situation, and don't need repeated alerts or multiple variations of the same alert firing constantly while troubleshooting and trying to find the root cause.

One of the cool things about the Prometheus operator is that it manages everything in CRDs. That includes all the rules, including the alert rules:

```
$ kubectl get prometheusrules -n monitoring
```

NAME	AGE
monitoring-prometheus-oper-alertmanager.rules	2d9h
monitoring-prometheus-oper-etcd	2d9h
monitoring-prometheus-oper-general.rules	2d9h
monitoring-prometheus-oper-k8s.rules	2d9h
monitoring-prometheus-oper-kube-apiserver-error	2d9h

monitoring-prometheus-oper-kube-apiserver.rules	2d9h
monitoring-prometheus-oper-kube-prometheus-node-recording.rules	2d9h
monitoring-prometheus-oper-kube-scheduler.rules	2d9h
monitoring-prometheus-oper-kubernetes-absent	2d9h
monitoring-prometheus-oper-kubernetes-apps	2d9h
monitoring-prometheus-oper-kubernetes-resources	2d9h
monitoring-prometheus-oper-kubernetes-storage	2d9h
monitoring-prometheus-oper-kubernetes-system	2d9h
monitoring-prometheus-oper-kubernetes-system-apiserver	2d9h
monitoring-prometheus-oper-kubernetes-system-controller-manager	2d9h
monitoring-prometheus-oper-kubernetes-system-kubelet	2d9h
monitoring-prometheus-oper-kubernetes-system-scheduler	2d9h
monitoring-prometheus-oper-node-exporter	2d9h
monitoring-prometheus-oper-node-exporter.rules	2d9h
monitoring-prometheus-oper-node-network	2d9h
monitoring-prometheus-oper-node-time	2d9h
monitoring-prometheus-oper-node.rules	2d9h
monitoring-prometheus-oper-prometheus	2d9h
monitoring-prometheus-oper-prometheus-operator	2d9h

Here is the node time rule, which checks every second if the node time has deviated more than 0.05 of a second from the time of the node running the Prometheus pod (of course, you want to make sure this node's clock is correct by having NTP properly configured):

```
$ kubectl get prometheusrules monitoring-prometheus-oper-node-time -n
monitoring -o yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  ...
spec:
  groups:
  - name: node-time
    rules:
    - alert: ClockSkewDetected
      annotations:
        message: Clock skew detected on node-exporter {{ $labels.namespace
}}/{{ $labels.pod
}}. Ensure NTP is configured correctly on this host.
      expr: abs(node_timex_offset_seconds{job="node-exporter"}) > 0.05
      for: 2m
      labels:
        severity: warning
```

Alerts are very important, but there are cases where you want to visualize the overall state of your system or drill down into specific aspects. This is where visualization comes into play.

Visualizing your metrics with Grafana

You've already seen the Prometheus Expression browser, which can display your metrics as a graph or in table form. However, we can do much better. **Grafana** (<https://grafana.com/>) is an open source monitoring system that specializes in stunningly beautiful visualizations of metrics. It doesn't store the metrics itself, but works with many data sources, and Prometheus is one of them. Grafana has alerting capabilities too, but when working with Prometheus, it's best to rely on its Alertmanager.

The Prometheus operator installs Grafana and configures a large number of useful Kubernetes dashboards. Check out this beautiful dashboard of Kubernetes networking of pods filtered by namespace:



Figure 13.10: Pods filtered by namespace

To access Grafana, type the following commands:

```
$ POD_NAME=$(kubectl get pods -n monitoring -l "app=grafana" \
-o jsonpath="{.items[0].metadata.name}")
$ kubectl port-forward -n monitoring $POD_NAME 3000
```

Then, you can browse to <http://localhost:3000> and have some fun with Grafana. Grafana requires a username and password. The default credentials are admin for the user and prom-operator for the password.

Here are the dashboards that are pre-configured:

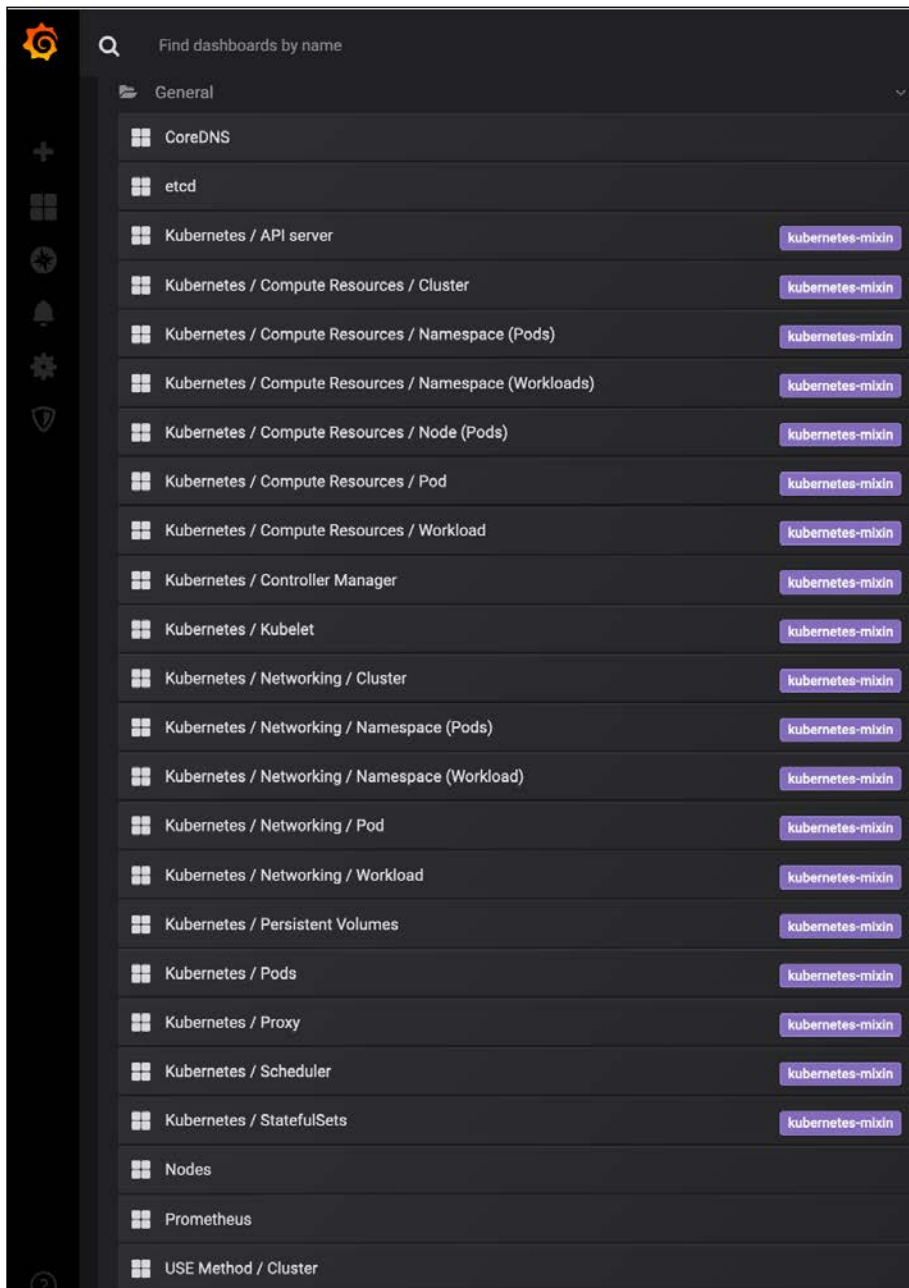


Figure 13.11: pre-configured dashboards

As you can see, the list is pretty extensive, but you can define your own dashboards if you want. There are a lot of fancy visualizations you can create with Grafana. I encourage you to explore it further. The Grafana dashboard is stored as config maps. If you want to add a custom dashboard, just add a config map that contains your dashboard spec. There is a dedicated sidecar container that watches new config maps being added and it will make sure to add your custom dashboard.

Considering Loki

If you like Prometheus and Grafana and you haven't settled on a centralized logging solution yet (or if you're unhappy with your current logging solution), then you should consider **Grafana Loki** (<https://grafana.com/oss/loki/>). Loki is an open source project for log aggregation, inspired by Prometheus. Unlike most log aggregation systems, it doesn't index the log contents but rather a set of labels applied to the log. That makes it very efficient. It is still relatively new (started in 2018), so you should evaluate whether it fits your needs before making the decision to adopt it. One thing is for sure: Loki has excellent Grafana support.

There are several advantages for Loki compared to something like EFK when Prometheus is used as the metrics platform. In particular, the set of labels you use to tag your metrics will serve just as well to tag your logs. Also, the fact that Grafana is used as a uniform visualization platform for both logs and metrics is useful.

We dedicated a lot of time to discussing metrics on Kubernetes. Let's talk about distributed tracing and the Jaeger project.

Distributed tracing with Jaeger

In microservice-based systems, every request may travel between multiple microservices calling each other, wait in queues, and trigger serverless functions. To debug and troubleshoot such systems, you need to be able to keep track of requests and follow them along their path.

Distributed tracing provides several capabilities that allow you, the developers, and the operators to understand their distributed systems:

- Distributed transaction monitoring
- Performance and latency tracking
- Root cause analysis
- Service dependency analysis
- Distributed context propagation

Distributed tracing often requires participation of the applications and services instrumenting endpoints. Since the microservices world is polyglot, multiple programming languages may be used. It makes sense to use a shared distributed tracing specification and framework that supports many programming languages. Enter OpenTracing...

What is OpenTracing?

OpenTracing (<https://opentracing.io/>) is an API specification and a set of frameworks and libraries in different languages. It is also an incubating CNCF project. OpenTracing is supported by multiple products and became a de facto standard. By using a product that complies with OpenTracing, you are not locked in and you work with an API that may be familiar to your developers. Note that OpenTracing recently merged with OpenCensus to form OpenTelemetry, which is a specification and platform for collecting both metrics and distributed traces. It is still in early development (Sandbox CNCF project), so we'll stick with OpenTracing at the moment.

Here is a list of the tracers that support OpenTracing:

- Jaeger
- LightStep
- Instana
- Apache SkyWalking
- inspectIT
- stagemonitor
- Datadog
- Wavefront by VMware
- Elastic APM

Most of the mainstream programming languages are supported:

- Go
- JavaScript
- Java
- Python
- Ruby
- PHP

- Objective-C
- C++
- C#

OpenTracing concepts

The two main concepts of OpenTracing are **Span** and **Trace**.

A **Span** is the basic unit of work or operation. It has a name, start time, and a duration. Spans can be nested if one operation starts another operation. Spans propagate with a unique ID and context. A **Trace** is an acyclic graph of Spans that originated from the same request and share the same context. A **Trace** represents the execution path of a request throughout the system. The following diagram illustrates the relationship between a Trace and Spans:

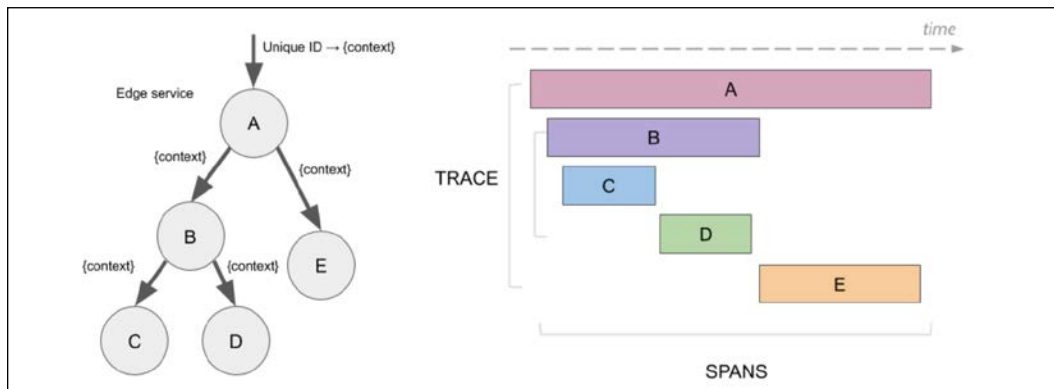


Figure 13.12: Trace and Spans relationship

Let's take a look at Jaeger.

Introducing Jaeger

Jaeger (<https://www.jaegertracing.io/>) is yet another CNCF graduated project, just like Fluentd and Prometheus. It completes the trinity of CNCF-graduated observability projects for Kubernetes. Jaeger was developed originally by Uber and quickly became the forerunner distributed tracing solution for Kubernetes.

There are other open source distributed tracing systems available, like **Zipkin** (<https://zipkin.io/>) and **AppDash** (<https://github.com/sourcegraph/appdash>). The inspiration for most of these systems (as well as Jaeger) is Google's **Dapper** (<https://research.google/pubs/pub36356/>). The cloud platform provides their own tracers, like AWS X-Ray.

There are various differences between all these systems. Jaeger's strong points are:

- Scalable design
- Multiple OpenTracing-compatible clients
- Light memory footprint
- Agents collect metrics over UDP

Jaeger architecture

Jaeger is a scalable system. It can be deployed as a single binary with all its components and stores the data in memory, but also as a distributed system where spans and traces are stored in persistent storage.

Jaeger has several components that collaborate to provide a world-class distributed tracing experience. The following diagram illustrates the architecture:

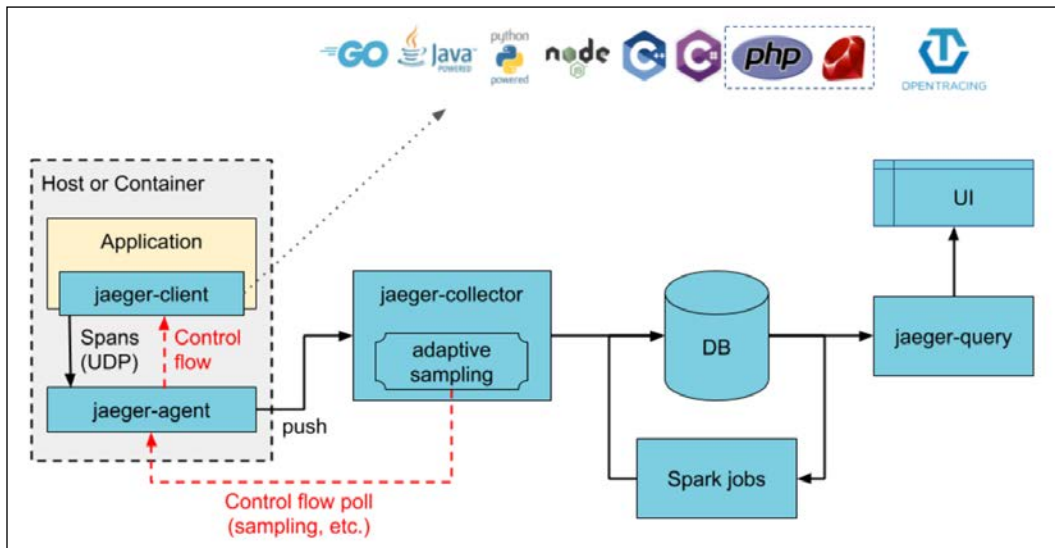


Figure 13.13: Jaeger architecture

Let's understand what the purpose of each component is.

Jaeger client

The Jaeger client is a library that implements the OpenTracing API in order to instrument a service or application for distributed tracing. The client library is used by the service or application to create spans and attach context like trace ID, span ID, and additional payload.

A very important aspect of Jaeger instrumentation is that it uses sampling and only 1 out of 1,000 traces are actually sampled. This is very different than logs and metrics, which record each and every event. This makes distributed tracing relatively lightweight, while still providing enough insight for high-volume applications.

Jaeger agent

The role of the agent is deployed locally to each node. It listens to spans over UDP – which makes it pretty performant – batches them, and sends them in bulk to the collector. This way, services don't need to discover collector or worry about connecting to them. Instrumented services simply send their spans to the local agent. The agent can also inform the client about sampling strategies.

Jaeger collector

The collector receives traces from all the agents. It is responsible for validating, indexing, transforming, and eventually storing the traces. The storage component can be a data store like Cassandra or Elasticsearch. However, it can also be a Kafka instance that enables async processing of traces.

Jaeger query

The Jaeger query service is responsible for presenting a UI to query the traces and spans the collector puts in storage.

Installing Jaeger

There are Helm charts that can be used to install Jaeger and the Jaeger operator, which is in beta at the time of writing. However, let's give it a try and see how far we can go:

```
$ helm repo add jaegertracing https://jaegertracing.github.io/helm-charts
```

```
$ helm search repo jaegertracing
```

NAME	CHART VERSION	APP VERSION	DESCRIPTION
jaegertracing/jaeger chart for Kubernetes	0.18.3	1.16.0	A Jaeger Helm
jaegertracing/jaeger-operator Helm chart for Kubernetes	2.12.3	1.16.0	jaeger-operator

Let's install Jaeger itself first, into the monitoring namespace:

```
$ helm install jaeger jaegertracing/jaeger -n monitoring
```

```
NAME: jaeger
```

```
LAST DEPLOYED: Fri Jun 12 20:03:24 2020
```

```
NAMESPACE: monitoring
```

```
STATUS: deployed
```

```
REVISION: 1
```

```
TEST SUITE: None
```

```
NOTES:
```

You can log into the Jaeger Query UI here:

```
export POD_NAME=$(kubectl get pods --namespace monitoring -l "app.kubernetes.io/instance=jaeger,app.kubernetes.io/component=query" -o jsonpath="{.items[0].metadata.name}")
echo http://127.0.0.1:8080/
kubectl port-forward --namespace monitoring $POD_NAME 8080:16686
```

Unfortunately, we can't use Helm 3 to install the Jaeger operator. We must resort to Helm 2 again:

```
$ helm2 install jaegertracing/jaeger-operator -n jaeger-operator
--namespace monitoring
```

```
NAME: jaeger-operator
```

```
LAST DEPLOYED: Sa Jun 13 00:42:17 2020
```

```
NAMESPACE: monitoring
```

```
STATUS: DEPLOYED
```

RESOURCES:

```
==> v1/Deployment
```

```
NAME AGE
```

```
jaeger-operator 0s
```

```
==> v1/Pod(related)
```

```
NAME AGE
```

```
jaeger-operator-b7f44c755-fwmrr 0s
```

```
==> v1/Role
```

```
NAME AGE
```

```
jaeger-operator 0s
```

```
==> v1/RoleBinding
```

```
NAME AGE
```

```
jaeger-operator 0s
```

```
==> v1/Service
```

```
NAME AGE
```

```
jaeger-operator-metrics 0s
```

```
==> v1/ServiceAccount
```

```
NAME AGE
```

```
jaeger-operator 0s
```

NOTES:

```
jaeger-operator is installed.
```

Check the jaeger-operator logs

```
export POD=$(kubectl get pods -l app.kubernetes.io/instance=jaeger-
```

```
operator -lapp.kubernetes.io/name=jaeger-operator --namespace monitoring
--output name)
kubectl logs $POD --namespace=monitoring
```

Let's bring up the Jaeger UI:

```
$ export POD_NAME=$(kubectl get pods --namespace monitoring -l "app.
kubernetes.io/instance=jaeger,app.kubernetes.io/component=query" -o
jsonpath="{.items[0].metadata.name}")
$ kubectl port-forward --namespace monitoring $POD_NAME 8080:16686
```

Now, we can browse to <http://localhost:8080> and see the Jaeger UI:

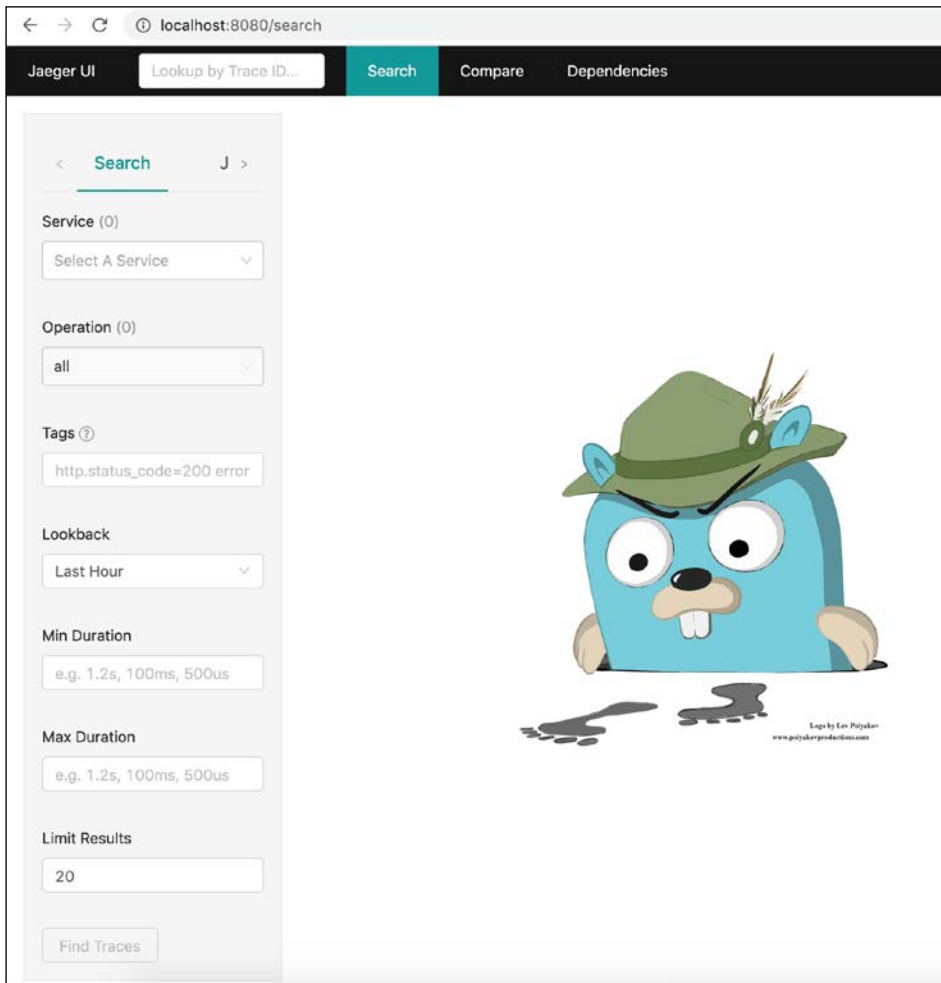


Figure 13.14: The Jaeger UI

In the next chapter – *Chapter 14, Utilizing Service Meshes* – we will see more of Jaeger and how to use it. Now, let's turn our attention to troubleshooting using all the monitoring and observability mechanisms we've discussed.

Troubleshooting problems

Troubleshooting a complex distributed system is no picnic. Abstractions, separation of concerns, information hiding, and encapsulation are great during development, testing, and when making changes to the system. But when things go wrong, you need to cross all those boundaries and layers of abstraction from the user action in their app through the entire stack, all the way to the infrastructure, thus crossing all the business logic, asynchronous processes, legacy systems, and third-party integrations. This is a challenge, even with large monolithic systems, but even more so with microservice-based distributed systems. Monitoring will assist you, but let's talk first about preparation, processes, and best practices.

Taking advantage of staging environments

When building a large system, developers work on their local machines (ignoring cloud development environments here) and eventually, the code is deployed to the production environment. However, there are a few steps between those two extremes. Complex systems operate in an environment that is not easy to duplicate locally. You should test changes that have been made to code or configuration in an environment that is similar to your production environment. This is your staging environment, where you should catch most problems that can't be caught by the developer running tests locally in their development environment.

The software delivery process should accommodate the detection of bad code and configuration as early as possible. However, sometimes, bad changes will be detected only in production and cause an incident. You should have an incident management process in place as well, which typically involves reverting to the previous version of whatever component caused the issue and then trying to find the root cause, often by debugging in the staging environment.

But sometimes, the problem is not with your code or configuration. In the end, your Kubernetes cluster runs on nodes (yes, even if it's managed), and those nodes can suffer many issues.

Detecting problems at the node level

In Kubernetes' conceptual model, the unit of work is the pod. However, pods are scheduled on nodes. When it comes to monitoring and the reliability of the infrastructure, the nodes are what require the most attention, because Kubernetes itself (the scheduler, replica sets, and horizontal pod autoscalers) takes care of the pods. Nodes can suffer from a variety of problems that Kubernetes is unaware of. As a result, it will keep scheduling pods to the bad nodes, and the pods might fail to function properly. Here are some of the problems that nodes may suffer while still appearing functional:

- Bad CPU
- Bad memory
- Bad disk
- Kernel deadlock
- Corrupt filesystem
- Problems with the container runtime (for example, the Docker daemon)

The kubelet running on each node can't detect these problems. We need another solution. Enter the node problem detector.

The node problem detector is a pod that runs on every node. It needs to solve a difficult problem. It must detect various low-level problems across different environments, different hardware, and different operating systems. It must be reliable enough not to be affected itself (otherwise, it can't report the problem), and it needs to have relatively low overhead to avoid spamming the master. In addition, it needs to run on every node. The source code can be found at <https://github.com/kubernetes/node-problem-detector>.

The most natural way is to deploy the node problem detector as a DaemonSet so that every node always has a problem detector. On Google's GCE clusters, it runs as an add-on.

Problem daemons

The problem with the node problem detector (pun intended) is that there are too many problems that it needs to handle. Trying to cram all of them into a single codebase can lead to a complex, bloated, and never-stabilizing codebase. The design of the node problem detector calls for separation of the core functionality of reporting node problems to the master from specific problem detection.

The reporting API is based on generic conditions and events. Problem detection should be done by separate problem daemons (each in its own container). This way, it is possible to add and evolve new problem detectors without impacting the core node problem detector. In addition, the control plane may have a remedy controller that can resolve some node problems automatically, therefore implementing self-healing.

At this stage (Kubernetes 1.18), problem daemons are baked into the node problem detector binary, and they execute as Goroutines, so you don't get the benefits of the loosely coupled design just yet. In the future, each problem daemon will run in its own container.

In addition to problems with nodes, the other area where things can break down is networking. The various monitoring tools we discussed earlier can help us identify problems across the infrastructure, in our code, or with third-party dependencies.

Let's talk about various options in our toolbox, how they compare, and how to utilize them for maximal effect.

Dashboards versus alerts

Dashboards are purely for humans. The idea of a good dashboard is to provide, at a glance, a lot of useful information about the state of the system or a particular component. There are many user experience elements to designing good dashboards, just like designing any UI. Monitoring dashboards can cover a lot of data across many components, over long time periods, and may support drilling down into finer and finer levels of detail.

Alerts, on the other hand, are constantly checking certain conditions (often based on metrics) and, when triggered, can either result in automatic resolution of the cause of the alert or eventually notify a human, who will probably start the investigation by looking at some dashboards.

Beyond self-healing systems that handle certain alerts automatically (or ideally resolve the issue before an alert is even raised), humans will typically be involved in troubleshooting. Even in cases where the system automatically recovered from a problem, at some point, a human will review the actions the system took and verify that the current behavior, including automatic recovery from problems, is adequate.

In many cases, severe problems (incidents) discovered by humans looking at dashboards (not scalable) or notified by alerts will require some investigation, remediation, and, later, post-mortem. In all those stages, the next layer of monitoring comes into play.

Logs versus metrics versus error reports

Let's understand where each of these tools excel and how best to combine their strengths to debug difficult problems. Let's assume we have good test coverage and our business/domain logic code is by and large correct. We run into problems in the production environment. There could be several types of problems that happen only in production:

- Misconfiguration (production configuration is incorrect out of date)
- Infrastructure provisioning
- Insufficient permissions and access to data, services, or third-party integrations
- Environment-specific code
- Software bugs that are exposed by production inputs
- Scalability issues

That's quite a list, and it's probably not even complete. Typically, when something goes wrong, it is in response to some change. What kind of changes are we talking about? Here are a few:

- Deployment of a new version of the code
- Dynamic reconfiguration of a deployed application
- New users or existing users changing the way they interact with the system
- Changes to the underlying infrastructure (for example, by cloud provider)
- New path in the code is utilized for the first time (for example, fallback to another region)

Since there is such a broad spectrum of problems and causes, it is difficult to suggest a linear path to resolution. For example, if the failure caused an error, then looking at an error report might be the best starting point. However, if the issue is that some action that was supposed to happen didn't happen, then there is no error to look at. In this case, it might make sense to look at the logs and compare them to the logs from a previous successful request. In the case of infrastructure or scalability problems, metrics may give us the best initial insight.

The bottom line is that debugging distributed systems requires using multiple tools together in the pursuit of the ever-elusive root cause.

Of course, in a distributed system with lots of components and microservices, it is not even clear where to look. This is where distributed tracing shines and can help us narrow down and identify the culprit.

Detecting performance and root cause with distributed tracing

With distributed tracing in place, every request will generate a trace with a graph of spans. Jaeger uses sampling of 1/1,000 by default, so once in a blue moon, issues might escape it, but for persistent problems, we will be able to follow the path of a request, see how long each span takes, and if the processing of a request bails out for some reason, it will be very easy to notice. At this point, you need to go back to the logs, metrics, and errors to hunt the root cause.

Summary

In this chapter, we covered the topics of monitoring, observability, and, in general, day 2 operations. We started with a review of the various aspects of monitoring: logs, metrics, error reporting, and distributed tracing. Then, we discussed how to incorporate monitoring capabilities into your Kubernetes cluster. We looked at several CNCF projects like Fluentd for log aggregation, Prometheus for metrics collection and alert management, Grafana for visualization, and Jaeger for distributed tracing. Then, we explored troubleshooting large distributed systems. We realized how difficult it can be and why we need so many different tools to conquer the issues.

In the next chapter, we will take things to the next level and dive into service meshes. I'm super excited about service meshes because they take much of the complexity related to cloud-native, microservice-based applications and externalize them outside of the microservices. That has a lot of real-world value.

14

Utilizing Service Meshes

In the previous chapter, we looked at monitoring and observability. One of the obstacles of a comprehensive monitoring story is that it requires a lot of changes to the code that are orthogonal to the business logic.

In this chapter, we will learn about service meshes, which allow you to externalize many of those cross-cutting concerns from the application code. The service mesh is a true paradigm shift in the way you can design, evolve, and operate distributed systems on Kubernetes. I like to think of it as aspect-oriented programming for cloud-native distributed systems. The topics we will cover are:

- What is a service mesh?
- Choosing a service mesh
- Incorporating Istio into your Kubernetes cluster

Let's jump right in.

What is a service mesh?

A service mesh is an architectural pattern for large-scale cloud native applications that are composed of many microservices. When your application is structured as a collection of microservices, there is a lot going on in the boundary between the microservices internally, inside your Kubernetes cluster.

This is different from traditional monolithic applications, where most of the processing is within the same process.

Here are some of the concerns that are relevant for each microservice or interaction between microservices:

- Advanced load balancing
- Service discovery
- Support canary deployments
- Caching
- Tracing a request across multiple microservices
- Authentication between services
- Throttling the number requests a service handles at a given time
- Automatically retrying failed requests
- Failing over to an alternative component when a component fails consistently
- Collecting metrics

All these concerns are completely orthogonal to the domain logic of the service, but they are all very important. A naive approach is to simply code all these concerns directly in each microservice. This obviously doesn't scale. So, a typical approach is to package all this functionality into a big library or set of libraries and use these libraries in each service:

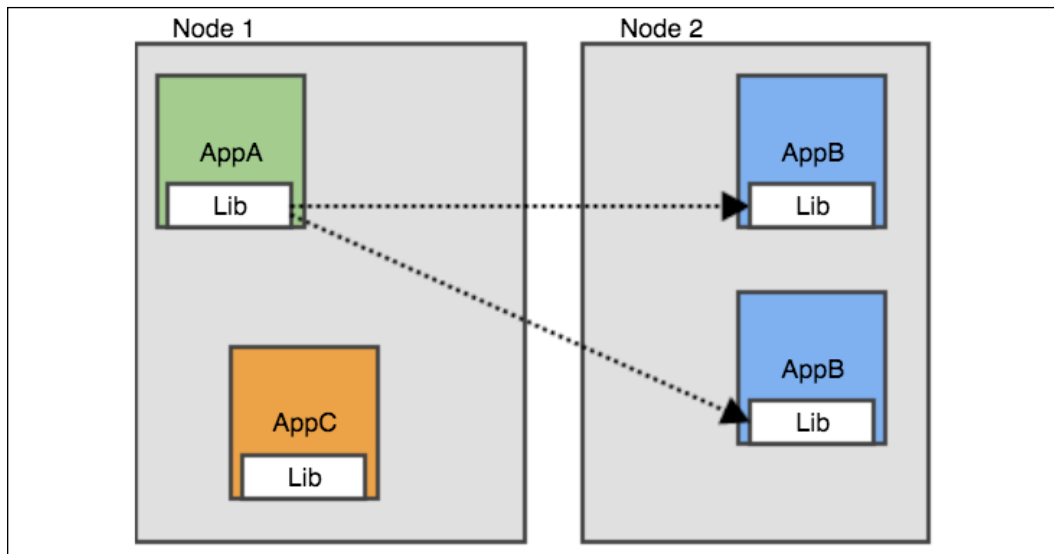


Figure 14.1: The big library approach

There are several problems with the big library approach:

- You need a library that supports all the programming languages you use
- If you want to update your library, you need to bump all your services
- It's difficult to upgrade all services at the same time

The service mesh doesn't touch your application. It injects a sidecar proxy container into each pod and a service mesh controller. The proxies intercept all communication between the pods and, in collaboration with the mesh controller, take care of all the cross-cutting concerns:

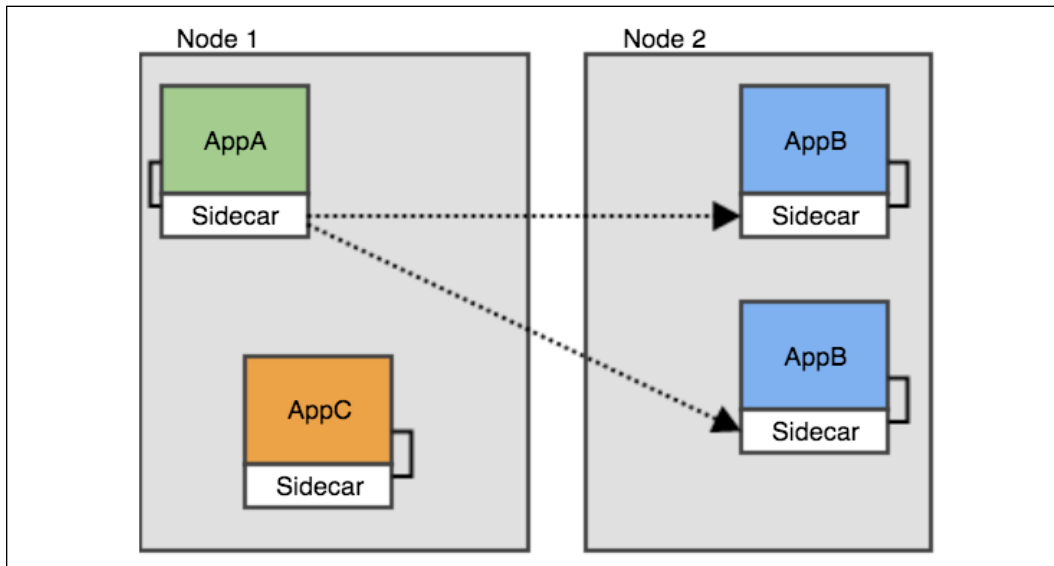


Figure 14.2: Using a service mesh controller

Here are some of the attributes of the proxy injection approach:

- The application is unaware of the service mesh
- You can turn the mesh on or off per pod and update the mesh independently
- No need to deploy an agent on each node
- Different pods on the same node can have different sidecars (or versions)
- Each pod has its own copy of the proxy

On Kubernetes, it looks like:

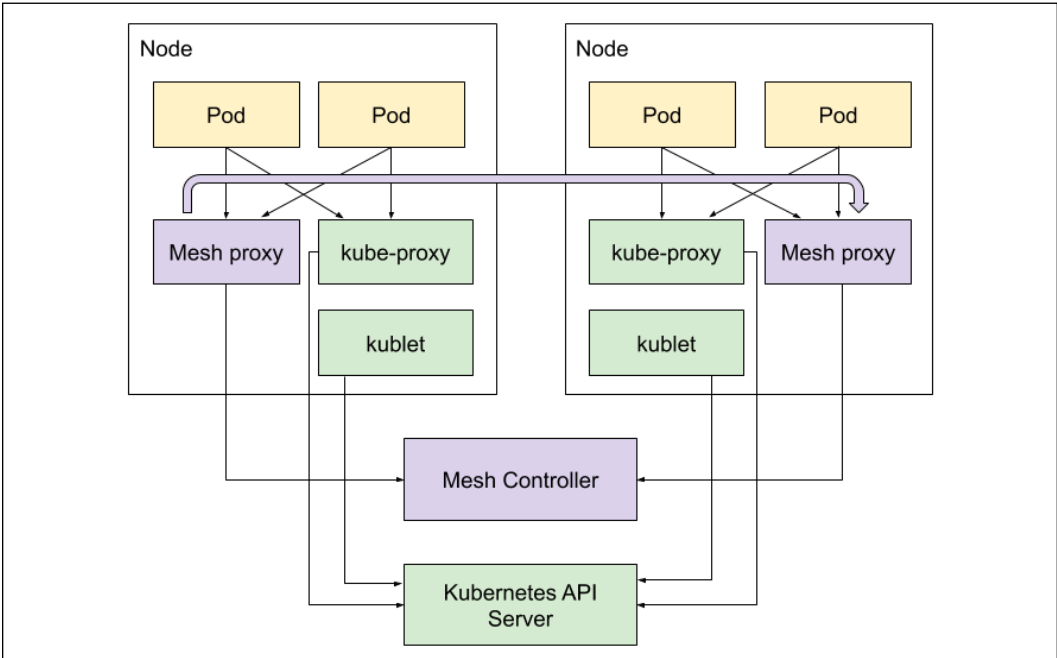


Figure 14.3: Kubernetes with a service mesh controller

There is another way to implement the service mesh proxy as a node agent, where it is not injected into each pod. This approach is less common, but in some cases (especially in non-Kubernetes environments), it is useful:

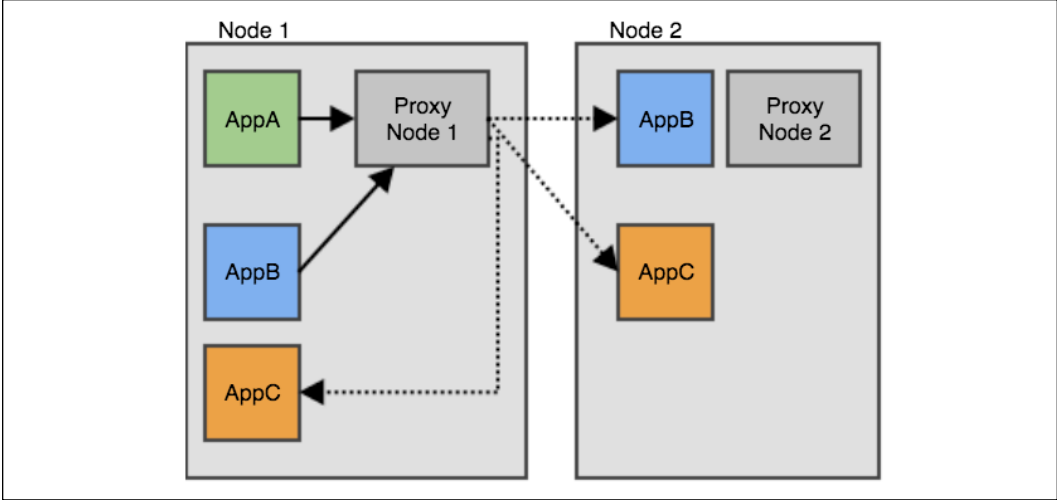


Figure 14.4: Service mesh proxy as a node agent

Control plane and data plane

In the service mesh world, there is a control plane, which is typically a set of controllers on Kubernetes, and there is a data plane, which contains the proxies that connect all the services in the mesh. The data plane consists of all the sidecar containers (or node agent) that intercept all communication between services in the mesh. The control plane is responsible for what actually happens when any traffic between services or a service and the outside world is intercepted.

Now that we have a good idea what a service mesh is, how it works, and why it is so useful, let's review some of the service meshes out there.

Choosing a service mesh

The service mesh concept is relatively new, but there are already many choices out there. Here is a concise review of the current cohort of service meshes.

Envoy

Envoy (<https://www.envoyproxy.io/>) is yet another CNCF graduated project. It is a very versatile and high-performance L7 proxy. It provides many service mesh capabilities, but it is considered pretty low-level and difficult to configure. It is also not Kubernetes-specific. Some of the Kubernetes service meshes use Envoy as the underlying data plane and provide a Kubernetes-native control plane to configure and interact with it. If you want to use Envoy directly on Kubernetes, then the recommendation is to use another open source projects like Ambassador or Gloo as an Ingress controller and/or API gateway.

Linkerd 2

Linkerd 2 (<https://linkerd.io/>) is a Kubernetes-specific service, as well as a CNCF incubating project. It is developed by **Buoyant** (<https://buoyant.io/>). Buoyant coined the term service mesh and introduced it to the world a few years ago. They started with a Scala-based service mesh for multiple platforms, including Kubernetes, called **Linkerd**. However, they decided to develop a better and more performant service mesh targeting Kubernetes only. That's where Linkerd 2 comes in. They implemented the data plane (proxy layer) in Rust and the control plane in Go.

Kuma

Kuma (<https://kuma.io/>) is a service mesh powered by Envoy. It works on Kubernetes, as well as other environments. It is developed by Kong. Its claim to fame is that it is super easy to configure.

AWS App Mesh

AWS, of course, has its own proprietary service mesh – **AWS App Mesh** (<https://aws.amazon.com/app-mesh/>). App Mesh also uses Envoy as its data plane. It can run on EC2, Fargate, ECS and EKS, and plain Kubernetes. App Mesh is a bit late to the service mesh scene, so it's not as mature as some of the other service meshes. However, it is based on the solid Envoy, so if you're an AWS shop, it may be the best choice due to its tight integration with AWS services.

Maesh

Maesh (<https://containo.us/maesh/>) was developed by the makers of **Traefik** (<https://containo.us/traefik/>). It is interesting because it uses the node agent approach as opposed to sidecar containers. It is based heavily on Traefik middleware in order to implement the service mesh functionality. You can configure it by using annotations on your services. It may be an interesting and lightweight approach to trying service meshes if you utilize Traefik at the edge of your cluster.

Istio

Last, but not least, **Istio** (<https://istio.io/>) is the most well-known service mesh on Kubernetes. It is built on top of Envoy and allows you to configure it in a Kubernetes-native way via YAML manifests. Istio was started by Google, IBM, and Lyft (the Envoy developers). It's a one-click install on Google GKE, but it is widely used in the Kubernetes community. It is also the default ingress/API gateway solution for Knative, which promotes its adoption even further.

Now that we've discussed the various service meshes choices, let's take Istio for a ride.

Incorporating Istio into your Kubernetes cluster

In this section, we will get to know Istio a little better, install it into a fresh cluster, and explore all the service goodness it provides.

Understanding the Istio architecture

First, let's meet the main components of Istio and understand what they do and how they relate to each other.

Istio is a large framework that provides a lot of capabilities, and it has multiple parts that interact with each other and with Kubernetes components (mostly indirectly and unobtrusively). It is divided into a control plane and a data plane. The data plane is a set of proxies (one per pod). Their control plane is a set of components that are responsible for configuring the proxies and collecting telemetry data.

The following diagram illustrates the different parts of Istio, how they are related to each other, and what information is exchanged between them:

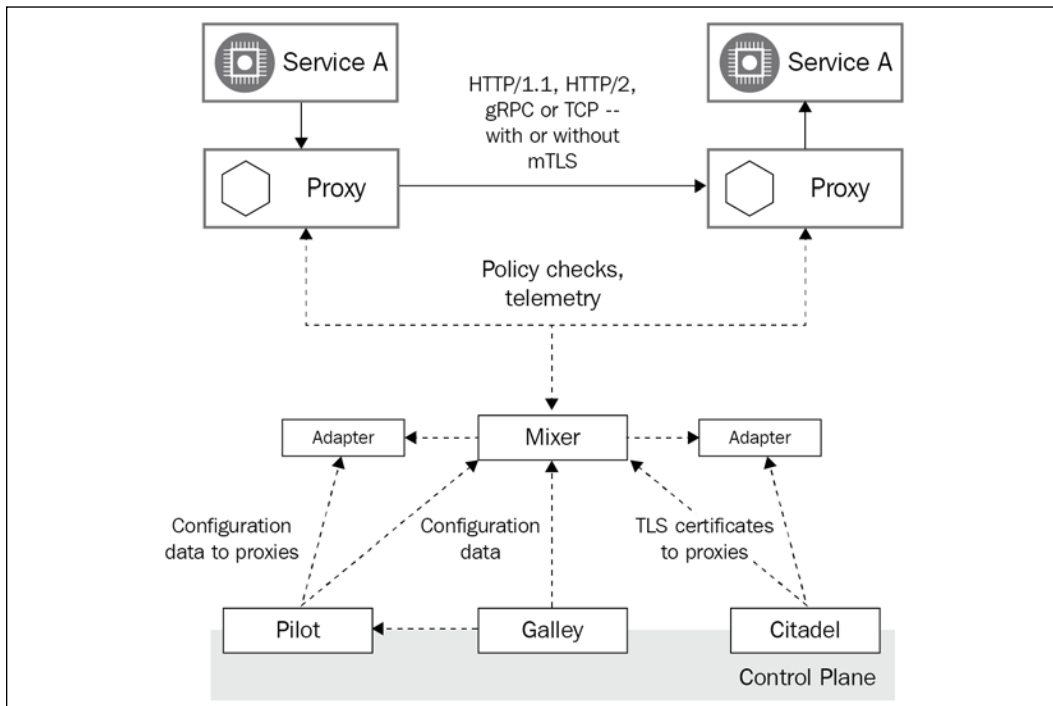


Figure 14.5: Istio architecture

Let's go a little deeper into each component, starting with the Envoy proxy.

Envoy

We discussed Envoy briefly when we reviewed service meshes for Kubernetes. Here, it serves as the data plane of Istio. Envoy is implemented in C++ and is a high-performance proxy. For each pod in the service mesh, Istio injects (either automatically or through the `istioctl` CLI) an Envoy side container that does all the heavy lifting, such as:

- Proxy HTTP, HTTP/2, and gRPC traffic between pods
- Sophisticated load balancing
- mTLS termination
- HTTP/2 and gRPC proxies
- Providing service health
- Circuit breaking for unhealthy services
- Percent-based traffic shaping
- Injecting faults for testing
- Detailed metrics

The Envoy proxy controls all the incoming and outgoing communication to its pod. It is, by far, the most important component of Istio. The configuration of Envoy is not trivial, and this is a large part of what the Istio control plane deals with.

The next component is Pilot.

Pilot

Pilot is responsible for platform-agnostic service discovery, dynamic load balancing, and routing. It translates high-level routing rules into an Envoy configuration. This abstraction layer allows Istio to run on multiple orchestration platforms. Pilot takes all the platform-specific information, converts it into the Envoy data plane configuration format, and propagates it to each Envoy proxy with the Envoy data plane API. Pilot is stateless; in Kubernetes, all the configuration is stored as **custom resources definitions (CRDs)** in etcd.

Let's move on to Mixer.

Mixer

Mixer is responsible for abstracting the metrics collection, policies, and auditing. These aspects are typically implemented in services by accessing APIs directly for specific backends. This has the benefit of offloading this burden from service developers and putting the control into the hands of the operators that configure Istio. It also enables switching backends easily without code changes. Here are some the backend types that Mixer can work with:

- Logging
- Authorization
- Quota
- Telemetry
- Billing

The interaction between the Envoy proxy and Mixer is straightforward – before each request, the proxy calls Mixer for precondition checks, which might cause the request to be rejected. After each request, the proxy reports the metrics to Mixer. Mixer has an adapter API to facilitate extensions for arbitrary infrastructure backends. It is a major part of its design.

The next component is Citadel.

Citadel

Citadel is responsible for certificate and key management. It is a key part of Istio security. Citadel integrates with various platforms and aligns with their identity mechanisms. For example, in Kubernetes, it uses service accounts; on AWS, it uses AWS IAM; and on GCP/GKE, it can use GCP IAM. The Istio PKI is based on Citadel. It uses X.509 certificates in SPIFEE format as a vehicle for service identity.

Here is the workflow in Kubernetes:

- Citadel creates certificates and key pairs for existing service accounts.
- Citadel watches the Kubernetes API server for new service accounts to provision with a certificate a key pair.
- Citadel stores the certificates and keys as Kubernetes secrets.
- Kubernetes mounts the secrets into each new pod that is associated with the service account (this is standard Kubernetes practice).
- Citadel automatically rotates the Kubernetes secrets when the certificates expire.

- Pilot generates secure naming information that associates a service account with an Istio service. Pilot then passes the secure naming information to the Envoy proxy.

The final major component that we will cover is Galley.

Galley

Galley is responsible for abstracting the user configuration on different platforms. It provides the ingested configuration to Pilot and Mixer. It is a pretty simple component.

Now that we have broken down Istio into its major components, let's get ready to install Istio into a Kubernetes cluster.

Preparing a minikube cluster for Istio

We will use a minikube cluster to check out Istio. Before installing Istio, we should make sure our cluster has enough capacity to handle Istio, as well as its demo application, **Bookinfo**. We will start minikube with 16 MB of memory and four CPUs, which should be adequate:

```
$ minikube start --memory=16384 --cpus=4
```

Minikube can provide a load balancer for Istio. Let's run this command in a separate Terminal as it will block:

```
$ minikube tunnel
```

Status:

```
machine: minikube
pid: 20463
route: 10.96.0.0/12 -> 192.168.64.5
minikube: Running
services: []
errors:
  minikube: no errors
  router: no errors
  loadbalancer emulator: no errors
```

Minikube sometimes doesn't clean up the tunnel network, so it's recommended to run the following command after you stop the cluster:

```
$ minikube tunnel --cleanup
```

Installing Istio

With minikube up and running, we can install Istio itself. There are multiple ways to install Istio:

- Customized installation with Istioctl (the Istio CLI)
- Customized installation with Helm
- Using the Istio operator (experimental)
- Multicluster installation

The Helm installation will not be supported in the future, so we will go with the recommended istioctl option:

```
$ curl -L https://istio.io/downloadIstio | sh -
```

The istioctl tool is located in /istio-1.6.3/bin (the version may be different when you download it). Make sure it's in your PATH. The Kubernetes installation manifests are in /istio-1.6.3/install/kubernetes and the examples are in /istio-1.6.3/samples.

We will install the built-in demo profile, which is great for evaluating Istio:

```
$ istioctl manifest apply --set profile=demo
- Applying manifest for component Base...
✓ Finished applying manifest for component Base.
- Applying manifest for component Citadel...
- Applying manifest for component Tracing...
- Applying manifest for component IngressGateway...
- Applying manifest for component Galley...
- Applying manifest for component Kiali...
- Applying manifest for component EgressGateway...
- Applying manifest for component Prometheus...
- Applying manifest for component Pilot...
- Applying manifest for component Policy...
- Applying manifest for component Injector...
- Applying manifest for component Telemetry...
- Applying manifest for component Grafana...
✓ Finished applying manifest for component Citadel.
✓ Finished applying manifest for component Kiali.
✓ Finished applying manifest for component Galley.
✓ Finished applying manifest for component Prometheus.
✓ Finished applying manifest for component Injector.
✓ Finished applying manifest for component Tracing.
```

- ✓ Finished applying manifest for component Policy.
- ✓ Finished applying manifest for component Grafana.
- ✓ Finished applying manifest for component IngressGateway.
- ✓ Finished applying manifest for component Pilot.
- ✓ Finished applying manifest for component EgressGateway.
- ✓ Finished applying manifest for component Telemetry.

✓ **Installation complete**

Some familiar names like Prometheus and Grafana pop up already. Let's review our cluster and see what is actually installed. Istio installs itself into the `istio-system` namespace, which is very convenient since it installs a lot of stuff. Let's check out what services Istio installed:

```
$ kubectl get svc -n istio-system -o name
service/grafana
service/istio-citadel
service/istio-egressgateway
service/istio-galley
service/istio-ingressgateway
service/istio-pilot
service/istio-policy
service/istio-sidecar-injector
service/istio-telemetry
service/jaeger-agent
service/jaeger-collector
service/jaeger-query
service/kiali
service/Prometheus
service/tracing
service/zipkin
```

There are quite a few services with an `istio-` prefix and then a bunch of other services:

- Prometheus
- Grafana
- Jaeger
- Zipkin
- Tracing
- Kiali

This means that if we use Istio in our Kubernetes cluster, we don't need to install Prometheus, Grafana, and Jaeger. They come with Istio. Also, the fact that Istio endorses them gives even more staying power.

OK. We've installed Istio successfully. Let's install the Bookinfo application, which is Istio's sample application, into our cluster.

Installing Bookinfo

Bookinfo is a simple microservice-based application that displays, as the name suggests, information on a single book such as name, description, ISBN, and even reviews. The Bookinfo developers really embraced the polyglot lifestyle, and each microservice is implemented in a different programming language:

- ProductPage service in Python
- Reviews service in Java
- Details service in Ruby
- Ratings service in JavaScript (Node.js)

The following diagram describes the relationships and flow of information between the Bookinfo services:

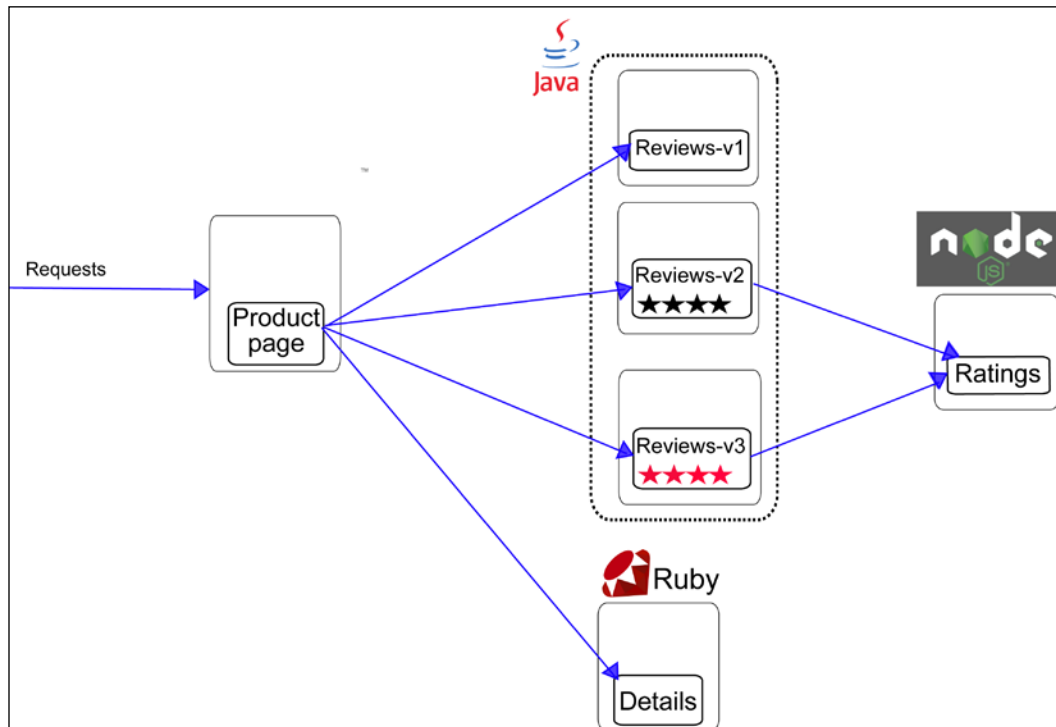


Figure 14.6: Bookinfo service relationships

Conveniently enough, Bookinfo comes with Istio, so we already have it in the `samples` sub-directory and we can install it from there:

```
$ cd samples/bookinfo
```

We're going to install it into its own `bookinfo` namespace. Let's create the namespace and then enable the Istio auto injection of the sidecar proxies by adding a label to the namespace:

```
$ kubectl create ns bookinfo
namespace/bookinfo created
$ kubectl label namespace bookinfo istio-injection=enabled
namespace/bookinfo labeled
```

Installing the application itself is a simple one-liner:

```
$ kubectl apply -f platform/kube/bookinfo.yaml -n bookinfo
service/details created
serviceaccount/bookinfo-details created
deployment.apps/details-v1 created
service/ratings created
serviceaccount/bookinfo-ratings created
deployment.apps/ratings-v1 created
service/reviews created
serviceaccount/bookinfo-reviews created
deployment.apps/reviews-v1 created
deployment.apps/reviews-v2 created
deployment.apps/reviews-v3 created
service/productpage created
serviceaccount/bookinfo-productpage created
deployment.apps/productpage-v1 created
```

Alright, the application was deployed successfully, including separate service accounts for each service. As you can see, three version of the reviews service were deployed. This will come in handy later, when we play with upgrades and advanced routing and deployment patterns.

Before moving on, we still need to wait for all the pods to initialize and then Istio will inject its sidecar proxy container. When the dust settles, you should see something like this:

```
$ kubectl get po -n bookinfo
```

NAME	READY	STATUS	RESTARTS	AGE
details-v1-78d78fbddf-sdssb	2/2	Running	0	101s
productpage-v1-596598f447-h9576	2/2	Running	0	100s
ratings-v1-6c9dbf6b45-bpqbl	2/2	Running	0	99s
reviews-v1-7bb8ffd9b6-7s6xh	2/2	Running	0	100s
reviews-v2-d7d75ff8-p5lh5	2/2	Running	0	100s
reviews-v3-68964bc4c8-4hqvr	2/2	Running	0	100s

Note that under the READY column, each pod shows 2/2, which means two containers per pod. One is the application container and the other is the injected proxy.

Since we're going to operate in the bookinfo namespace, let's define a little alias that will make our life simpler:

```
$ alias kb='kubectl -n bookinfo'
```

Now, armed with our little kb alias, we can verify that we can get the product page from the ratings service:

```
$ kb exec -it $(kb get pod -l app=ratings -o jsonpath='{.items[0].  
metadata.name}') -c ratings -- curl productpage:9080/productpage | grep -o  
"<title>.*</title>"
```

```
<title>Simple Bookstore App</title>
```

However, the application is not accessible to the outside world yet. This is where the Istio gateway comes in. Let's deploy it:

```
$ kb apply -f networking/bookinfo-gateway.yaml  
gateway.networking.istio.io/bookinfo-gateway created  
virtualservice.networking.istio.io/bookinfo created
```

On minikube, the external URL for the gateway can be constructed as:

```
export INGRESS_PORT=$(kubectl -n istio-system get service istio-  
ingressgateway -o jsonpath='{.spec.ports[?(@.name=="http2")].nodePort}')
```

```
$ export GATEWAY_URL=$(minikube ip):${INGRESS_PORT}
```

Now, we can try it from the outside:

```
$ http http://${GATEWAY_URL}/productpage | grep -o "<title>.*</title>"  
<title>Simple Bookstore App</title>
```

You can also open the URL in your browser and see some information about Shakespeare's *The Comedy of Errors*:

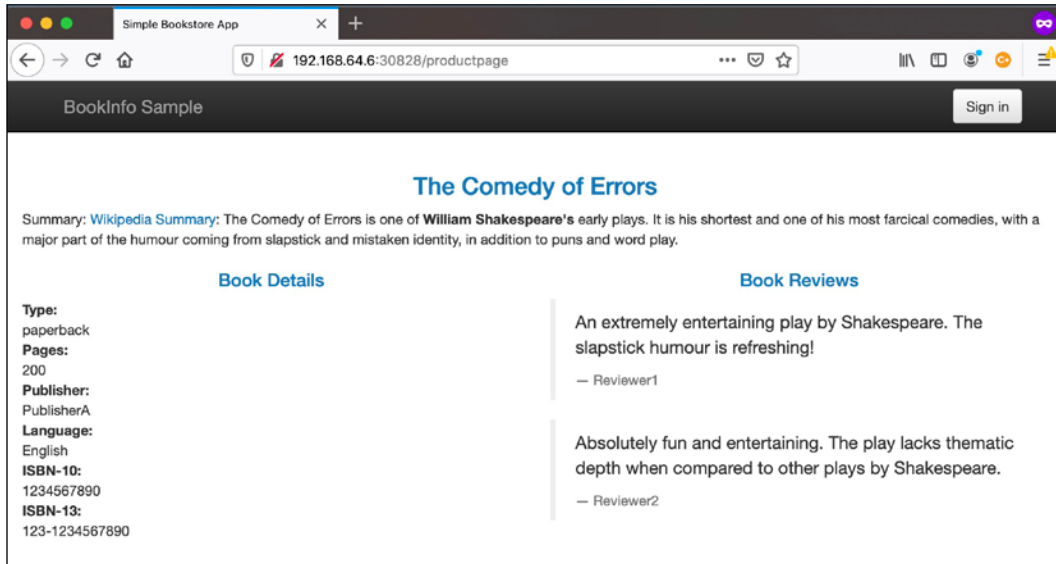


Figure 14.7: The Comedy of Errors

Alright. We're all set to start exploring the capabilities that Istio brings to the table. Let's start with traffic management.

Traffic management

Istio traffic management is about routing traffic to your services according to the destination rules you define. Istio keeps a service registry for all your services and their endpoints. The basic traffic management allows traffic between each pair of services and does simple round-robin load balancing between each service instance.

However, Istio can do much more. The traffic management API of Istio consists of five resources:

- Virtual services
- Destination rules
- Gateways
- Service entries
- Sidecars

Let's start by applying the default destination rules for Bookinfo:

```
$ kubectl apply -f networking/destination-rule-all.yaml
destinationrule.networking.istio.io/productpage created
destinationrule.networking.istio.io/reviews created
destinationrule.networking.istio.io/ratings created
destinationrule.networking.istio.io/details created
```

Then, let's create the Istio virtual services that represent the services in the mesh:

```
$ kubectl apply -f networking/virtual-service-all-v1.yaml
virtualservice.networking.istio.io/productpage created
virtualservice.networking.istio.io/reviews created
virtualservice.networking.istio.io/ratings created
virtualservice.networking.istio.io/details created
```

We need to wait a little for the virtual service configuration to propagate. Let's then check out the product page virtual service:

```
$ kubectl get virtualservices productpage -o yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  ...
  generation: 1
  name: productpage
  namespace: default
spec:
  hosts:
  - productpage
  http:
  - route:
    - destination:
        host: productpage
        subset: v1
```

It is pretty straightforward, specifying the HTTP route and the version. The v1 subset is important for the review service, which has multiple versions. The product page service will hit its v1 version because that is the subset that's configured.

Let's make it a little more interesting and do routing based on the logged-in user. Istio itself doesn't have a concept of user identity, but it routes traffic based on headers. The Bookinfo application adds an end user header to all requests.

The following command will update the routing rules:

```
$ kubectl apply -f networking/virtual-service-reviews-test-v2.yaml
virtualservice.networking.istio.io/reviews configured
```

Let's check the new rules:

```
$ kubectl get virtualservice reviews -o yaml
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  ...
spec:
  hosts:
  - reviews
  http:
  - match:
    - headers:
        end-user:
          exact: json
    route:
    - destination:
        host: reviews
        subset: v2
    - route:
        destination:
          host: reviews
          subset: v1
```

As you can see, if the HTTP header end user matches "json," then the request will be routed to subset 2 of the reviews service; otherwise, to subset 1. Version 2 of the reviews service adds a star rating to the reviews part of the page.

To test it, we can sign in as user "json" (any password will do) and get the following page:

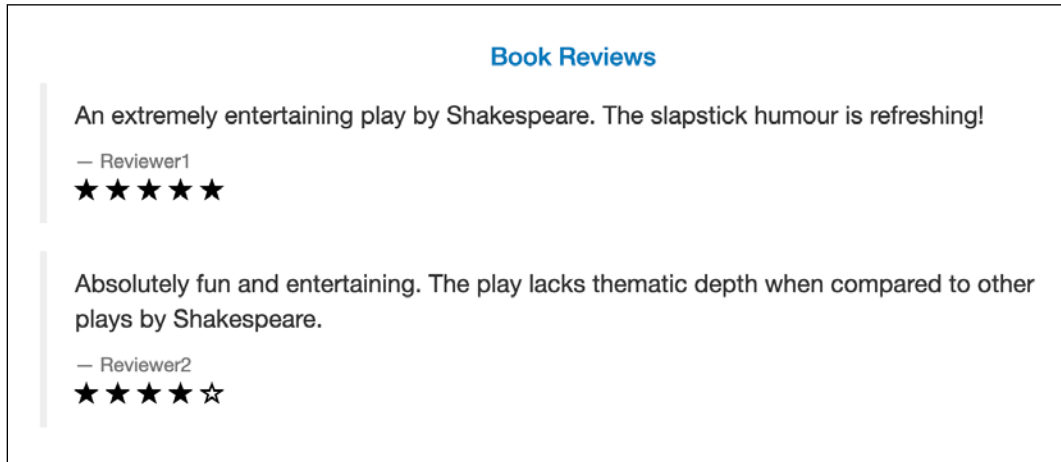


Figure 14.8: Reviews after signing in

There is much more Istio can do in the arena of traffic management:

- Fault injection for test purposes
- HTTP and TCP traffic shifting (gradually shift traffic from one version to the next)
- Request timeouts
- Circuit breaking
- Mirroring

In addition to internal traffic management, Istio supports configuring ingress into the cluster and egress from the cluster, including secure options with TLS and mutual TLS.

Security

Security is a core fixture of Istio. It provides identity management, authentication and authorization, security policies, and encryption. The security support is spread across many layers using multiple industry-standard protocols and best-practice security principles like defense in depth, security by default, and zero trust.

Here is the big picture of the Istio security architecture:

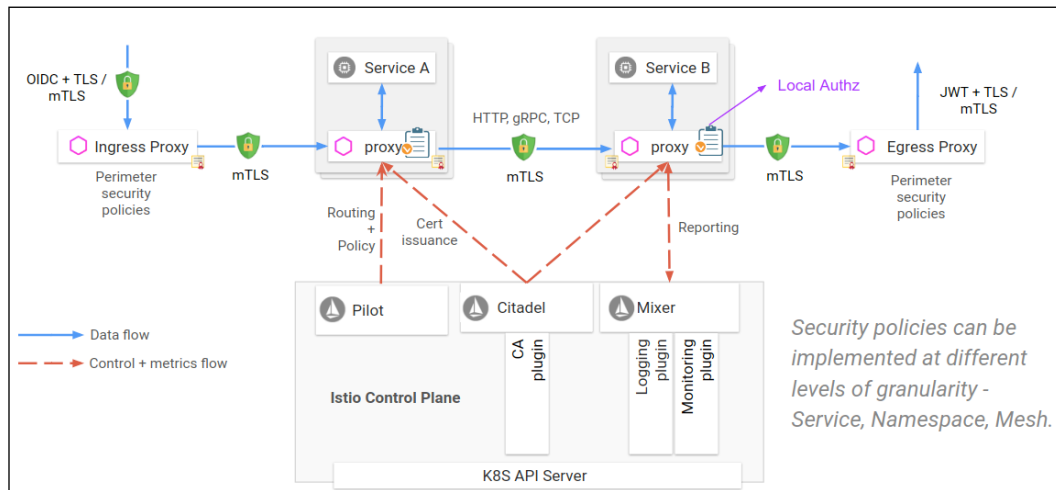


Figure 14.9: The Istio security architecture

All these components collaborate to enable a strong security posture:

- Citadel manages keys and certificates
- Sidecar and perimeter proxies implement authenticated and authorized communication between clients and servers
- Pilot distributes security policies and secure naming information to the proxies
- Mixer manages auditing

Let's break it down, piece by piece.

Istio identity

Istio utilizes secure naming where service names, as defined by the service discovery mechanism (for example, DNS), are mapped to server identities based on certificates. The clients verify the server identities. The server may be configured to verify the client identity. All the security policies apply to given identities. The servers decide what access a client has, based on their identity.

The Istio identity model can utilize existing identity infrastructure on the platform it is running on. On Kubernetes, it utilizes Kubernetes service accounts, of course.

Istio supports **SPIFEE** (<https://spiffe.io/>) — a standard for the secure identity framework. This is convenient because it allows Istio to integrate quickly with any SPIFEE compliant system. Specifically, Kubernetes X.509 certificates are SPIFEE-compliant.

Istio PKI

The Istio **public key infrastructure (PKI)** is based on Citadel to create SPIFEE certificates. The process on Kubernetes involves the following stages:

1. Citadel watches the Kubernetes API server. For each service account, it creates a SPIFFE certificate and a key pair, which it then stores as standard Kubernetes secrets.
2. Now, whenever Kubernetes creates a pod, it mounts the certificate and key pair as a secret volume called **istio-certs** that matches the service account.
3. Citadel watches the lifetime of each certificate and automatically rotates the certificates by rewriting the Kubernetes secrets.
4. Pilot generates the secure naming information, which defines what service account or accounts can run a certain service. Pilot then passes the secure naming information to the sidecar, Envoy.

Istio authentication

The secure identity model underlies the authentication framework of Istio. Istio supports two modes of authentication: transport authentication and origin authentication.

Transport authentication

Transport authentication is used for service to service authentication. The cool thing about it is that Istio provides it with no code changes. It ensures that service to service communication will take place only between services you configure with authentication policies.

Here is an authentication policy for the reviews service that requires mutual TLS:

```
apiVersion: "authentication.istio.io/v1alpha1"
kind: "Policy"
metadata:
  name: "reviews"
```

```
spec:
  targets:
    - name: reviews
  peers:
    - mtls: {}
```

Origin authentication

Origin authentication is used for end user authentication. Istio will verify that the end user making the request is allowed to make this request. This request-level authentication utilizes **JSON Web Tokens (JWTs)** and supports many OpenID Connect backends.

Here is an example of an origin authentication policy that excludes the `/health` endpoint for callers with a JWT token issued by Google:

origins:

```
- jwt:
  issuer: https://accounts.google.com
  jwksUri: https://www.googleapis.com/oauth2/v3/certs
  trigger_rules:
    - excluded_paths:
      - exact: /health
```

Once the identity of the caller has been established, the authentication framework passes it along with other claims to the next link in the chain – the authorization framework.

Istio authorization

Istio can authorize requests at many levels:

- Entire mesh
- Entire namespace
- Workload level

Here is the authorization architecture of Istio:

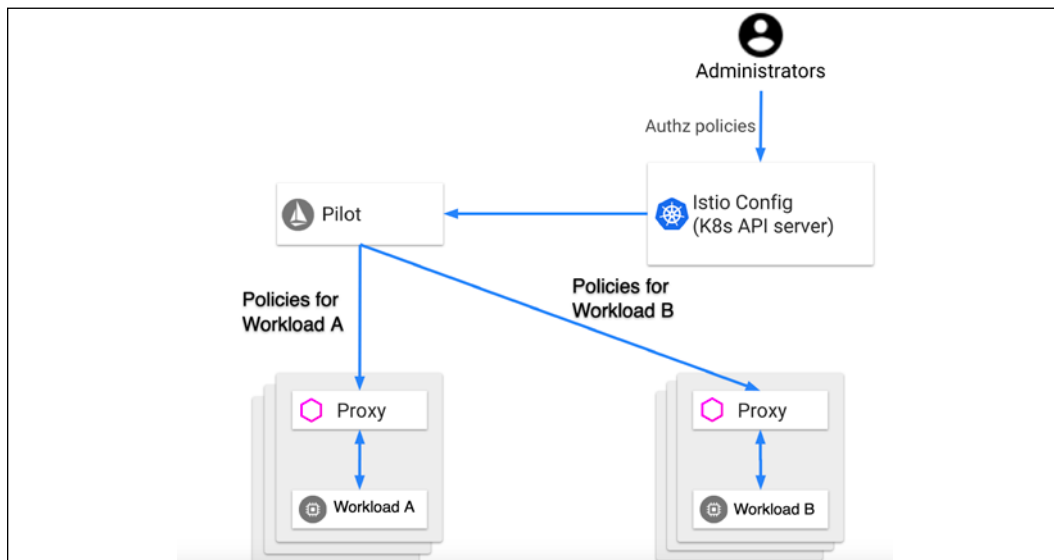


Figure 14.10: Istio's authorization architecture

Authorization is based on authorization policies. Each policy has a selector (what workloads it applies to) and rules (who is allowed to access a resource and under what conditions).

If no policy is defined on a workload, all requests are allowed. However, if a policy is defined for a workload, only requests that are allowed by a rule in the policy are allowed. There is no way to define exclusion rules.

Here is an `AuthorizationPolicy` that allows two sources (service account `cluster.local/ns/default/sa/sleep` and namespace `dev`) to access the workloads with the labels `app: httpbin` and `version: v1` in the namespace `foo` when the request is sent with a valid JWT token:

```

apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1

```

```

rules:
- from:
  - source:
    principals: ["cluster.local/ns/default/sa/sleep"]
  - source:
    namespaces: ["dev"]
  to:
  - operation:
    methods: ["GET"]
  when:
  - key: request.auth.claims[iss]
    values: ["https://accounts.google.com"]

```

The granularity doesn't have to be at the workload level. We can limit the access to specific endpoints and methods too. We can specify the operation using prefix match, suffix match, or presence match, in addition to exact match. For example, the following policy allows access to all paths that start with /test/ and all the paths that end in /info. The allowed methods are GET and HEAD only:

```

apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: tester
  namespace: default
spec:
  selector:
    matchLabels:
      app: products
  rules:
  - to:
    - operation:
      paths: ["/test/*", "*/info"]
      methods: ["GET", "HEAD"]

```

If we want to get even more fancy, we can specify conditions. For example, we can allow only requests with a specific header. Here is a policy that requires a version header with a value of v1 or v2:

```

apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin
  namespace: foo

```

```
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/sleep"]
      to:
    - operation:
        methods: ["GET"]
      when:
    - key: request.headers[version]
      values: ["v1", "v2"]
```

For TCP services, the paths and methods fields of the operation don't apply. Istio will simply ignore them. However, we can specify policies for specific ports:

```
apiVersion: "security.istio.io/v1beta1"
kind: AuthorizationPolicy
metadata:
  name: mongodb-policy
  namespace: default
spec:
  selector:
    matchLabels:
      app: mongodb
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/bookinfo-ratings-v2"]
      to:
    - operation:
        ports: ["27017"]
```

It is possible to plug in other authorization mechanisms by extending Mixer, but it is frowned upon. It's best to stick to Istio authorization, which is very powerful and flexible.

Let's move on to the topic of custom policies.

Policies

Istio is very flexible and lets us enable and define various custom policies to control how requests are handled. Here are some of the policies we can apply:

- Dynamically rate limit traffic to a service
- Whitelisting, blacklisting, and denying access to services
- Rewriting headers
- Redirecting requests
- Custom authorization policies

Policies are powered by the Mixer adapter model. Mixer abstracts away infrastructure backends such as telemetry systems, access control systems, quota enforcements systems, billing systems, and so on. The mixer adapter operates on a generic set of attributes that it receives from Envoy. It then feeds them to the adapter that knows how to interact with the infrastructure backends. The operators (you and me) can configure Mixer with policies that, based on the attribute, define behaviors for the adapter. Here is a diagram that illustrates the interactions between the different components:

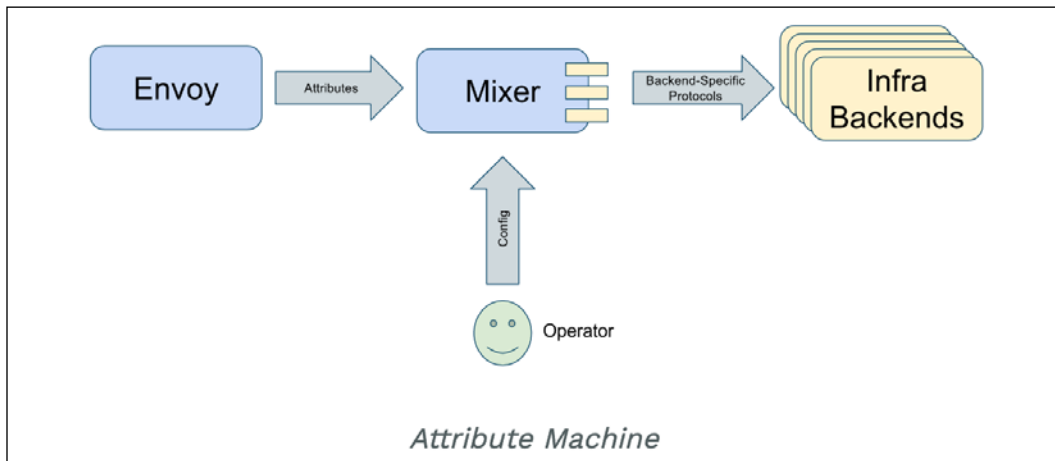


Figure 14.11: Interactions between Mixer components

Custom policies are not trivial. You have to understand the various objects involved and how to configure them correctly. Also, policy enforcement is disabled by default. If we don't enable it, Istio will ignore our policies.

It is configured in the istio ConfigMap. Here is how to verify that:

```
$ kubectl -n istio-system get cm istio -o jsonpath="{@.data.mesh}" | grep
disablePolicyChecks
disablePolicyChecks: true
```

Here is an easy way to enable it:

```
$ istioctl manifest apply --set values.global.disablePolicyChecks=false
```

If this doesn't work, you can directly edit the ConfigMap:

```
$ kubectl edit -n istio-system cm istio
```

Now, we can apply some policies. For example, here is how to apply the rate limit policy:

```
$ kubectl apply -f policy/mixer-rule-productpage-ratelimit.yaml
```

```
handler.config.istio.io/quotahandler created
instance.config.istio.io/requestcountquota created
quotaspec.config.istio.io/request-count created
quotaspecbinding.config.istio.io/request-count created
rule.config.istio.io/quota created
```

This created several objects: a handler, an instance, a quota spec, a quota spec binding, and finally a rule that ties all of them together.

Let's take a look at these objects and see how they collaborate to implement rate limiting. The instance is an instance of a quota template. It defines the dimensions, which in this case are the source, destination, and destination version:

```
$ kubectl -n istio-system get instance requestcountquota -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: instance
metadata:
  generation: 1
  name: requestcountquota
spec:
  compiledTemplate: quota
  params:
    dimensions:
      destination: destination.labels["app"] | destination.service.name |
"unknown"
```

```
destinationVersion: destination.labels["version"] | "unknown"
source: request.headers["x-forwarded-for"] | "unknown"
```

The handler here is a memquota adapter. For production systems, it is recommended to use a Redis adapter.

You can retrieve the spec using the following command:

```
$ kubectl -n istio-system get handler quotahandler -o yaml
```

The spec defines multiple quota schemes. First, a default quota of 500 requests per second:

```
spec:
  compiledAdapter: memquota
  params:
    quotas:
      - name: requestcountquota.instance.istio-system
        maxAmount: 500
        validDuration: 1s
```

Then, it defines overrides for specific services. For example, for the reviews service, only one request is allowed every 5 seconds:

```
overrides:
  - dimensions:
      destination: reviews
      maxAmount: 1
      validDuration: 5s
```

The productpage service allows two requests every 5 seconds:

```
- dimensions:
    destination: productpage
    maxAmount: 2
    validDuration: 5s
```

Unless the source has IP address 10.28.11.20, in which case, it's back to 500 requests per second:

```
- dimensions:
    destination: productpage
    source: 10.28.11.20
    maxAmount: 500
    validDuration: 1s
```


When a request is rejected due to a rate limit, Mixer will return a `RESOURCE_EXHAUSTED` message to Envoy, which will return an HTTP 429 status code to the caller.

The rule just ties together the quota instance with the handler:

```
$ kubectl -n istio-system get rule quota -o yaml
```

```
apiVersion: config.istio.io/v1alpha2
kind: rule
metadata:
  generation: 1
  name: quota
  namespace: istio-system
spec:
  actions:
  - handler: quotahandler
    instances:
    - requestcountquota
```

The policy can be applied restricted to a namespace. When the namespace is `istio-system`, it applies to the entire service mesh.

Let's look at one of the areas where Istio provides tremendous value – telemetry.

Monitoring and observability

Instrumenting your application for telemetry is a thankless job. You need to log, collect metrics, and create spans for tracing. This has several downsides:

- It takes time and effort to do it in the first place
- It takes more time and effort to ensure it is consistent across all the services in your cluster
- You can easily miss some important instrumentation point or configure it incorrectly
- If you want to change your log provider or distributed tracing solution, you might need to modify all your services
- It litters your code with lots of stuff that obscures the business logic
- You might need to explicitly turn it off for testing

What if all of this was taken care of automatically and never required any code changes? That's the promise of service mesh telemetry. Of course, you may need to do some work, especially if you want to capture custom metrics or do some specific logging. If your system is divided into coherent microservices along boundaries that really represent your domain and workflows, then Istio can help you get decent instrumentation, right out of the box. The idea is that Istio can keep track of what's going on in the seams between your services.

Logs

Istio can be configured for log collection, similar to the way we defined policies. The following command will create a log instance and a log handler:

```
$ kubectl apply -f telemetry/log-entry.yaml
instance.config.istio.io/newlog created
handler.config.istio.io/newloghandler created
```

It uses the logentry template and the stdio built-in adapter. On Kubernetes, the logs are collected as the container logs of the mixer. You can find them via the following command:

```
$ kubectl -n istio-system logs -l istio-mixer-type=telemetry | rg newlog
{"level":"warn","time":"2020-06-14T19:27:11.752616Z","instance":"newlog.
instance.istio-system","destination":"details","latency":"1.708946ms","resp
onseCode":200,"responseSize":178,"source":"productpage","user":"unknown"}
{"level":"warn","time":"2020-06-14T19:27:11.912198Z","instance":"newlog.
instance.istio-system","destination":"details","latency":"1.756211ms","resp
onseCode":200,"responseSize":178,"source":"productpage","user":"unknown"}
{"level":"warn","time":"2020-06-14T19:27:11.918363Z","instance":"newlog.
instance.istio-system","destination":"reviews","latency":"29.029062ms","res
ponseCode":200,"responseSize":375,"source":"productpage","user":"unknown"}
{"level":"warn","time":"2020-06-14T19:27:11.758456Z","instance":"newlog.
instance.istio-system","destination":"reviews","latency":"4.624288ms","resp
onseCode":200,"responseSize":295,"source":"productpage","user":"unknown"}
{"level":"warn","time":"2020-06-14T19:27:11.918611Z","instance":"newlog.
instance.istio-system","destination":"reviews","latency":"27.833674ms","res
ponseCode":200,"responseSize":375,"source":"productpage","user":"unknown"}
istio-mixer-type=telemetry
```

As you can see, each log entry contains the log level, the time stamp, the source, the destination, the response code, and the latency.

It is also possible to access the Envoy logs if necessary. Note that the Envoy logs are disabled by default. You can enable them in the Istio ConfigMap by setting `accessLogFile` to `/dev/output`.

Here are the Envoy logs of the `productpage` service:

```
$ kb logs -l app=productpage -c istio-proxy
[2020-06-14T19:27:11.565Z] "GET /reviews/0 HTTP/1.1" 200 - "-" "-"
0 375 19 19 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36"
"f482f9a7-2033-945f-8885-fd55038fb3ce" "reviews:9080" "172.17.0.18:9080"
outbound|9080||reviews.bookinfo.svc.cluster.local - 10.96.145.169:9080
172.17.0.24:54918 - default
[2020-06-14T19:27:11.547Z] "GET /productpage HTTP/1.1" 200 - "-" "-"
0 5282 40 40 "172.17.0.1" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
Safari/537.36" "f482f9a7-2033-945f-8885-fd55038fb3ce" "192.168.64.6:30828"
"127.0.0.1:9080" inbound|9080|http|productpage.bookinfo.svc.cluster.local -
172.17.0.24:9080 172.17.0.1:0 - default
[2020-06-14T19:27:11.752Z] "GET /details/0 HTTP/1.1" 200 - "-" "-" 0 178 1
1 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36" "d3c13544-
fc06-9e49-8aeb-077608d70316" "details:9080" "172.17.0.27:9080"
outbound|9080||details.bookinfo.svc.cluster.local - 10.96.117.135:9080
172.17.0.24:56584 - default
[2020-06-14T19:27:11.758Z] "GET /reviews/0 HTTP/1.1" 200 - "-" "-" 0 295 5
5 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36" "d3c13544-
fc06-9e49-8aeb-077608d70316" "reviews:9080" "172.17.0.21:9080"
outbound|9080||reviews.bookinfo.svc.cluster.local - 10.96.145.169:9080
172.17.0.24:54928 - default
[2020-06-14T19:27:11.747Z] "GET /productpage HTTP/1.1" 200 - "-" "-"
0 4286 18 18 "172.17.0.1" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
Safari/537.36" "d3c13544-fc06-9e49-8aeb-077608d70316" "192.168.64.6:30828"
"127.0.0.1:9080" inbound|9080|http|productpage.bookinfo.svc.cluster.local -
172.17.0.24:9080 172.17.0.1:0 - default
[2020-06-14T19:27:11.912Z] "GET /details/0 HTTP/1.1" 200 - "-" "-" 0 178 1
1 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36" "9f6f8d93-
fb3f-98b7-9911-1b1e74aabfbf" "details:9080" "172.17.0.27:9080"
outbound|9080||details.bookinfo.svc.cluster.local - 10.96.117.135:9080
172.17.0.24:56596 - default
[2020-06-14T19:27:11.918Z] "GET /reviews/0 HTTP/1.1" 200 - "-" "-"
0 375 29 28 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130 Safari/537.36"
```

```
"9f6f8d93-fb3f-98b7-9911-1b1e74aabfbf" "reviews:9080" "172.17.0.18:9080"
outbound|9080||reviews.bookinfo.svc.cluster.local - 10.96.145.169:9080
172.17.0.24:54940 - default
[2020-06-14T19:27:11.906Z] "GET /productpage HTTP/1.1" 200 - "-" "-"
0 5282 43 43 "172.17.0.1" "Mozilla/5.0 (Macintosh; Intel Mac OS X
10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.130
Safari/537.36" "9f6f8d93-fb3f-98b7-9911-1b1e74aabfbf" "192.168.64.6:30828"
"127.0.0.1:9080" inbound|9080|http|productpage.bookinfo.svc.cluster.local -
172.17.0.24:9080 172.17.0.1:0 - default
[Envoy (Epoch 0)] [2020-06-14 19:36:20.012][19][warning][config] [bazel-
out/k8-opt/bin/external/envoy/source/common/config/_virtual_includes/grpc_
stream_lib/common/config/grpc_stream.h:91] gRPC config stream closed: 13,
[Envoy (Epoch 0)] [2020-06-14 20:06:38.171][19][warning][config] [bazel-
out/k8-opt/bin/external/envoy/source/common/config/_virtual_includes/grpc_
stream_lib/common/config/grpc_stream.h:91] gRPC config stream closed: 13,
```

The format of the Envoy logs is text, but you can configure it to be JSON, by setting the `accessLogEncoding` to `JSON` in the `ConfigMap`. You can even set the format of the logs.

On Kubernetes, you can use `fluentd` to send all these logs to a centralized logging system.

Let's deploy a complete **Elasticsearch, Kibana, and Fluentd (EFK)** logging stack and see how it integrates with Istio. We will run the following command (the `logging-stack.yaml` file is in the code folder):

```
$ kubectl apply -f logging-stack.yaml
namespace/logging created
service/elasticsearch created
deployment.apps/elasticsearch created
service/fluentd-es created
deployment.apps/fluentd-es created
configmap/fluentd-es-config created
service/kibana created
deployment.apps/kibana created
```

The stack is deployed in its own logging namespace. We need to configure Istio to send its logs through `fluentd`. As usual, this is done by going through `Mixer` and defining `Instance`, `Handler`, and `Rule`:

```
$ kubectl apply -f telemetry/fluentd-istio.yaml
instance.config.istio.io/newlog created
handler.config.istio.io/handler created
rule.config.istio.io/newlogtofluentd created
```

It can take a few minutes for some pods to be ready. The next step is to make Kibana – the logging UI – accessible:

```
$ kubectl -n logging port-forward $(kubectl -n logging get pod -l app=kibana -o jsonpath='{.items[0].metadata.name}') 5601:5601
```

Now, we can browse to <http://localhost:5601> and start playing with Kibana. Here is what it looks like:



Figure 14.12: The Kibana UI

That pretty much covers logging with Istio. Let's see what Istio has to offer for metrics.

Metrics

Istio collects three types of metrics:

- Proxy metrics
- Control plane metrics
- Service metrics

The collected metrics cover all traffic into, from, and inside the service mesh. As operators, we need to configure Istio. Istio follows the four golden signals doctrine and records the latency, traffic, errors, and saturation.

Istio installs Prometheus and Grafana as its metrics collection and visualization backend.

To set up metrics collection, let's run the following command:

```
$ kubectl apply -f telemetry/metrics.yaml
instance.config.istio.io/doublerequestcount created
handler.config.istio.io/doublehandler created
rule.config.istio.io/doubleprom created
```

Here is an example of proxy-level metrics:

```
envoy_cluster_internal_upstream_rq{response_code_class="2xx",cluster_
name="xds-grpc"} 7163
envoy_cluster_upstream_rq_completed{cluster_name="xds-grpc"} 7164
envoy_cluster_ssl_connection_error{cluster_name="xds-grpc"} 0
envoy_cluster_lb_subsets_removed{cluster_name="xds-grpc"} 0
envoy_cluster_internal_upstream_rq{response_code="503",cluster_name="xds-
grpc"} 1
```

And here is an example of a service-level metric:

```
istio_requests_total{
  connection_security_policy="mutual_tls",
  destination_app="details",
  destination_principal="cluster.local/ns/default/sa/default",
  destination_service="details.default.svc.cluster.local",
  destination_service_name="details",
  destination_service_namespace="default",
  destination_version="v1",
  destination_workload="details-v1",
  destination_workload_namespace="default",
  reporter="destination",
```

```
    request_protocol="http",
    response_code="200",
    response_flags="- ",
    source_app="productpage",
    source_principal="cluster.local/ns/default/sa/default",
    source_version="v1",
    source_workload="productpage-v1",
    source_workload_namespace="default"
} 214
```

We can also collect metrics for TCP services. Let's install v2 of the ratings service, which uses MongoDB (a TCP service):

```
$ kb apply -f platform/kube/bookinfo-ratings-v2.yaml
serviceaccount/bookinfo-ratings-v2 created
deployment.apps/ratings-v2 created
```

Next, we install MongoDB itself:

```
$ kb apply -f platform/kube/bookinfo-db.yaml
service/mongodb created
deployment.apps/mongodb-v1 created
```

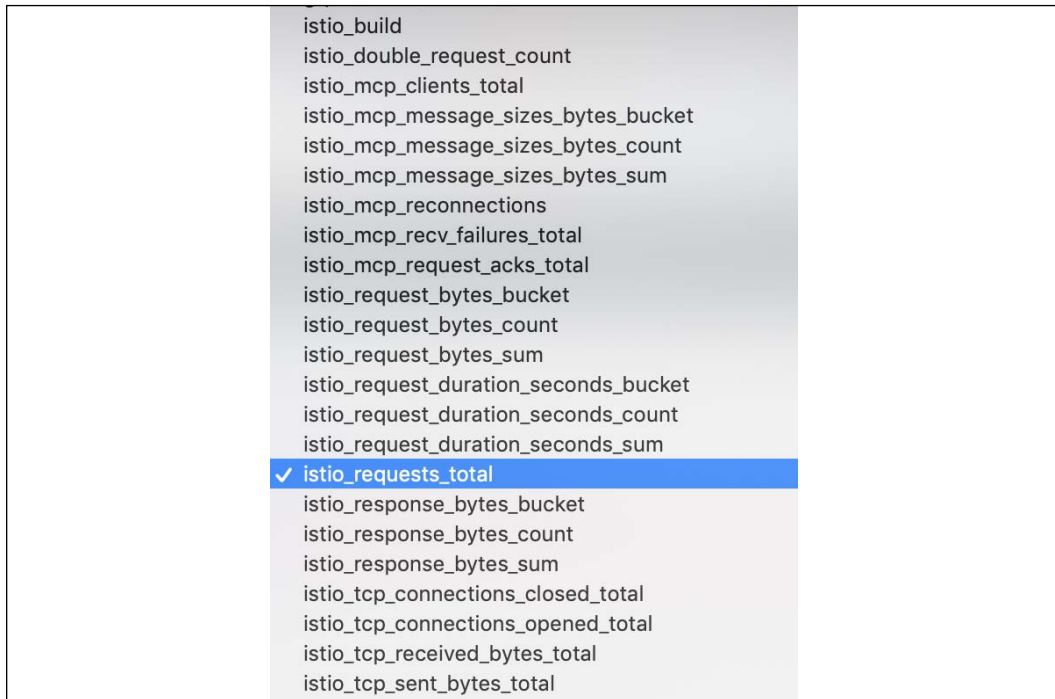
Finally, we need to create virtual services for the reviews and ratings service:

```
$ kb apply -f networking/virtual-service-ratings-db.yaml
virtualservice.networking.istio.io/reviews configured
virtualservice.networking.istio.io/ratings configured
```

At this point, we can expose Prometheus:

```
kubect1 -n istio-system port-forward \
$(kubect1 -n istio-system get pod -l app=prometheus \
-o jsonpath='{.items[0].metadata.name}') 9090:9090 &
```

We can view the slew of new metrics available from both Istio services, the Istio control plane, and especially Envoy. Here is a very small subset of the available metrics:



istio_build
istio_double_request_count
istio_mcp_clients_total
istio_mcp_message_sizes_bytes_bucket
istio_mcp_message_sizes_bytes_count
istio_mcp_message_sizes_bytes_sum
istio_mcp_reconnections
istio_mcp_recv_failures_total
istio_mcp_request_acks_total
istio_request_bytes_bucket
istio_request_bytes_count
istio_request_bytes_sum
istio_request_duration_seconds_bucket
istio_request_duration_seconds_count
istio_request_duration_seconds_sum
✓ istio_requests_total
istio_response_bytes_bucket
istio_response_bytes_count
istio_response_bytes_sum
istio_tcp_connections_closed_total
istio_tcp_connections_opened_total
istio_tcp_received_bytes_total
istio_tcp_sent_bytes_total

Figure 14.13: Some of the available metrics

The last pillar of observability is distributed tracing.

Distributed tracing

Istio configures the Envoy proxies to generate trace spans for the associated services. The services themselves are responsible for forwarding the request context. Istio can work with multiple tracing backends, such as:

- Gaeger
- Zipkin
- LightStep
- DataDog

Here are the request headers that services should propagate (only some may be present for each request, depending on the tracing backend):

- x-request-id**
- x-b3-traceid**
- x-b3-spanid**
- x-b3-parentspanid**
- x-b3-sampled**
- x-b3-flags**
- x-ot-span-context**
- x-cloud-trace-context**
- traceparent**
- grpc-trace-bin**

The sampling rate of traces is controlled by an environment variable of the Pilot: `PILOT_TRACE_SAMPLING`:

```
$ kubectl -n istio-system get deploy istio-pilot -o yaml \
| grep "name: PILOT_TRACE_SAMPLING" -A 1
```

```
  - name: PILOT_TRACE_SAMPLING
    value: "100"
```

The demo profile of Bookinfo samples 100% of the requests. We can change it to a lower rate with a granularity of 0.01. The default is 1%.

Now, we can start the Jaeger UI and explore the traces:

```
$ istioctl dashboard jaeger
http://localhost:52466
Handling connection for 9090
```

Your browser will automatically open and you should see the familiar Jaeger dashboard, where you can select a service and search for traces:

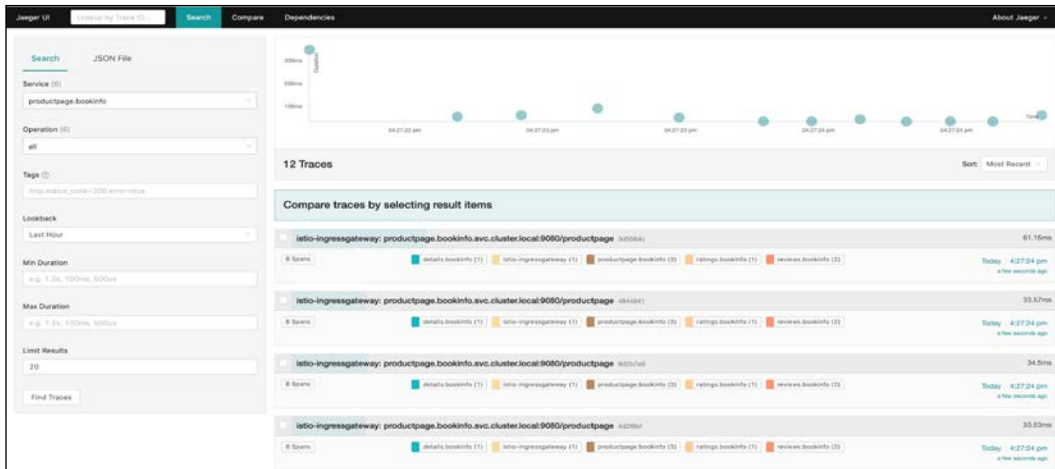


Figure 14.14: Selecting a service in Jaeger

You can click on a trace to see a detailed view of the flow of the request through the system:

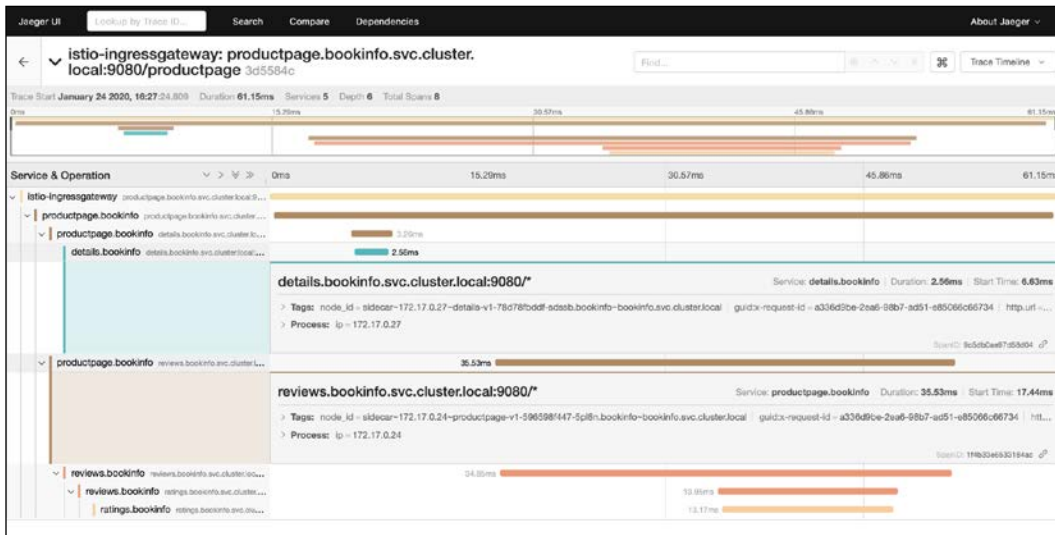


Figure 14.15: Detailed flow of the request

We've seen a lot of different tools with their own UI. Let's look at dedicated service mesh visualization.

Visualizing your service mesh with Kiali

Kiali is an open source project that ties together Prometheus, Grafana, and Jaeger to provide an observability console to your Istio service mesh. It can answer questions like:

- What microservices participate in the Istio service mesh?
- How are these microservices connected?
- How are these microservices performing?

It has various views and it really allows you to slice and dice your service mesh with zooming in and out, filtering and selecting various properties to display. It's got several views that you can switch between. Here is the overview page:

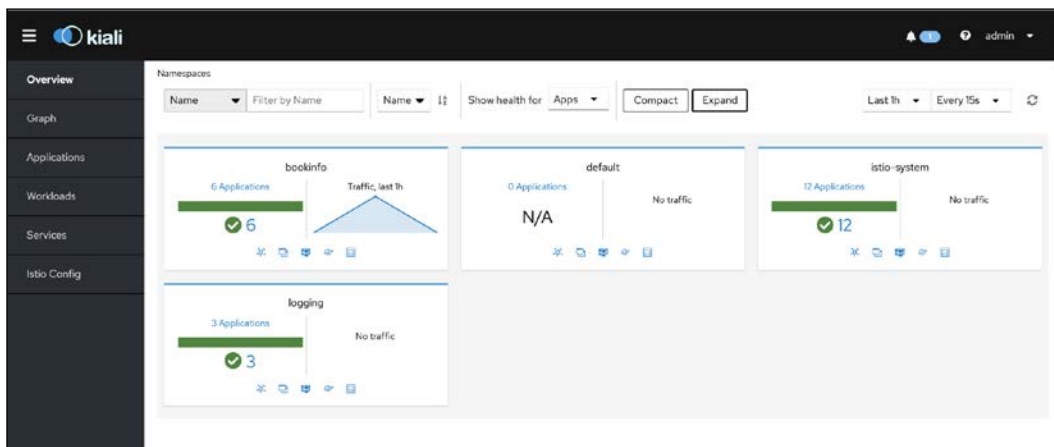


Figure 14.16: Kiali's overview page

However, the most interesting view is the graph view, which can show your services and how they relate to each other. It is fully aware of versions and requests flowing between different workloads, including percentage of requests and latency. It can show both HTTP and TCP services and really provides a great picture of how your service mesh behaves:

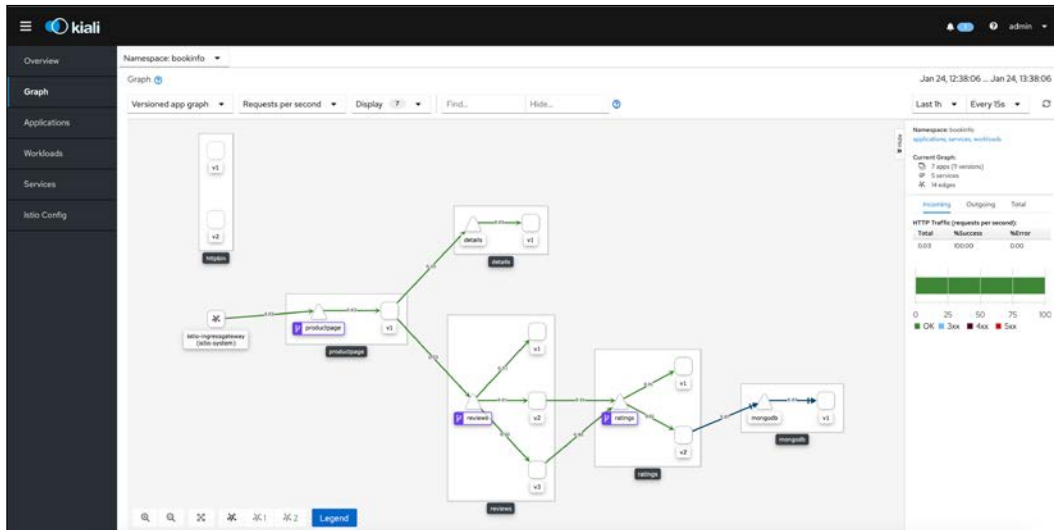


Figure 14.17: Graph view in Kiali

Summary

In this chapter, we did a very comprehensive study of service meshes on Kubernetes. Service meshes are here to stay. They are simply the right way to operate a complex distributed system. Separating all the operations concerns out to the proxies and having the service mesh to control them is a paradigm shift. Kubernetes, of course, is designed primarily for complex distributed systems, so the value of the service mesh becomes clear right away. It is also great to see that there are many options for service meshes on Kubernetes. While most service meshes are not specific to Kubernetes, it is one of the most important deployment platforms. In addition, we did a thorough review of Istio – arguably the service mesh with the most momentum – and took it through its paces. We demonstrated many of the benefits of service meshes and how they integrate with various other systems. You should be able to evaluate how useful a service mesh can be for your system and if you should deploy it and start enjoying the benefits.

In the next chapter, we'll look at the myriad of ways that we can extend Kubernetes and take advantage of its modular and flexible design. This is one of the hallmarks of Kubernetes and why it was adopted so quickly by so many communities.

15

Extending Kubernetes

In this chapter, we will dig deep into the guts of Kubernetes. We will start with the Kubernetes API and learn how to work with Kubernetes programmatically via direct access to the API, the Python client, and automating Kubectl. Then, we'll look into extending the Kubernetes API with custom resources. The last part is all about the various plugins Kubernetes supports. Many aspects of Kubernetes operation are modular and designed for extension. We will examine the API aggregation layer and several types of plugins, such as custom schedulers, authorization, admission control, custom metrics, and volumes. Finally, we'll look into extending Kubectl and adding your own commands.

The topics covered in this chapter are as follows:

- Working with the Kubernetes API
- Extending the Kubernetes API
- Writing Kubernetes and Kubectl plugins
- Writing webhooks

Working with the Kubernetes API

The Kubernetes API is comprehensive and encompasses the entire functionality of Kubernetes. As you may expect, it is huge. But it is designed very well using best practices, and it is consistent. If you understand the basic principles, you can discover everything you need to know.

Understanding OpenAPI

OpenAPI allows API providers to define their operations and models and enables developers to automate their tools and generate their favorite language's client to talk to that API server. Kubernetes has supported Swagger 1.2 (an older version of the OpenAPI spec) for a while, but the spec was incomplete and invalid, making it hard to generate tools/clients based on it.

In Kubernetes 1.4, alpha support was added for the OpenAPI spec (formerly known as Swagger 2.0 before it was donated to the OpenAPI Initiative) by upgrading the current models and operations. In Kubernetes 1.5, support for the OpenAPI spec has been completed by auto-generating the spec directly from the Kubernetes source, which will keep the spec and documentation completely in sync with future changes in operations/models.

The new spec enables better API documentation and an auto-generated Python client that we will explore later.

The spec is modular and divided by group version. This is future-proof. You can run multiple API servers that support different versions. Applications can transition gradually to newer versions.

The structure of the spec is explained in detail in the OpenAPI spec definition. The Kubernetes team used the operation's tags to separate each group version and fill in as much information as possible about paths/operations and models. For a specific operation, all parameters, methods of call, and responses are documented. The result is impressive.

Setting up a proxy

To simplify access, you can use Kubectl to set up a proxy:

```
$ kubectl proxy --port 8080
```

Now, you can access the API server on `http://localhost:8080` and it will reach the same Kubernetes API server that Kubectl is configured for.

Exploring the Kubernetes API directly

The Kubernetes API is highly discoverable. You can just browse to the URL of the API server at `http://localhost:8080` and get a nice JSON document that describes all the available operations under the paths key.

Here is a partial list due to space constraints:

```
{
  "paths": [
    "/api",
    "/api/v1",
    "/apis",
    "/apis/apps",
    "/apis/storage.k8s.io/v1",
    .
    .
    .
    "/healthz",
    "/healthz/ping",
    "/logs",
    "/metrics",
    "/swaggerapi/",
    "/ui/",
    "/version"
  ]
}
```

You can drill down any one of the paths. For example, here is the response from the `/api/v1/namespaces/default` endpoint:

```
{
  "apiVersion": "v1",
  "kind": "Namespace",
  "metadata": {
    "creationTimestamp": "2017-12-25T10:04:26Z",
    "name": "default",
    "resourceVersion": "4",
    "selfLink": "/api/v1/namespaces/default",
    "uid": "fd497868-e95a-11e7-adce-080027c94384"
  },
  "spec": {
    "finalizers": [
      "kubernetes"
    ]
  },
  "status": {
    "phase": "Active"
  }
}
```


I discovered this endpoint by going first to `/api`, then discovered `/api/v1`, which told me there is `/api/v1/namespaces`, which pointed me to `/api/v1/namespaces/default`.

Using Postman to explore the Kubernetes API

Postman (<https://www.getpostman.com>) is a very polished application for working with RESTful APIs. If you lean more to the GUI side, you may find it extremely useful.

The following screenshot shows the available endpoints in the batch V1 API group:

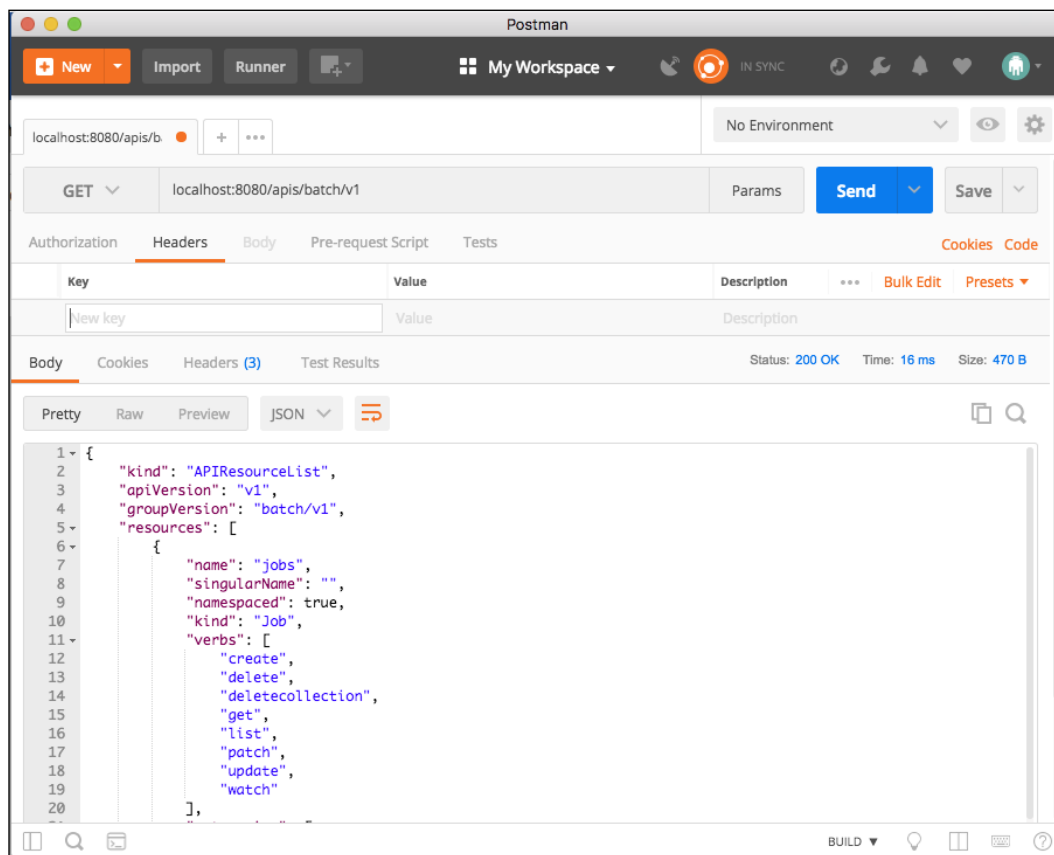


Figure 15.1: Available endpoints in the batch V1 API group

Postman has a lot of options and it organizes the information in a very pleasing way. Give it a try.

Filtering the output with HTTPie and jq

The output from the API can be too verbose sometimes. Often, you're interested just in one value out of a huge chunk of JSON response. For example, if you want to get the names of all running services you can hit the `/api/v1/services` endpoint. The response, however, includes a lot of additional information that is irrelevant. Here is a very partial subset of the output:

```
$ http http://localhost:8080/api/v1/services
{
  "apiVersion": "v1",
  "items": [
    {
      "metadata": {
        "creationTimestamp": "2020-06-15T05:18:30Z",
        "labels": {
          "component": "apiserver",
          "provider": "kubernetes"
        },
        "name": "kubernetes",
        ...
      },
      "spec": {
        ...
      },
      "status": {
        "loadBalancer": {}
      }
    },
    ...
  ],
  "kind": "ServiceList",
  "metadata": {
    "resourceVersion": "1076",
    "selfLink": "/api/v1/services"
  }
}
```

The complete output is 121 lines long! Let's see how to use **HTTPie** and **jq** to gain full control over the output and show only the names of the services. I prefer (<https://httpie.org/>) over `cURL` for interacting with REST APIs on the command line. The `jq` (<https://stedolan.github.io/jq/>) command-line JSON processor is great for slicing and dicing JSON.

Examining the full output, you can see that the service name is in the `metadata` section of each item in the `items` array. The `jq` expression that will select just the name is as follows:

```
.items[].metadata.name
```

Here is the full command and output:

```
$ http http://localhost:8080/api/v1/services | jq .items[].metadata.name
"kubernetes"
"kube-dns"
"kubernetes-dashboard"
```

Creating a pod via the Kubernetes API

The API can be used for creating, updating, and deleting resources too. Unlike working with Kubectl, the API requires specifying the manifests in JSON and not YAML syntax (although every JSON document is also valid YAML). Here is a JSON pod definition (`nginx-pod.json`):

```
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "nginx",
    "namespace": "default",
    "labels": {
      "name": "nginx"
    }
  },
  "spec": {
    "containers": [{
      "name": "nginx",
      "image": "nginx",
      "ports": [{"containerPort": 80}]
    }]
  }
}
```

The following command will create the pod via the API:

```
$ http POST http://localhost:8080/api/v1/namespaces/default/pods @nginx-pod.json
```

To verify it worked, let's extract the name and status of the current pods. The endpoint is `/api/v1/namespaces/default/pods`.

The jq expression is `items[].metadata.name,.items[].status.phase`.

Here is the full command and output:

```
$ FILTER='.items[].metadata.name,.items[].status.phase'
$ http http://localhost:8080/api/v1/namespaces/default/pods | jq $FILTER
"nginx"
"Running"
```

Accessing the Kubernetes API via the Python client

Exploring the API interactively using `httplib` and `jq` is great, but the real power of APIs comes when you consume and integrate them with other software. The Kubernetes incubator project provides a full-fledged and very well-documented Python client library. It is available at <https://github.com/kubernetes-incubator/client-python>.

First, make sure you have Python installed (both 2.7 and 3.5+ work). Then install the Kubernetes package:

```
$ pip install kubernetes
```

To start talking to a Kubernetes cluster, you need to connect to it. Start an interactive Python session:

```
$ python
Python 3.8.0 (default, Jun 15 2020, 16:12:10)
[Clang 10.0.1 (clang-1001.0.46.4)] on Darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The Python client can read your `Kubect`l config:

```
>>> from kubernetes import client, config
>>> config.load_kube_config()

>>> v1 = client.CoreV1Api()
```

Or it can connect directly to an already running proxy:

```
>>> from kubernetes import client, config
>>> client.Configuration().host = 'http://localhost:8080'
>>> v1 = client.CoreV1Api()
```

Note that the client module provides methods to get access to different group versions, such as CoreV1API.

Dissecting the CoreV1API group

Let's dive in and understand the CoreV1API group. The Python object has 397 public attributes!

```
>>> attributes = [x for x in dir(v1) if not x.startswith('__')]
>>> len(attributes)
397
```

We ignore the attributes that start with double underscores because they are special class/instance methods unrelated to Kubernetes.

Let's pick ten random methods and see what they look like:

```
>>> import random
>>> from pprint import pprint as pp
>>> pp(random.sample(attributes, 10))
['list_namespaced_secret',
 'connect_post_namespaced_pod_proxy',
 'patch_namespaced_replication_controller_with_http_info',
 'patch_node_status_with_http_info',
 'replace_persistent_volume',
 'read_namespaced_service_status',
 'read_namespaced_replication_controller_status',
 'list_namespaced_secret_with_http_info',
 'replace_namespaced_event_with_http_info',
 'replace_namespaced_resource_quota_with_http_info']
```

Very interesting. The attributes begin with a verb such as list, patch, or read. Many of them have a notion of a namespace and many have a with_http_info suffix. To understand better, let's count how many verbs exist and how many attributes use each verb (where the verb is the first token before the underscore):

```
>>> from collections import Counter
```

```
>>> verbs = [x.split('_')[0] for x in attributes]
>>> pp(dict(Counter(verbs)))
{'api': 1,
 'connect': 96,
 'create': 36,
 'delete': 56,
 'get': 2,
 'list': 56,
 'patch': 48,
 'read': 52,
 'replace': 50}
```

We can drill further and look at the interactive help for a specific attribute:

```
>>> help(v1.create_node)
```

Help on method `create_node` in module `kubernetes.client.apis.core_v1_api`:

`create_node(body, **kwargs)` method of `kubernetes.client.apis.core_v1_api.CoreV1Api` instance

create a Node

This method makes a synchronous HTTP request by default. To make an asynchronous HTTP request, please pass `async_req=True`:

```
>>> thread = api.create_node(body, async_req=True)
>>> result = thread.get()
```

:param `async_req` bool

:param `V1Node body`: (required)

:param `str pretty`: If 'true', then the output is pretty printed.

:param `str dry_run`: When present, indicates that modifications should not be persisted. An invalid or unrecognized dryRun directive will result in an error response and no further processing of the request. Valid values are:

- All: all dry run stages will be processed

:param `str field_manager`: `fieldManager` is a name associated with the actor or entity that is making these changes. The value must be less than or 128 characters long, and only contain printable characters, as defined by <https://golang.org/pkg/unicode/#IsPrint>.

:return: `V1Node`

If the method is called asynchronously,
returns the request thread.

You can poke around yourself and learn more about the API. Let's look at some common operations, such as listing, creating, watching, and deleting objects.

Listing objects

You can list different kinds of object. The method names start with `list_`. Here is an example listing all namespaces:

```
for ns in v1.list_namespace().items:
...   print(ns.metadata.name)
...
default
kube-public
kube-system
```

Creating objects

To create an object, you need to pass a body parameter to the `create` method. The body must be a Python dictionary that is equivalent to a YAML configuration file you would use with `Kubectl`. The easiest way to do it is to actually use a YAML file and then use the Python YAML module (not part of the standard library and must be installed separately) to read the YAML file and load it into a dictionary. For example, to create an `nginx-deployment` with three replicas, we can use this YAML configuration file (`nginx-deployment.yaml`):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.17.8
          ports:
            - containerPort: 80
```

To install the YAML Python module, type this command:

```
$ pip install pyyaml
```

Then the following Python 3 program (create_nginx_deployment.py) will create the deployment:

```
from os import path
import yaml from kubernetes import client, config
def main():
    # Configs can be set in Configuration class directly or using
    # helper utility. If no argument provided, the config will be
    # loaded from default location.
    config.load_kube_config()

    with open(path.join(path.dirname(__file__),
                        'nginx-deployment.yaml')) as f:

        dep = yaml.safe_load(f)
        k8s = client.AppsV1Api()
        dep = k8s.create_namespaced_deployment(body=dep,
                                              namespace="default")

        print(f"Deployment created. status='{dep.status}'")
    if __name__ == '__main__':
        main()
```

Watching objects

Watching objects is an advanced capability. It is implemented using a separate watch module. Here is an example to watch for 10 namespace events and print them to the screen (watch_demo.py):

```
from kubernetes import client, config, watch
# Configs can be set in Configuration class directly or using helper
utility
config.load_kube_config()
v1 = client.CoreV1Api()
count = 3
w = watch.Watch()
for event in w.stream(v1.list_namespace, _request_timeout=60):
    print(f"Event: {event['type']} {event['object'].metadata.name}")
    count -= 1 if count == 0: w.stop()
print('Done.')
```


Here is the output:

```
$ python watch_demo.py
Event: ADDED default
Event: ADDED kube-public
Event: ADDED kube-system
Done.
```

Invoking Kubectl programmatically

If you're not a Python developer and don't want to deal with the REST API directly or client libraries, you have another option. Kubectl is used mostly as an interactive command-line tool, but nothing is stopping you from automating it and invoking it through scripts and programs. There are some benefits to using Kubectl as your Kubernetes API client:

- It's easy to find examples for any usage.
- It's easy to experiment on the command line to find the right combination of commands and arguments.
- Kubectl supports output in JSON or YAML for quick parsing.
- Authentication is built in via Kubectl configuration.

Using Python subprocesses to run Kubectl

I'll use Python again, so you can compare using the official Python client versus rolling your own. Python has a module called subprocess that can run external processes such as Kubectl and capture the output. Here is a Python 3 example running Kubectl on its own and displaying the beginning of the usage output:

```
>>> import subprocess
>>> out = subprocess.check_output('kubectl').decode('utf-8')
>>> print(out[:276])
```

kubectl controls the Kubernetes cluster manager.

Find more information at: <https://kubernetes.io/docs/reference/kubectl/overview/>

Here are some basic Commands for beginners:

create	Create a resource from a file or from stdin.
expose	Take a replication controller, service

The `check_checkout()` function captures the output as a bytes array that needs to be decoded to UTF-8 to display it properly. We can generalize it a little bit and create a convenience function called `k()` in the `k.py` file. It accepts any number of arguments it feeds to `Kubect1`, and then decodes the output and returns it:

```
from subprocess import check_output
def k(*args):
    out = check_output(['kubect1'] + list(args))
    return out.decode('utf-8')
```

Let's use it to list all the running pods in the default namespace:

```
>>> from k import k
>>> print(k('get', 'po'))
```

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	4h48m
nginx-deployment-679f9c75b-c79mv	1/1	Running	0	132m
nginx-deployment-679f9c75b-cnvmk	1/1	Running	0	132m
nginx-deployment-679f9c75b-gzfgk	1/1	Running	0	132m

This is nice for display, but `Kubect1` already does that. The real power comes when you use the structured output options with the `-o` flag. Then the result can be converted automatically to a Python object. Here is a modified version of the `k()` function, which accepts a Boolean `use_json` keyword argument (default to `False`), and if `True`, adds `-o json`, and then parses the JSON output to a Python object (a dictionary):

```
from subprocess import check_output
import json

def k(*args, use_json=False):
    cmd = ['kubect1'] + list(args)
    if use_json:
        cmd += ['-o', 'json']
    out = check_output(cmd).decode('utf-8')
    if use_json:
        out = json.loads(out, )
    return out
```

That returns a full-fledged API object, which can be navigated and drilled down just like when accessing the REST API directly or using the official Python client:

```
>>> result = k('get', 'po', use_json=True)
>>> for r in result['items']:
...     print(r['metadata']['name'])
...
nginx-deployment-679f9c75b-c79mv
nginx-deployment-679f9c75b-cnmvk
nginx-deployment-679f9c75b-gzfgk
```

Let's see how to delete the deployment and wait until all the pods are gone. The Kubectl delete command doesn't accept the `-o json` option (although it has the `-o name`), so let's leave out `use_json`:

```
k('delete', 'deployment', 'nginx-deployment')
while len(k('get', 'po', use_json=True)['items']) > 0:
    print('.') print('Done.')
.
.
.
.
Done.
```

Now that we have accessed Kubernetes programmatically via its REST API and by controlling Kubectl, it's time to learn how to extend Kubernetes.

Extending the Kubernetes API

Kubernetes is an extremely flexible platform. It was designed from the get-go for extensibility, and as it evolved, more parts of Kubernetes were opened up, exposed through robust interfaces, and they can be replaced by alternative implementations. I venture to say that the exponential adoption of Kubernetes across the board by start-ups, large companies, infrastructure providers, and cloud providers is a direct result of Kubernetes providing a lot of capabilities out of the box, but also allowing easy integration with other actors. In this section, we will cover many of the available extensions points, such as the following:

- User-defined types (custom resources)
- API access extensions
- Infrastructure extensions
- Operators
- Scheduler extensions

Let's understand the various ways you can extend Kubernetes.

Understanding Kubernetes extension points and patterns

Kubernetes is made of multiple components: the API server, etcd state store, controller manager, kube-proxy, kubelet, and the container runtime. You can extend and customize deeply each and every one of these components as well as adding your own custom components that watch and react to events, handle new requests, and modify everything about incoming requests.

The following diagram shows some of the available extension points and how they are connected to various Kubernetes components:

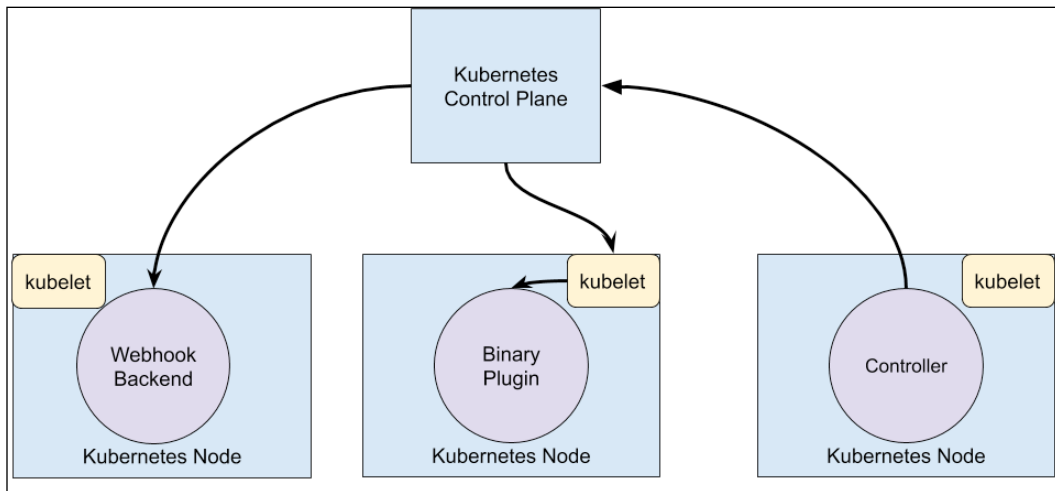


Figure 15.2: Extension points connecting to Kubernetes components

Extending Kubernetes with plugins

Kubernetes defines several interfaces that allow it to interact with a wide variety of plugins by infrastructure providers. We discussed some of these interfaces and plugins in detail in previous chapters. We will just list them here for completeness:

- **CNI:** The container networking interface supports a large number of networking solutions for connecting nodes and containers.
- **CSI:** The container storage interface supports a large number of storage options for Kubernetes.

- **Device plugins:** These allow a node to discover new node resources beyond CPU and memory (for example, GPU).

Extending Kubernetes with the cloud controller manager

Kubernetes needs to be deployed eventually on some nodes and use some storage and networking resources. Initially, Kubernetes supported only Google Cloud Platform and AWS. Other cloud providers had to customize multiple Kubernetes core components (Kubelet, Kubernetes Controller Manager, Kubernetes API server) in order to integrate with Kubernetes. The Kubernetes developers identified it as a problem for adoption and created the **Cloud Controller Manager (CCM)**. The CCM cleanly defines the interaction between Kubernetes and the infrastructure layer it is deployed on. Now, cloud providers just provide an implementation of the CCM tailored to their infrastructure and they can utilize upstream Kubernetes without costly and error-prone modifications to the Kubernetes code. All the Kubernetes components interact with the CCM via the predefined interfaces and Kubernetes is blissfully unaware which cloud (or no cloud) it is running on. The following diagram demonstrates the interaction between Kubernetes and a cloud provider via the CCM:

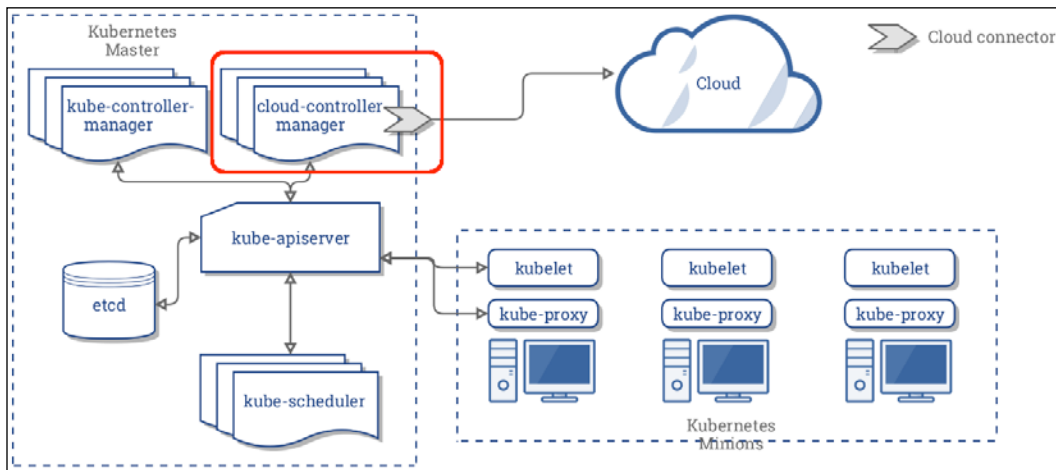


Figure 15.3: Interaction of the cloud and Kubernetes via a CCM

If you want to learn more about the CCM, check out this concise article I wrote a couple of years ago: <https://medium.com/@the.gigi/kubernetes-and-cloud-providers-b7a6227d3198>.

Extending Kubernetes with webhooks

Plugins run in the cluster, but in some cases a better extensibility pattern is to delegate a function to an out-of-cluster service. This is very common in the area of access control, where companies and organizations may already have a centralized solution for identity and access control. In those cases, the webhook extensibility pattern is useful. The idea is that you can configure Kubernetes with an endpoint (webhook). Kubernetes will call the endpoint where you can implement your own custom functionality and Kubernetes will take action based on the response. We've seen this pattern when we discussed authentication, authorization, and dynamic admission control.

Kubernetes defines the expected payloads for each webhook. The webhook implementation must adhere to them in order to successfully interact with Kubernetes.

Extending Kubernetes with controllers and operators

The controller pattern is where you write a program that can run inside the cluster or outside the cluster, watch for events, and respond to them. The conceptual model for a controller is to reconcile the current state of the cluster (the parts the controller is interested in) with a desired state. A common practice for controllers is to read the `.spec` of an object, take some actions, and update its `.status`. A lot of the core logic of Kubernetes is implemented by a large set of controllers managed by the controller manager, but there is nothing stopping us from deploying our own controllers to the cluster or running controllers that access the API server remotely.

The operator pattern is another flavor of the controller pattern. Think of an operator as a controller that also has its own set of custom resources that represents some application it manages. The goal of operators is to manage the lifecycle of some application that is deployed in the cluster. A great example is **etcd-operator**: <https://github.com/coreos/etcd-operator>.

If you plan to build your own controllers, I recommend starting with **kubebuilder** (<https://github.com/kubernetes-sigs/kubebuilder>). It is a project maintained by the Kubernetes API Machinery SIG and has support for defining multiple custom APIs using CRDs and scaffolds out the controller code to watch these resources.

For operators consider using the **Operator framework** (<https://github.com/operator-framework>) as your starting point.

Extending Kubernetes scheduling

Kubernetes' job, in one sentence, is to schedule pods to nodes. Scheduling is at the heart of what Kubernetes does and it does it well. The Kubernetes scheduler can be configured in very advanced ways (daemon sets, taints, tolerations, and so on). But the Kubernetes developers recognize that there may be extraordinary circumstances where you may want to control the core scheduling algorithm. It is possible to replace the core Kubernetes scheduler with your own scheduler or run another scheduler alongside the built-in scheduler to control the scheduling of a subset of the pods. We will see how to do that later in the chapter.

Extending Kubernetes with custom container runtimes

Kubernetes originally supported only Docker as a container runtime. The Docker support was embedded in the core Kubernetes codebase. Later, dedicated support for rkt was added. But the Kubernetes developers saw the light and introduced the CRI (container runtime interface), a gRPC interface, which enabled any container runtime that implements it to communicate with the kubelet. Eventually, the hard-coded support for Docker and rkt was phased out and now the kubelet talks to the container runtime only through CRI:

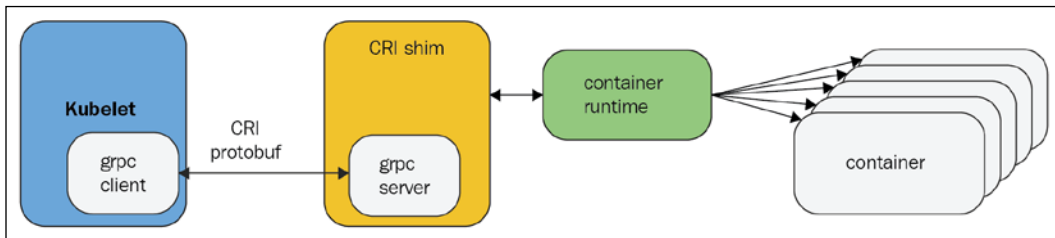


Figure 15.4: Kubelet communicating with container runtime

Since the introduction of CRI the number of container runtimes that work with Kubernetes exploded.

Introducing custom resources

One of the primary ways to extend Kubernetes is to define new types of resources called custom resources. What can you do with custom resources? Plenty. You can use them to manage resources that live outside the Kubernetes cluster, but with which your pods communicate, through the Kubernetes API. By adding those external resources as custom resources, you get a full picture of your system and you benefit from many Kubernetes API features, such as the following:

- Custom CRUD REST endpoints
- Versioning
- Watches
- Automatic integration with generic Kubernetes tooling

Other use cases for custom resources are metadata for custom controllers and automation programs.

Custom resources that were introduced in Kubernetes 1.7 are a significant improvement over the now deprecated third-party resources. Let's dive in and see what custom resources are all about.

In order to play nice with the Kubernetes API server, custom resources must conform to some basic requirements. Similar to built-in API objects, they must have the following fields:

- `apiVersion`: `apiextensions.k8s.io/v1`
- `metadata`: Standard Kubernetes object metadata
- `kind`: `CustomResourceDefinition`
- `spec`: Describes how the resource appears in the API and tools
- `status`: Indicates the current status of the CRD

The spec has an internal structure that includes fields such as group, names, scope, validation, and version. The status includes the fields `acceptedNames` and `Conditions`. In the next section, I'll show you an example that clarifies the meaning of these fields.

Developing custom resource definitions

You develop your custom resources using **Custom Resource Definitions (CRD)**. The intention is for CRDs to integrate smoothly with Kubernetes and its API and tooling, so you need to provide a lot of information. Here is an example for a custom resource called **Candy**:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form:
  # <plural>.<group>
  name: candies.awesome.corp.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
```



```
group: awesome.corp.com
# version name to use for REST API: /apis/<group>/<version>
versions:
  - name: v1
    # Each version can be enabled/disabled by Served flag.
    served: true
    # One and only one version must be marked as the storage version.
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              flavor:
                type: string
# either Namespaced or Cluster
scope: Namespaced
names:
  # plural name to be used in the URL: /
  apis/<group>/<version>/<plural>
  plural: candies
  # singular name to be used as an alias on the CLI and for display
  singular: candy
  # kind is normally the CamelCased singular type. Your resource
  manifests use this.
  kind: Candy
  # shortNames allow shorter string to match your resource on the CLI
  shortNames:
    - cn
```

The Candy CRD has several interesting parts. The metadata has a fully qualified name that should be unique since CRDs are cluster-scoped. The spec has a versions entry, which can contain multiple versions with a schema for each version that specifies the field of the custom resource. The schema follows the OpenAPI v3 (<https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#schemaObject>) specification.

The scope field could be either Namespaced or Cluster. If the scope is Namespaced then the custom resources you create from the CRD will exist only in the namespace they were created in; cluster-scoped custom resources are available in any namespace.

Finally, the names section refers to the names of the custom resource (not the name of the CRD from the metadata section). The names have plural, singular, kind, and short name options.

Let's create the CRD:

```
$ kubectl create -f candy-crd.yaml
customresourcedefinition.apiextensions.k8s.io/candies.awesome.corp.com
created
```

Note that the metadata name is returned. It is common to use a plural name. Now, let's verify we can access it:

```
$ kubectl get crd
NAME                                CREATED AT
candies.awesome.corp.com 2020-06-15T10:19:09Z
```

There is also an API endpoint for managing this new resource:

```
/apis/awesome.corp.com/v1/namespaces/<namespace>/candies/
```

Integrating custom resources

Once the CustomResourceDefinition object has been created, you can create custom resources of that resource kind, Candy in this case (candy becomes CamelCase Candy). Custom resources must respect the schema from the CRD. In the following example, the flavor field is set on the Candy object with a name of chocolate. The apiVersion field is derived from the CRD spec's group and version fields:

```
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: chocolate
spec:
  flavor: sweeeeeeeet
```

Let's create it:

```
$ k create -f chocolate.yaml
candy.awesome.corp.com/chocolate created
```

Note that the spec must contain the flavor field from the schema.

At this point, Kubectl can operate on Candy objects just like it works on built-in objects. Note that resource names are case-insensitive when using Kubectl:

```
$ kubectl get candies
```

```
NAME AGE chocolate 2m
```

We can also view the raw JSON data using the standard `-o json` flag. Let's use the short name `cn` this time:

```
$ kubectl get cn -o json
{
  "apiVersion": "v1",
  "items": [
    {
      "apiVersion": "awesome.corp.com/v1",
      "kind": "Candy",
      "metadata": {
        "creationTimestamp": "2020-06-15T10:22:25Z",
        "generation": 1,
        "name": "chocolate",
        "namespace": "default",
        "resourceVersion": "1664",
        "selfLink": "/apis/awesome.corp.com/v1/namespaces/default/candies/chocolate",
        "uid": "1b04f5a9-9ae8-475d-bc7d-245042759304"
      },
      "spec": {
        "flavor": "sweeeeeeeet"
      }
    }
  ],
  "kind": "List",
  "metadata": {
    "resourceVersion": "",
    "selfLink": ""
  }
}
```

Dealing with unknown fields

The schema in the spec was introduced with the `apiextensions.k8s.io/v1` version of CRDs that became stable in Kubernetes 1.17. With `apiextensions.k8s.io/v1beta` a schema wasn't required, so arbitrary fields were the way to go. If you just try to change the version of your CRD from `v1beta` to `v1`, you're in for a rude awakening. Kubernetes will let you update the CRD, but when you try to create a custom resource later with unknown fields it will fail.

You must define a schema for all your CRDs. If you must deal with custom resources that may have additional unknown fields you can turn validation off, but the additional fields will be stripped off.

Here is a Candy resource that has an extra field texture not specified in the schema:

```
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: gummy-bear
spec:
  flavor: delicious
  texture: rubbery
```

If we try to create it with validation it will fail:

```
$ kubectl create -f gummy-bear.yaml
error: error validating "gummy-bear.yaml": error validating data:
ValidationError(Candy.spec): unknown field "texture" in com.corp.awesome.
v1.Candy.spec; if you choose to ignore these errors, turn validation off
with --validate=false
```

But if we turn validation off then all is well, except that only the flavor field will be present and the texture field will not:

```
$ kubectl create -f gummy-bear.yaml --validate=false
candy.awesome.corp.com/gummy-bear created

$ kubectl get cn gummy-bear -o yaml
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  creationTimestamp: "2020-06-15T22:02:18Z"
  generation: 1
  name: gummy-bear
  namespace: default
  resourceVersion: "93551"
  selfLink: /apis/awesome.corp.com/v1/namespaces/default/candies/gummy-bear
  uid: 1900b97e-55ba-4235-8366-24f469f449e3
spec:
  flavor: delicious
```

If you want to add arbitrary fields, you need to turn validation off with `--validate=false`.

Finalizing custom resources

Custom resources support finalizers just like standard API objects. A finalizer is a mechanism where objects are not deleted immediately but have to wait for special controllers that run in the background and watch for deletion requests. The controller may perform any necessary cleanup options and then remove its finalizer from the target object. There may be multiple finalizers on an object. Kubernetes will wait until all finalizers have been removed and only then delete the object. The finalizers in the metadata are just arbitrary strings that their corresponding controller can identify. Kubernetes doesn't know what they mean. It just waits patiently for all the finalizers to be removed before deleting the object. Here is an example with a candy object that has two finalizers: eat-me and drink-me:

```
apiVersion: awesome.corp.com/v1
kind: Candy
metadata:
  name: chocolate
  finalizers:
    - eat-me
    - drink-me
spec:
  flavor: sweeeeeeeet
```

Adding custom printer columns

By default, when you list custom resources with Kubectl you get only the name and the age of the resource:

```
$ kubectl get cn
NAME          AGE
chocolate    11h
gummy-bear    16m
```

But the CRD schema allows you to add your own columns. Let's add the flavor and the age fields as printable columns to our candy objects:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: candies.awesome.corp.com
spec:
  group: awesome.corp.com
  versions:
    - name: v1
```

```

...
additionalPrinterColumns:
- name: Flavor
  type: string
  description: The flavor of the candy
  jsonPath: .spec.flavor
- name: Age
  type: date
  jsonPath: .metadata.creationTimestamp
...

```

Then we can create it, add our candies again, and list them:

```

$ kubectl create -f candy-with-flavor-crd.yaml
customresourcedefinition.apiextensions.k8s.io/candies.awesome.corp.com
created
$ kubectl create -f chocolate.yaml
candy.awesome.corp.com/chocolate created
$ kubectl create -f gummy-bear.yaml --validate=false
candy.awesome.corp.com/gummy-bear created
$ kubectl get candies

```

NAME	FLAVOR	AGE
chocolate	sweeeeeeeet	64s
gummy-bear	delicious	59s

Understanding API server aggregation

CRDs are great when all you need is some CRUD operations on your own types. You can just piggyback on the Kubernetes API server, which will store your objects and provide API support and integration with tooling such as Kubectl. If you need more power, you can run controllers that watch for your custom resources and perform some operations when they are created, updated, or deleted. The **kubebuilder** (<https://github.com/kubernetes-sigs/kubebuilder>) project is a great framework for building Kubernetes APIs on top of CRDs with your own controllers.

But CRDs have limitations. If you need more advanced features and customization, you can use API server aggregation and write your own API server that the Kubernetes API server will delegate to. Your API server will use the same API machinery as the Kubernetes API server itself. Some of the advanced capabilities that are only available through the aggregation layer are:

- Makes your apiserver adopt different storage APIs rather than etcd v3
- Extends long-running subresources/endpoints such as websockets for your own resources

- Integrates your apiserver with whatever other external systems
- Controls the storage of your objects (custom resources are always stored in etcd)
- Provides long-running resources such as websockets for your own resources
- Custom operations beyond CRUD (for example, exec or scale)
- Using protocol buffer payloads
- Integrates your API server with any external system

Writing an extension API server is a non-trivial effort. If you decide you need all that power, I recommend using the API builder alpha project: <https://github.com/kubernetes-sigs/apiserver-builder-alpha>.

It is a young project, but it takes care of a lot of the necessary boilerplate code. The API builder provides the following capabilities:

- Bootstrap complete type definitions, controllers, and tests, as well as documentation.
- The extension control plane you can run locally, inside minikube, or on an actual remote cluster.
- Your generated controllers will be able to watch and update API objects.
- Adding resources (including sub-resources).
- Default values you can override if needed.

Utilizing the service catalog

The Kubernetes **catalog** (<https://github.com/kubernetes-sigs/service-catalog>) project allows you to integrate smoothly and painlessly any external service that supports the **Open Service Broker API** specification: <https://github.com/openservicebrokerapi/servicebroker>.

The intention of the open service broker API is to expose external services to any cloud environment through a standard specification with supporting documentation and a comprehensive test suite. That lets providers implement a single specification and support multiple cloud environments. The current environments include Kubernetes and Cloud Foundry. The project works towards broad industry adoption.

The service catalog is useful in particular for integrating the services of cloud platform providers. Here are some examples of such services:

- Microsoft Azure Cloud Queue
- Amazon Simple Queue Service
- Google Cloud Pub/Sub

The following diagram describes the architecture and workflow of the service catalog, which is implemented as an API server extension using the aggregation layer:

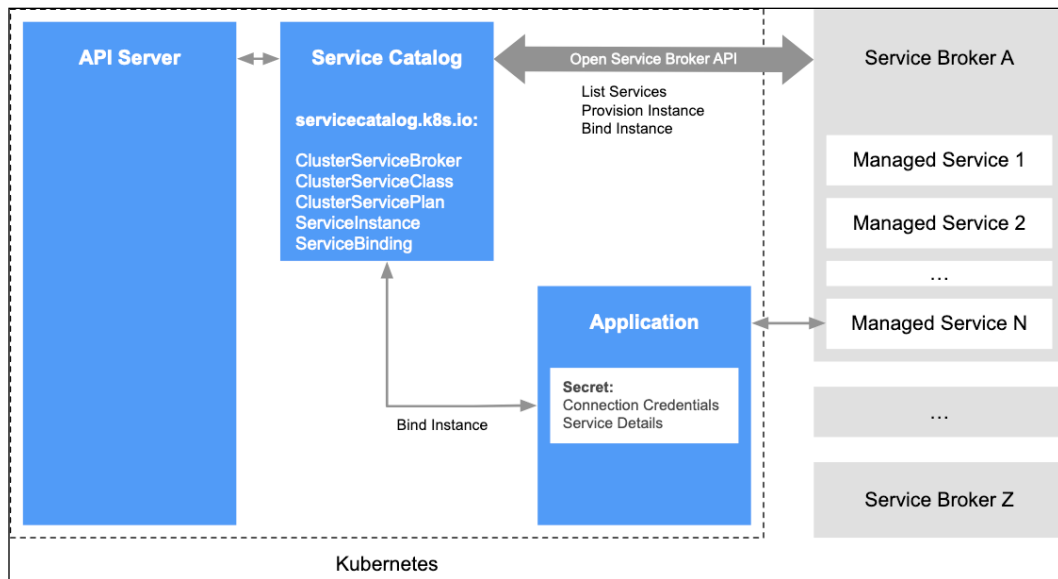


Figure 15.5: The service catalog

This capability is a real boon for organizations that are committed to the cloud. You get to build your system on Kubernetes, but you don't have to deploy, manage, and maintain every service in your cluster yourself. You can offload that to your cloud provider, enjoy deep integration, and focus on your application.

The service catalog can potentially make your Kubernetes cluster fully autonomous by allowing you to provision cloud resources through service brokers. We're not there yet, but the direction is very promising.

This concludes our discussion of accessing and extending Kubernetes from the outside. In the next section, we will direct our gaze inward and look into customizing the inner workings of Kubernetes itself via plugins.

Writing Kubernetes plugins

In this section, we will dive into the guts of Kubernetes and learn to take advantage of its famous flexibility and extensibility. We will learn about different aspects that can be customized via plugins and how to implement such plugins and integrate them with Kubernetes.

Writing a custom scheduler

Kubernetes is all about orchestrating containerized workloads. The most fundamental responsibility is to schedule pods to run on cluster nodes. Before we can write our own scheduler, we need to understand how scheduling works in Kubernetes.

Understanding the design of the Kubernetes scheduler

The Kubernetes scheduler has a very simple role. When a new pod needs to be created, it assigns it to a target node. That's it. The Kubelet on the target node will take it from there and instruct the container runtime on the node to run the pod's container.

The Kubernetes scheduler implements the controller pattern:

- Watch for pending pods
- Select the right node for the pod
- Update the node's spec by setting the `nodeName` field

The only complicated part is selecting the target node. This process involves two steps:

1. Filtering nodes
2. Ranking nodes

The scheduler takes a tremendous amount of information and configuration into account. Filtering removes nodes that don't satisfy one of the hard constraints from the candidate list. Ranking nodes assigns a score for each of the remaining nodes and chooses the best node.

Here are the factors the scheduler evaluates when filtering nodes:

- Checks if a node has free ports for the pod ports the pod is requesting
- Checks if a pod specifies a specific node by its hostname
- Checks if the node has enough resources (CPU, memory, and so on) to meet the requirement of the pod
- Checks if the pod's node selector matches the node's labels
- Evaluates if the volumes that the pod requests are available on the node, given the failure zone restrictions for that storage
- Checks if a pod can fit on a node due to the volumes it requests, and those that are already mounted
- Decides how many CSI volumes should be attached, and whether that's over a configured limit
- Checks if a node is reporting memory pressure
- Checks if a node is reporting that process IDs are scarce
- Checks if a node is reporting a filesystem is full or nearly full
- Checks other conditions reported by the node, like networking is unavailable or kubelet is not ready
- Checks if a pod's tolerations can tolerate the node's taints
- Checks if a pod can fit due to the volumes it requests

Once the nodes have been filtered the scheduler will score the nodes based on the following policies (that you can configure):

- Spread pods across hosts, considering pods that belong to the same service, StatefulSet or ReplicaSet.
- Inter-pod affinity priority.
- Least requested priority – favors nodes with fewer requested resources. This policy spreads pods across all nodes of the cluster.
- Most requestedPriority – favors nodes with the most requested resources. This policy will pack the pods into the smallest set of nodes.

- Requested to capacity ratio priority – creates a `requestedToCapacity` based `ResourceAllocationPriority` using a default resource scoring function shape.
- Balanced resource allocation – favors nodes with balanced resource usage.
- Node prefer avoid pods priority – prioritizes nodes according to the node annotation `scheduler.alpha.kubernetes.io/preferAvoidPods`. You can use this to hint that two different pods shouldn't run on the same node.
- Node affinity priority – prioritizes nodes according to node affinity scheduling preferences indicated in `PreferredDuringSchedulingIgnoredDuringExecution`.
- Taint toleration priority – prepares the priority list for all the nodes, based on the number of intolerable taints on the node. This policy adjusts a node's rank taking that list into account.
- Image locality priority – favors nodes that already have the container images the pod required.
- Service spreading priority – favors spreading the pods backing up a service across different nodes.
- Pod anti-affinity.
- Equal priority map – all nodes get the same weight. No favorites.

As you can see, the default scheduler is very sophisticated and can be configured in a very fine-grained way to accommodate most of your needs. But under some circumstances, it might not be the best choice. Particularly in large clusters with many nodes (hundreds or thousands), every time a pod is scheduled, all the nodes need to go through this rigorous and heavyweight procedure of filtering and scoring. Now, consider a situation where you need to schedule a large number of pods at once (for example, training machine learning models). This can put a lot of pressure on your cluster and lead to performance issues.

Kubernetes has recently introduced ways to make the filtering and scoring process more lightweight by allowing you to filter and score only some of the nodes, but still you may want better control.

Fortunately, Kubernetes allows us to influence the scheduling process in several ways. Those ways include the following:

- Direct scheduling of pods to nodes
- Replacing the scheduler with your own scheduler
- Extending the scheduler with additional filters
- Adding another scheduler that runs alongside the default scheduler

Scheduling pods manually

Guess what? We can just tell Kubernetes where to place our pod when we create it. All it takes is to specify a node name in the pod's spec and the scheduler will ignore it. If you think about the loosely coupled nature of the controller pattern, it all makes sense. The scheduler is watching for pending pods that DON'T have a node name assigned yet. If you are passing the node name yourself, then the Kubelet on the target node, who watches for pending pods that DO have a node name, will just go ahead and make sure to create a new pod.

Here is a pod with a pre-defined node name:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod-manual-scheduling
spec:
  containers:
  - name: some-container
    image: gcr.io/google_containers/pause:2.0
    nodeName: k3d-k3s-default-worker-1
  schedulerName: no-such-scheduler
```

If we create and describe the pod, we can see that it was indeed scheduled to the k3d-k3s-default-worker-1 node as requested:

```
$ kubectl describe po some-pod-manual-scheduling
Name: some-pod-manual-scheduling
Node: k3d-k3s-default-worker-1/172.19.0.4
Status: Running
...
Containers:
...
Conditions:
  Type           Status
  Initialized     True
  Ready          True
  ContainersReady True
  PodScheduled   True
Events:          <none>
```

Direct scheduling is also useful for troubleshooting, when you want to schedule a temporary pod to any tainted node without mucking around with adding tolerations.

Let's create our own custom scheduler now.

Preparing our own scheduler

Our scheduler will be super simple. It will just schedule all pending pods that request to be scheduled by the custom-scheduler to node k3d-k3s-default-worker-1. Here is the Python implementation that uses the Kubernetes client package:

```
from kubernetes import client, config, watch

def schedule_pod(cli, name):
    target = client.V1ObjectReference()
    target.kind = 'Node'
    target.apiVersion = 'v1'
    target.name = 'k3d-k3s-default-worker-1'
    meta = client.V1ObjectMeta()
    meta.name = name
    body = client.V1Binding(metadata=meta, target=target)
    return cli.create_namespaced_binding('default', body)

def main():
    config.load_kube_config()
    cli = client.CoreV1Api()
    w = watch.Watch()
    for event in w.stream(cli.list_namespaced_pod, 'default'):
        o = event['object']
        if o.status.phase != 'Pending' or o.spec.scheduler_name !=
'custom-scheduler':
            continue

        schedule_pod(cli, o.metadata.name)

if __name__ == '__main__':
    main()
```

If you want to run a custom scheduler long-term then you should deploy it into the cluster just like any other workload. But if you just want to play around with it or you're still developing your custom scheduler logic, you can run it locally as long as it has the correct credentials to access the cluster and have permissions to watch for pending pods and update their node name.

Assigning pods to the custom scheduler

OK. We have a custom scheduler that we can run alongside the default scheduler. But how does Kubernetes choose which scheduler to use to schedule a pod when there are multiple schedulers?

The answer is that Kubernetes doesn't care. The pod can specify which scheduler it wants to schedule it. The default scheduler will schedule any pod that doesn't specify the scheduler or that specifies explicitly the default-scheduler. Other custom schedulers should be responsible and only schedule pods that request them. If multiple schedulers try to schedule the same pod, we will probably end up with multiple copies or naming conflicts.

For example, our simple custom scheduler is looking for pending pods that specify a scheduler name of custom-scheduler. All other pods will be ignored by it:

```
if o.status.phase != 'Pending' or o.spec.scheduler_name != 'custom-
scheduler':
    continue
```

Here is a pod spec that specifies the custom-scheduler in its spec:

```
apiVersion: v1
kind: Pod
metadata:
  name: some-pod-with-custom-scheduler
spec:
  containers:
    - name: some-container
      image: gcr.io/google_containers/pause:2.0
  schedulerName: custom-scheduler
```

What happens if our custom scheduler is not running and we try to create this pod?

```
$ kubectl create -f some-pod-with-custom-scheduler.yaml
pod/some-pod-with-custom-scheduler created
```

```
$ kubectl get po
```

NAME	READY	STATUS	RESTARTS	AGE
some-pod-manual-scheduling	1/1	Running	0	23h
some-pod-with-custom-scheduler	0/1	Pending	0	25s

The pod is created just fine (meaning the Kubernetes API server stored it in etcd), but it is pending, which means it hasn't been scheduled yet. Since it specified an explicit scheduler, the default scheduler ignores it.

But if we run our scheduler... it will immediately get scheduled:

```
$ python custom_scheduler.py
Waiting for pods to schedule
Scheduling pod some-pod-with-custom-scheduler
```

Now we can see that the pod was assigned to a node and it is in a running state:

```
$ kubectl get pod -o wide
NAME                                READY  STATUS   IP            NODE
some-pod-manual-scheduling          1/1    Running  10.42.0.5     k3d-k3s-default-worker-1
some-pod-with-custom-scheduler      1/1    Running  10.42.0.8     k3d-k3s-default-worker-1
```

Verifying that the pods were scheduled using the correct scheduler

We can look at the pod events and see that, for pods scheduled using the default scheduler, you can expect the following events:

```
$ kubectl describe po some-pod | grep Events: -A 10
Events:
  Type    Reason      Age    From              Message
  ----    -
  Normal  Scheduled   <unknown>  default-scheduler
Successfully assigned default/some-pod to k3d-k3s-default-worker-0
  Normal  Pulled      12m     kubelet, k3d-k3s-default-worker-0
Container image "gcr.io/google_containers/pause:2.0" already present on
machine
  Normal  Created     12m     kubelet, k3d-k3s-default-worker-0  Created
container some-container
  Normal  Started     12m     kubelet, k3d-k3s-default-worker-0  Started
container some-container
```

But for our custom scheduler, there is no Scheduled event:

```
$ k describe po some-pod-with-custom-scheduler | grep Events: -A 10
Events:
  Type    Reason    Age    From              Message
```

```

-----
Normal    Pulled    22m    kubelet, k3d-k3s-default-worker-1    Container image
"gcr.io/google_containers/pause:2.0" already present on machine
Normal    Created    22m    kubelet, k3d-k3s-default-worker-1    Created
container some-container
Normal    Started    22m    kubelet, k3d-k3s-default-worker-1    Started
container some-container

```

That was a deep dive into scheduling and custom schedulers. Let's check out Kubectl plugins.

Writing Kubectl plugins

Kubectl is the workhorse of the aspiring Kubernetes developer and admin. There are now very good visual tools, such as **k9s** (<https://github.com/derailed/k9s>), **octant** (<https://github.com/vmware-tanzu/octant>), and of course the Kubernetes dashboard. But Kubectl is the most complete way to work interactively with your cluster and participate in automation workflows.

Kubectl encompasses an impressive list of capabilities, but you will often need to string together multiple commands or a long chain of parameters to accomplish some tasks. You may also want to run some additional tools installed in your cluster.

You can package such functionality as scripts or containers or any other way, but then you'll run into the issue of where to place them, how to discover them, and how to manage them. The Kubectl plugin gives you a one-stop shop for those extended capabilities. For example, recently I needed to periodically list and move around files on the SFTP server managed by a containerized application running on a Kubernetes cluster. I quickly wrote a few Kubectl plugins that took advantage of my KUBECONFIG credentials to get access to secrets in the cluster that contained the credentials to access the SFTP server and then implemented a lot of application-specific logic for accessing and managing those SFTP directories and files.

Understanding Kubectl plugins

Until Kubernetes 1.12, Kubectl plugins required a dedicated YAML file where you specified various metadata and other files that implemented the functionality. In Kubernetes 1.12, Kubectl started using the Git extension model, where any executable on your path with the prefix `kubectl-` is treated as a plugin.

Kubectl provides the `kubectl plugins list` command to list all your current plugins. This model was very successful with Git, and it is extremely simple now to add your own Kubectl plugins.

If you added an executable called `kubect1-foo`, then you can run it via `kubect1 foo`. You can have nested commands too. Add `kubect1-foo-bar` to your path and run it via `kubect1 foo bar`. If you want to use dashes in your commands, then use underscores in your executable. For example, the executable `kubect1-do_stuff` can be run using `kubect1 do-stuff`.

The executable itself can be implemented in any language, have its own command-line arguments and flags, and display its own usage and help information.

Managing Kubect1 plugins with Krew

The lightweight plugin model is great for writing your own plugins, but what if you want to share your plugins with the community?

Krew (<https://github.com/kubernetes-sigs/krew>) is a package manager for Kubect1 plugins that lets you discover, install, and manage curated plugins.

You can install Krew with `brew` on Mac or follow the installation instructions for other platforms. Krew is itself a Kubect1 plugin as its executable is `kubect1-krew`. This means you can either run it directly, `kubect1-krew`, or through `kubect1 krew`. If you have a `k` alias for `kubect1` you would probably prefer the latter:

```
$ k krew
krew is the kubect1 plugin manager.
You can invoke krew through kubect1: "kubect1 krew [command]..."
```

```
Usage:
  krew [command]
```

Available Commands:

```
help      Help about any command
info      Show information about a kubect1 plugin
install   Install kubect1 plugins
list      List installed kubect1 plugins
search    Discover kubect1 plugins
uninstall Uninstall plugins
update    Update the local copy of the plugin index
upgrade   Upgrade installed plugins to newer versions
version   Show krew version and diagnostics
```

Flags:

```
-h, --help help for krew
-v, --v Level number for the log level verbosity
```

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

Note that the `krew list` command shows only Krew-managed plugins and not all Kubectl plugins. It doesn't even show itself!

I recommend that you check out the available plugins. Some of them are very useful, and they may give you some ideas for writing your own plugins. Let's see how easy it is to write our own plugin.

Creating your own Kubectl plugin

I was recently handed the unpleasant job of baby-sitting a critical application deployed in a Kubernetes cluster on GKE. The original development team wasn't around anymore and there were higher priorities than migrating it into our infrastructure on AWS. I noticed that some of the deployments owned multiple replica sets, but the desired number of replicas is zero. There was nobody around to ask why these replica sets were still around. My guess was that it was an artifact or updates where the old replica set was scaled down to zero, while the new replica set was scaled up, but the old replica set was left there hanging off the deployment. Anyway, let's write a Kubectl plugin that lists stale replica sets with zero replicas.

It turns out to be super simple. Let's use Python and the excellent `sh` module, which lets us run command-line commands naturally from Python. In this case, we're just going to run Kubectl itself and get all the replica sets with a custom columns format, then we're going to keep the replica sets that have zero replicas and display them with their owning deployment:

```
#!/usr/bin/env python3
import sh

def main():
    """ """
    o = "-o custom-columns='NAME:.metadata.name,DEPLOYMENT:.metadata.ownerReferences[0].name,REPLICAS:.spec.replicas'"
    all_rs = sh.kubectl.get.rs(o.split()).stdout.decode('utf-8').split('\n')
    all_rs = [r.split() for r in all_rs if r]
    results = ((name, deployment) for (name, deployment, replicas) in all_rs[1:] if replicas == '0')

    for name, deployment in results:
        print(name, deployment)

if __name__ == '__main__':
    main()
```

We can name the file `kubectl-show-stale_replica_sets` and run it with `kubectl show stale-replica-sets`. Before running it, we mustn't forget to make it executable and copy it to the PATH:

```
$ chmod +x kubectl-show-stale_replica_sets
$ kubectl show stale-replica-sets
cool-app-559f7bd67c cool-app
cool-app-55dc8c5949 cool-app
mice-app-5bd847f99c nice-app
```

If you want to develop plugins and share them on Krew, there is a more rigorous process there. I highly recommend developing the plugin in Go and taking advantage of projects such as the Kubernetes **cli-runtime** (<https://github.com/kubernetes/cli-runtime/>) and the **krew-plugin-template** (<https://github.com/replicatedhq/krew-plugin-template>) projects.

Kubectl plugins are awesome, but there are some gotchas you should be aware of.

Kubectl plugin gotchas

I ran into some of these issues when working with Kubectl plugins.

Don't forget your shebangs!

If you don't specify a shebang for your executable you will get an obscure error message:

```
$ k show stale-replica-sets
exec format error
```

Naming

Choosing a name for your plugin is not easy. Luckily there are some good guidelines: https://github.com/kubernetes-sigs/krew/blob/master/docs/NAMING_GUIDE.md.

Those naming guidelines are not just for Krew plugins, but make sense for any Kubectl plugin.

Overriding existing Kubectl commands

I originally named the plugin `kubectl-get-stale_replica_sets`. In theory, Kubectl should try to match the longest plugin name to resolve ambiguities. But, apparently, it doesn't work with built-in commands such as `kubectl get`.

This is the error I got:

```
$ kubectl get stale-replica-sets
error: the server doesn't have a resource type "stale-replica-sets"
```

Renaming `kubectl-show-stale_replica_sets` solved the problem.

Flat namespace for Krew plugins

The space of Kubectl plugins is flat. If you choose a generic plugin name such as `kubectl-login` you'll have a lot of problems. Even if you qualify it with something like `kubectl-gcp-login`, you might conflict with some other plugin. This is a scalability problem. I think the solution should involve some strong naming scheme for plugins, such as DNS, and then we should be able to define short names and aliases for convenience.

Let's now take a look at how to extend access control with webhooks.

Employing access control webhooks

Kubernetes always provided ways for you to customize access control. In Kubernetes, access control can be denoted as triple-A: Authentication, Authorization, and Admission control. In early versions it was through plugins that required Go programming, installing into your cluster, registration, and other invasive procedures. Now, Kubernetes lets you customize authentication, authorization, and admission control webhooks. Here is the access control workflow:

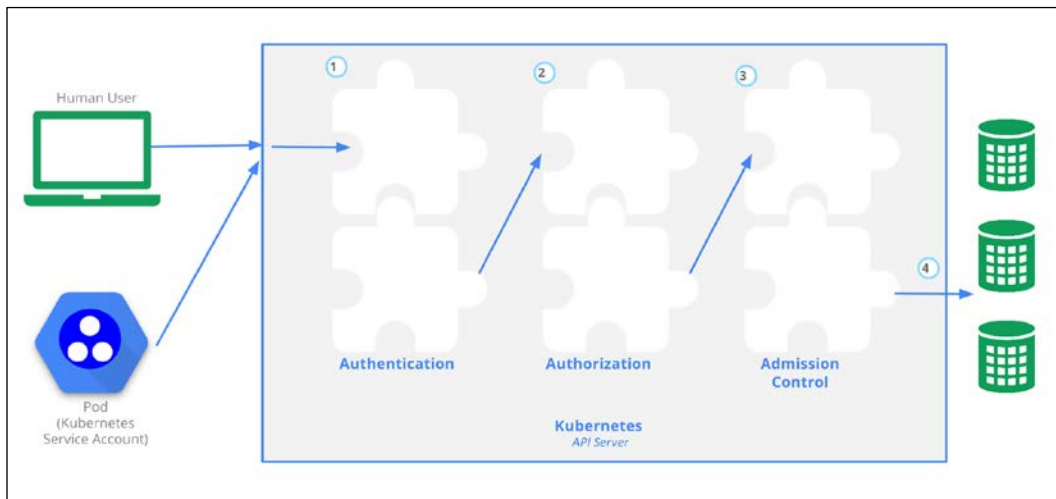


Figure 15.6: Kubernetes access control workflow

Using an authentication webhook

Kubernetes lets you extend the authentication process by injecting a webhook for bearer tokens. It requires two pieces of information: how to access the remote authentication service and the duration of the authentication decision (it defaults to two minutes).

To provide this information and enable authentication webhooks, start the API server with the following command-line arguments:

- `--authentication-token-webhook-config-file=`
- `--authentication-token-webhook-cache-ttl` (how long to cache auth decisions, default to 2 minutes)

The configuration file uses the kubeconfig file format. Here is an example:

```
# Kubernetes API version
apiVersion: v1
# kind of the API object
kind: Config
# clusters refers to the remote service.
clusters:
  - name: name-of-remote-authn-service
    cluster:
      certificate-authority: /path/to/ca.pem          # CA for verifying
the remote service.
      server: https://authn.example.com/authenticate # URL of remote
service to query. Must use 'https'.

# users refers to the API server's webhook configuration.
users:
  - name: name-of-api-server
    user:
      client-certificate: /path/to/cert.pem # cert for the webhook
plugin to use      client-key: /path/to/key.pem      # key
matching the cert

# kubeconfig files require a context. Provide one for the API server.
current-context: webhook
contexts:
  - context:
      cluster: name-of-remote-authn-service
      user: name-of-api-sever
      name: webhook
```

Note that a client certificate and key must be provided to Kubernetes for mutual authentication against the remote authentication service.

The cache TTL is useful because often users will make multiple consecutive requests to Kubernetes. Having the authentication decision cached can save a lot of round trips to the remote authentication service.

When an API HTTP request comes in, Kubernetes extracts the bearer token from its headers and posts a `TokenReview` JSON request to the remote authentication service via the webhook:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "<bearer token from original request headers>"
  }
}
```

The remote authentication service will respond with a decision. The status of authentication will either be `true` or `false`. Here is an example of a successful authentication:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      "username": "gigi@gg.com",
      "uid": "42",
      "groups": [
        "developers",
      ],
      "extra": {
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    }
  }
}
```

A rejected response is much more concise:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false
  }
}
```

Using an authorization webhook

The authorization webhook is very similar to the authentication webhook. It requires just a configuration file, which is in the same format as the authentication webhook configuration file. There is no authorization caching because unlike authentication, the same user may make lots of requests to different API endpoints with different parameters and authorization decisions may be different, so caching is not a viable option.

You configure the webhook by passing the following command-line argument to the API server:

--authorization-webhook-config-file=<configuration filename>

When a request passes authentication, Kubernetes will send a `SubjectAccessReview` JSON object to the remote authorization service. It will contain the requesting user (and any user groups it belongs to) and other attributes:

- requested API group
- namespace
- resource
- verb

```
{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "resourceAttributes": {
      "namespace": "awesome-namespace",
      "verb": "get",
      "group": "awesome.example.org",
      "resource": "pods"
    },
  },
}
```

```

    "user": "gigi@gg.com",
    "group": [
        "group1",
        "group2"
    ]
}

```

The request may be allowed:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": true
  }
}

```

Or it may be denied with a reason:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "status": {
    "allowed": false,
    "reason": "user does not have read access to the namespace"
  }
}

```

A user may be authorized to access a resource, but not some non-resource attributes, such as `/api`, `/apis`, `/metrics`, `/resetMetrics`, `/logs`, `/debug`, `/healthz`, `/swagger-ui`, `/swaggerapi`, `/ui`, and `/version`.

Here is how to request access to the logs:

```

{
  "apiVersion": "authorization.k8s.io/v1beta1",
  "kind": "SubjectAccessReview",
  "spec": {
    "nonResourceAttributes": {
      "path": "/logs",
      "verb": "get"
    },
  },
  "user": "gigi@gg.com",
}

```



```
    "group": [  
      "group1",  
      "group2"  
    ]  
  }  
}
```

We can check using `kubectl` if we are authorized to perform some operation using the `can-i` command. For example, let's see if we can create deployments:

```
$ kubectl auth can-i create deployments  
yes
```

We can also check if other users or service accounts are authorized to do something. The default service account is NOT allowed to create deployments:

```
$ kubectl auth can-i create deployments --as default  
no
```

Using an admission control webhook

Dynamic admission control supports webhooks too. It is generally available since Kubernetes 1.16. You need to enable the `MutatingAdmissionWebhook` and `ValidatingAdmissionWebhook` admission controllers using `--enable-admission-plugins=Mutating,ValidatingAdmissionWebhook`. There are several other admission controllers that the Kubernetes developers recommend you run (the order matters):

```
--admission-control=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultS  
torageClass,DefaultTolerationSeconds,MutatingAdmissionWebhook,ValidatingAd  
missionWebhook,ResourceQuota
```

Configuring a webhook admission controller on the fly

The authentication and authorization webhooks must be configured when you start the API server. The admission control webhooks can be configured dynamically by creating `MutatingWebhookConfiguration` or `ValidatingWebhookConfiguration` API objects. Here is an example:

```
apiVersion: admissionregistration.k8s.io/v1  
kind: ValidatingWebhookConfiguration  
...  
webhooks:  
- name: admission-webhook.example.com
```

```

rules:
- operations: ["CREATE", "UPDATE"]
  apiGroups: ["apps"]
  apiVersions: ["v1", "v1beta1"]
  resources: ["deployments", "replicasets"]
  scope: "Namespaced"
...

```

An admission server accesses AdmissionReview requests such as:

```

{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "request": {
    "uid": "705ab4f5-6393-11e8-b7cc-42010a800002",
    "kind": {"group":"autoscaling","version":"v1","kind":"Scale"},
    "resource": {"group":"apps","version":"v1","resource":"deploymen
ts"},
    "subResource": "scale",
    "requestKind": {"group":"autoscaling","version":"v1","kind":"Sca
le"},
    "requestResource": {"group":"apps","version":"v1","resource":"deplo
yments"},
    "requestSubResource": "scale",
    "name": "cool-deployment",
    "namespace": "cool-namespace",
    "operation": "UPDATE",
    "userInfo": {
      "username": "admin",
      "uid": "014fbff9a07c",
      "groups": ["system:authenticated","my-admin-group"],
      "extra": {
        "some-key":["some-value1", "some-value2"]
      }
    }
  },

  "object": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
  "oldObject": {"apiVersion":"autoscaling/v1","kind":"Scale",...},
  "options": {"apiVersion":"meta.k8s.io/v1","kind":"UpdateOptio
ns",...},
  "dryRun": false
}

```

If the request is admitted the response will be as follows:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": true
  }
}
```

If the request is not admitted, then allowed will be False. The admission server may provide a status section too with an HTTP status code and message:

```
{
  "apiVersion": "admission.k8s.io/v1",
  "kind": "AdmissionReview",
  "response": {
    "uid": "<value from request.uid>",
    "allowed": false,
    "status": {
      "code": 403,
      "message": "You cannot do this because I say so!!!"
    }
  }
}
```

Providing custom metrics for horizontal pod autoscaling

Prior to Kubernetes 1.6 custom metrics were implemented as a Heapster model. In Kubernetes 1.6 new custom metrics APIs landed and matured gradually. As of Kubernetes 1.9 they are enabled by default. Custom metrics rely on API aggregation. The recommended path is to start with the custom metrics API server boilerplate available here: <https://github.com/kubernetes-sigs/custom-metrics-apiserver>.

Then, you implement the CustomMetricsProvider interface:

```
type CustomMetricsProvider interface {
    // GetMetricByName fetches a particular metric for a particular
    // object.
    // The namespace will be empty if the metric is root-scoped.
```

```

    GetMetricByName(name types.NamespacedName,
                    info CustomMetricInfo,
                    metricSelector labels.Selector) (*custom_metrics.
MetricValue, error)

    // GetMetricBySelector fetches a particular metric for a set of
objects matching
    // the given label selector. The namespace will be empty if the
metric is root-scoped.
    GetMetricBySelector(namespace string, selector labels.Selector,
info CustomMetricInfo, metricSelector labels.Selector) (*custom_
metrics.MetricValueList, error)

    // ListAllMetrics provides a list of all available metrics at
// the current time. Note that this is not allowed to return
// an error, so it is recommended that implementors cache and
// periodically update this list, instead of querying every time.
    ListAllMetrics() []CustomMetricInfo
}

```

Extending Kubernetes with custom storage

Volume plugins are yet another type of plugin. Prior to Kubernetes 1.8 you had to write a kubelet plugin that required registration with Kubernetes and linking with the Kubelet. Kubernetes 1.8 introduced FlexVolume, which is much more versatile. Kubernetes 1.9 took it to the next level with the **Container Storage Interface (CSI)** that we covered in *Chapter 6, Managing Storage*. At this point, if you need to write storage plugins, CSI is the way to go. Since CSI uses the gRPC protocol, the CSI plugin must implement the following gRPC interface:

```

service Controller {
    rpc CreateVolume (CreateVolumeRequest)
        returns (CreateVolumeResponse) {}

    rpc DeleteVolume (DeleteVolumeRequest)
        returns (DeleteVolumeResponse) {}

    rpc ControllerPublishVolume (ControllerPublishVolumeRequest)
        returns (ControllerPublishVolumeResponse) {}

    rpc ControllerUnpublishVolume (ControllerUnpublishVolumeRequest)
        returns (ControllerUnpublishVolumeResponse) {}
}

```

```
rpc ValidateVolumeCapabilities (ValidateVolumeCapabilitiesRequest)
    returns (ValidateVolumeCapabilitiesResponse) {}

rpc ListVolumes (ListVolumesRequest)
    returns (ListVolumesResponse) {}

rpc GetCapacity (GetCapacityRequest)
    returns (GetCapacityResponse) {}

rpc ControllerGetCapabilities (ControllerGetCapabilitiesRequest)
    returns (ControllerGetCapabilitiesResponse) {}
}
```

This is not a trivial undertaking, and typically only storage solution providers should implement CSI plugins.

Summary

In this chapter, we covered three major topics: working with the Kubernetes API, extending the Kubernetes API, and writing Kubernetes plugins. The Kubernetes API supports the OpenAPI spec and is a great example of REST API design that follows all current best practices. It is very consistent, well organized, and well documented. Yet it is a big API and not easy to understand. You can access the API directly via REST over HTTP, using client libraries including the official Python client, and even by invoking Kubectl.

Extending the Kubernetes API involves defining your own custom resources and optionally extending the API server itself via API aggregation. Custom resources are most effective when you combine them with additional custom plugins or controllers when you query and update them externally.

Plugins and webhooks are the foundation of Kubernetes design. Kubernetes was always meant to be extended by users to accommodate any need. We looked at various plugins, such as custom schedulers, Kubectl plugins, and access control webhooks. It is very cool that Kubernetes provides such a seamless experience for writing, registering, and integrating all those plugins.

We also looked at custom metrics and even how to extend Kubernetes with custom storage options.

At this point, you should be well aware of all the major mechanisms to extend, customize, and control Kubernetes via API access, custom resources, controllers, operators, and custom plugins. You are in a great position to take advantage of these capabilities to augment the existing functionality of Kubernetes and adapt it to your needs and your systems.

In the next chapter, which will conclude the book, we will look at the future of Kubernetes and the road ahead. Spoiler alert – the future is very bright. Kubernetes has established itself as the gold standard for cloud native computing. It is being used across the board and it keeps evolving responsibly. An entire support system has developed around Kubernetes, including training, open source projects, tools, and products. The community is amazing and the momentum is very strong.

16

The Future of Kubernetes

In this chapter, we'll look at the future of Kubernetes from multiple angles. We'll start with the momentum of Kubernetes since its inception, across dimensions such as community, ecosystem, and mindshare. Spoiler alert – Kubernetes won the container orchestration wars by a landslide. As Kubernetes grows and matures, the battle lines shift from beating competitors to fighting against its own complexity. Usability, tooling, and education will play a major role as container orchestration is still new, fast-moving, and not a well-understood domain. Then, we will take a look at some very interesting patterns and trends and finally, we will review my predictions from the 2nd edition and make some new ones.

The topics we'll cover are as follows:

- The Kubernetes momentum
- The importance of CNCF
- Kubernetes extensibility
- Service mesh integration
- Serverless computing on Kubernetes
- Kubernetes and VMs
- Cluster autoscaling
- Ubiquitous operators

The Kubernetes momentum

Kubernetes is undeniably a juggernaut. Not only did Kubernetes beat all the other container orchestrators, but it is also the de facto solution on public clouds, utilized in many private clouds, and even VMware – the virtual machine company – is focused on Kubernetes solutions and integrating its products with Kubernetes.

Kubernetes works very well in multi-cloud and hybrid cloud scenarios due to its extensible design.

In addition, Kubernetes makes inroads in the edge too, with custom distributions that expand its broad applicability even more.

The Kubernetes project continues to release new version every 3 months like clockwork. The community just keeps growing.

The Kubernetes GitHub repository (<https://github.com/kubernetes/kubernetes>) has 64,000 stars and one of the most major drivers of this phenomenal growth is the **Cloud Native Computing Foundation (CNCF)**.

The importance of the CNCF

The CNCF has become a very important organization in the cloud computing scene. While it is not Kubernetes-specific, the dominance of Kubernetes is undeniable. Kubernetes is the first project to graduate from it, and most of the other projects lean heavily toward Kubernetes. In particular, the CNCF offers certification and training only for Kubernetes. The CNCF, among other roles, ensures the cloud technologies will not suffer from vendor lock-in. Check out this crazy diagram of the entire CNCF landscape: <https://landscape.cncf.io/zoom=60>.

Project curation

The CNCF (<https://www.cncf.io/>) assigns three maturity levels to projects: graduated, incubating, and sandbox:

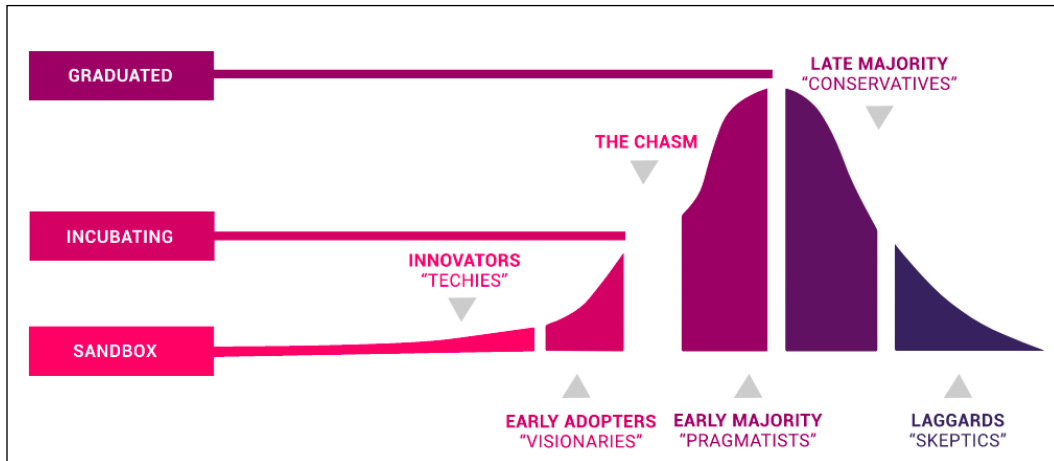


Figure 16.1: Project maturity levels

Projects (<https://www.cncf.io/projects/>) start at a certain level – sandbox or incubating – and over time can graduate. That doesn't mean that only graduated projects can be safely used. Many incubating and even sandbox projects are used heavily in production. For example, etcd is the persistent state store of Kubernetes itself and it is just an incubating project. Obviously, it is a highly trusted component. Virtual Kubelet is a sandbox project that powers AWS Fargate and Microsoft ACI. These are clearly enterprise-grade pieces of software.

The main benefit of the CNCF curation of projects is to help navigate the incredible eco-system that grew around Kubernetes. When you start looking to extend your Kubernetes solution with additional technologies and tools, the CNCF projects are a good place to start.

Certification

When technologies start to offer certification programs, you can tell they are here to stay. The CNCF offers several types of certifications:

- **Certified Kubernetes** (<https://www.cncf.io/certification/software-conformance/>), for conforming Kubernetes distributions and installers (about 90); **Kubernetes Certified Service Provider (KCSP)** (<https://www.cncf.io/certification/kcsp/>), for vetted service providers with deep Kubernetes experience (134 providers); and **Certified Kubernetes Administrator (CKA)** (<https://www.cncf.io/certification/cka/>), for administrators.
- **Certified Kubernetes Application Developer (CKAD)** (<https://www.cncf.io/certification/ckad/>) for developers.

Training

The CNCF offers training (<https://www.cncf.io/certification/training/>) too. There is a free introduction to the Kubernetes course and several paid courses that align with the CKA and CKAD certification exams. In addition, the CNCF maintains a list of Kubernetes training partners (<https://landscape.cncf.io/category=kubernetes-training-partner&format=card-mode&grouping=category>).

If you're looking for free Kubernetes training, here are a couple of options:

- VMware Kubernetes academy (<https://kubernetes.academy/>)
- Google Kubernetes Engine on Coursera (<https://www.coursera.org/learn/google-kubernetes-engine>)

Community and education

The CNCF also organizes conferences like KubeCon, CloudNativeCon, and meetups and maintains several communication avenues like slack channels and mailing lists. It also publishes surveys and reports.

The numbers of attendees and participants keeps growing year over year.

Tooling

The number of tools to manage containers and clusters, the various add-ons, extensions, and plugins just keeps growing and growing. Here is a subset of the tools, projects, and companies that participate in the Kubernetes ecosystem:

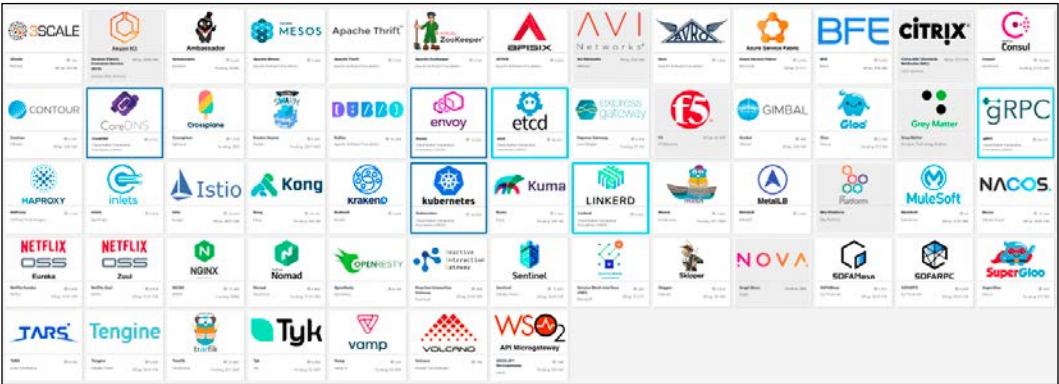


Figure 16.2: Participants in the Kubernetes ecosystem

The rise of managed Kubernetes platforms

Pretty much every cloud provider has a solid managed Kubernetes offering these days. Sometimes, there are multiple flavors and ways of running Kubernetes on a given cloud provider.

Public cloud Kubernetes platforms

Here are some of the prominent managed platforms:

- Google GKE (<https://cloud.google.com/kubernetes-engine/>)
- Microsoft AKS (<https://azure.microsoft.com/en-us/services/kubernetes-service/>)
- Amazon EKS (<https://aws.amazon.com/eks/>)
- Digital Ocean (<https://www.digitalocean.com/products/kubernetes/>)
- Oracle Cloud (<https://www.oracle.com/cloud/compute/container-engine-kubernetes.html>)
- IBM Cloud Kubernetes service (<https://www.ibm.com/cloud/container-service/>)
- Alibaba ACK (<https://www.alibabacloud.com/product/kubernetes>)
- Tencent TKE (<https://intl.cloud.tencent.com/product/tke>)

Of course, you can always roll your own and use the public cloud providers just as infrastructure providers. This is a very common use case with Kubernetes.

Bare-metal, private clouds, and Kubernetes on the edge

Here, you can find Kubernetes distributions that are designed or configured to run in special environments, often in your own data centers as a private cloud or in more restricted environments like edge computing on small devices:

- Google Anthos for GKE (<https://cloud.google.com/anthos/gke/>)
- OpenStack (<https://docs.openstack.org/openstack-helm/latest/install/developer/kubernetes-and-common-setup.html>)

- Rancher k3S (<https://rancher.com/docs/k3s/latest/en/>)
- Kubernetes on Raspberry PI (<https://www.shogan.co.uk/kubernetes/building-a-raspberry-pi-kubernetes-cluster-part-1-routing/>)
- KubeEdge (<https://kubedge.io/en/>)

Kubernetes Platform as a Service (PaaS)

This category of offerings aims to abstract some of the complexity of Kubernetes and put a simpler facade in front of it. There are many varieties here. Some of them cater to the multi-cloud and hybrid cloud scenarios, some expose the function as a service interface, while some just focus on a better installation and support experience:

- Google Cloud Run (<https://cloud.google.com/run/>)
- VMware PKS (<https://tanzu.vmware.com/kubernetes-grid>)
- Platform 9 PMK (<https://platform9.com/managed-kubernetes/>)
- Giant Swarm (<https://www.giantswarm.io/>)
- OpenShift (<https://www.openshift.com/>)
- Rancher RKE (<https://rancher.com/docs/rke/latest/en/>)

Upcoming trends

Let's talk about some of technological trends in Kubernetes that will be important in the near future. Some of these trends are already there.

Security

Security is, of course, a paramount concern for large-scale systems. Kubernetes is primarily a platform for managing containerized workloads. Those containerized workloads are often run in a multi-tenant environment. The isolation between tenants is super important. Containers are lightweight and efficient because they share an OS and maintain their isolation through various mechanisms like namespace isolation, filesystem isolation, and cgroup resource isolation. In theory, this should be enough. In practice, the surface area is large and there were multiple breakouts of container isolation.

To address this risk, multiple lightweight VMs were designed to add a hypervisor (machine-level virtualization) as an additional isolation level between the container and the OS kernel. The big cloud providers already support these technologies, and the Kubernetes CRI interface provides a streamlined way to take advantage of these more secure runtimes.

For example, **FireCracker** (<https://firecracker-microvm.github.io/>) is integrated with containerd via **firecracker-containerd** (<https://github.com/firecracker-microvm/firecracker-containerd>). Google gVisor is another sandbox technology. It is a user space kernel that implements most of the Linux system calls and provides a buffer between the application and the host OS. It is also available through containerd via **gvisor-containerd-shim** (<https://github.com/google/gvisor-containerd-shim>).

Networking

Networking is another area that is an ongoing source of innovation. The Kubernetes CNI allows any number of innovative networking solutions to work behind a simple interface. A major theme is incorporating eBPF – a relatively new Linux kernel technology – into Kubernetes.

eBPF stands for **extended Berkeley Packet Filter**. The core of eBPF is a mini-VM in the Linux kernel that executes special programs attached to kernel objects when certain events occur, such as a packet being transmitted or received. Originally, only sockets were supported, and the technology was called just BPF. Later, additional objects were added to the mix, and that's when the "e" for "extended" came along. eBPF's claim to fame is its performance due to the fact it runs highly-optimized, compiled BPF programs in the kernel and doesn't require extending the kernel with kernel modules.

There are many applications for eBPF:

- **Dynamic network control:** An iptables-based approach doesn't scale very well in a dynamic environment like a Kubernetes cluster where replacing iptables with BPF programs is both more performant and more manageable. **Cilium** (<https://github.com/cilium/cilium>) is focused on routing and filtering traffic using eBPF
- **Monitoring connections:** Creating an up-to-date map of TCP connections between containers is possible by attaching a BPF program, kprobes, that tracks socket-level events. **WdaveScope** (<https://github.com/weaveworks/scope>) utilizes this capability by running an agent on each node that collects this information, and then sends it to a server that provides a visual representation through a slick UI.
- **Restricting syscalls:** The Linux kernel provides more than 300 system calls. In a security-sensitive container environment, it is highly desirable. The original **seccomp** (<https://en.wikipedia.org/wiki/Seccomp>) facility was pretty coarse-grained. In Linux 3.5, seccomp was extended to support BPF for advanced custom filters.

- **Raw performance:** eBPF provides significant performance benefits, and projects like **Calico** (<https://www.projectcalico.org/>) take advantage and implement a faster data plane that uses less resources.

Custom hardware and devices

Kubernetes manages nodes, networking, and storage at a relatively high-level. However, there are many benefits for integrating specific hardware at a fine-grained level; for example, GPUs, high-performance network cards, FPGAs, InfiniBand adapters, and other compute, networking, and storage resources. This is where the **device plugin** (<https://github.com/kubernetes/community/blob/master/contributors/design-proposals/resource-management/device-plugin.md>) framework comes in. It is still in Beta (since Kubernetes 1.10) and there is ongoing innovation in this area. For example, monitoring device plugin resources is also in beta since Kubernetes 1.15. It is very interesting to see what devices will be harnessed with Kubernetes. The framework itself is following modern Kubernetes extensibility practices by utilizing gRPC.

Service mesh

The service mesh is arguably the most important trend over the last couple of years. We covered service meshes in depth in *Chapter 14, Utilizing Service Meshes*. The adoption is impressive, and I predict that most Kubernetes distributions will provide a default service mesh and allow easy integration with other service meshes. The benefits that service meshes provide are just too valuable, and it makes sense to provide a default platform that includes Kubernetes with an integrated service mesh. That said, Kubernetes itself will not absorb some service mesh and expose it through its API. This goes against the grain of keeping the core of Kubernetes small.

Google **Anthos** (<https://cloud.google.com/anthos/>) is a good example where Kubernetes + Knative + Istio are combined to provide a unified platform that provides an opinionated best-practices bundle. It would take an organization a lot of time and resources to build on top of vanilla Kubernetes.

Another push in this direction is the **sidecar container** KEP (<https://github.com/kubernetes/enhancements/blob/master/keps/sig-apps/sidecarcontainers.md>).

The sidecar container pattern has been a staple of Kubernetes from the get-go. After all, pods can contain multiple containers. However, there was no notion of a main container and a sidecar container. All containers in the pod has the same status. Most service meshes use sidecar containers to intercept traffic and perform their jobs. Formalizing sidecar containers will help those efforts and push service meshes even further.

It's not clear, at this stage, if Kubernetes and the service mesh will be hidden behind a simpler abstraction on most platforms or if they will be front and center.

Serverless computing

Serverless computing is another trend that is here to stay. We discussed it at length in *Chapter 12, Serverless Computing on Kubernetes*. Kubernetes and serverless can be combined on multiple levels. Kubernetes can utilize serverless cloud solutions like AWS Fargate and AKS **Azure Container Instances (ACI)** to save the cluster administrator from managing nodes. This approach also caters to integrating lightweight VMs transparently with Kubernetes, since the cloud platforms don't use naked Linux containers for their container-as-a-service platforms.

Another avenue is to reverse the roles and expose containers as a service powered by Kubernetes under the covers. This is exactly what Google **Cloud Run** (<https://cloud.google.com/run/>) is doing. The lines blur here as there are multiple products from Google to manage containers and/or Kubernetes ranging from just GKE, through Anthos GKE (bring your own cluster to a GKE environment for your private data center), Anthos (managed Kubernetes + service mesh), and Anthos cloud run.

Finally, there are function as a service and scale to zero projects running inside your Kubernetes cluster. I believe Knative will become the leader here, as it is already used by many frameworks and it is deployed heavily through various Google products.

Kubernetes on the Edge

Kubernetes is the poster boy of cloud native computing, but with the **Internet of Things (IoT)** revolution, there is more and more need to perform computation at the edge of the network. Sending all data to the backend for processing suffers from several drawbacks:

- Latency
- Need for enough bandwidth
- Cost

With edge locations collecting a lot of data via sensors, video cameras, and so on, the amount of edge data grows and it makes more sense to perform more and more sophisticated processing at the edge. Kubernetes grew out of Google's Borg, which was definitely not designed to run at the edge of the network. However, Kubernetes' design proved to be flexible enough to accommodate it.

I expect that we will see more and more Kubernetes deployments at the edge of the network, which will lead to very interesting systems that are composed of many Kubernetes clusters that will need to be managed centrally.

KubeEdge (<https://kubedge.io/en/>) is an open source framework that is built on top of Kubernetes and Mosquito – an open source implementation of MQTT message broker – to provide a foundation for networking, application deployment, and metadata synchronization between the cloud and the edge.

Native CI/CD

For developers, one of the most important questions is the construction of a CI/CD pipeline. There are many options available, and choosing between them can be difficult. The **CD foundation** (<https://cd.foundation/>) is an open source foundation that was formed to standardize concepts like pipelines and workflows, as well as define industry specifications that will allow different tools and communities to interoperate better. The current projects are:

- Jenkins (<https://www.jenkins.io/>)
- Jenkins X (<https://jenkins-x.io/>)
- Tekton (<https://github.com/tektoncd/pipeline>)
- Spinnaker (<https://www.spinnaker.io/>)

There is also an incubating project: **Screwdriver.cd** (<https://screwdriver.cd/>).

One of my favorite native CD projects, **Argo CD** (<https://github.com/argoproj/argo-cd>), is not part of the CD foundation at the moment. I took action and opened a GitHub issue (<https://github.com/argoproj/argo-cd/issues/3265>) asking to submit argo-cd to the CDF.

Another project to watch is **CNB – Cloud Native Buildpacks** (<https://buildpacks.io/>). The project takes a source and creates OCI (think Docker) images. It is important for **Function as a Service (FaaS)** frameworks and native in-cluster CI. It is also a CNCF sandbox project.

Operators

The Operator pattern emerged in 2016 from CoreOS (acquired by RedHat, acquired by IBM) and gained a lot of success in the community. An Operator is a combination of custom resources and a controller used to manage an application. At my current job, I write Operators to manage various aspects of infrastructure and it is a joy.

It is already the established way to distribute non-trivial applications to Kubernetes clusters. Check out <https://operatorhub.io/> for a huge list of existing operators. I expect this trend to continue.

Summary

In this chapter, we looked at the future of Kubernetes, and it looks great! The technical foundation, the community, the broad support, and the momentum are all very impressive. Kubernetes is still young, but the pace of innovation and stabilization is very encouraging. The modularization and extensibility principles of Kubernetes let it become the universal foundation for modern cloud-native applications.

At this point, you should have a clear idea of where Kubernetes is right now and where it's going from here. You should be confident that Kubernetes is not just here to stay, but that it will be the leading container orchestration platform for many years to come. It will be able to integrate with any major offering and environment you can possibly imagine, from planet-scale public cloud platforms, private clouds, data centers, and edge locations and all the way down to your development laptop and Raspberry Pi.

That's it! This is the end of this book.

Now, it's up to you to use what you've learned and build amazing things with Kubernetes!

References

- Kubernetes on GitHub: <https://github.com/kubernetes/kubernetes>
- CNCF: <https://cncf.io>
- CD foundation: <https://cd.foundation/>
- FireCracker: <https://firecracker-microvm.github.io/>
- gVisor: <https://github.com/google/gvisor-containerd-shim>
- Cilium: <https://github.com/cilium/cilium>
- Calico: <https://www.projectcalico.org/>
- Google Anthos: <https://cloud.google.com/anthos/>
- Google Cloud Run: <https://cloud.google.com/run/>
- KubeEdge: <https://kubedge.io/en/>
- OperatorHub: <https://operatorhub.io/>

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Python Automation Cookbook - Second Edition

Jaime Buelta

ISBN: 978-1-80020-708-0

- Learn data wrangling with Python and Pandas for your data science and AI projects
 - Automate tasks such as text classification, email filtering, and web scraping with Python
 - Use Matplotlib to generate a variety of stunning graphs, charts, and maps
- Automate a range of report generation tasks, from sending SMS and email campaigns to creating templates, adding images in Word, and even encrypting PDFs

- Master web scraping and web crawling of popular file formats and directories with tools like Beautiful Soup
- Build cool projects such as a Telegram bot for your marketing campaign, a reader from a news RSS feed, and a machine learning model to classify emails to the correct department based on their content
- Create fire-and-forget automation tasks by writing cron jobs, log files, and regexes with Python scripting



IoT and Edge Computing for Architects - Second Edition

Perry Lea

ISBN: 978-1-83921-480-6

- Understand the role and scope of architecting a successful IoT deployment
- Scan the landscape of IoT technologies, from sensors to the cloud and more
- See the trade-offs in choices of protocols and communications in IoT deployments
- Become familiar with the terminology needed to work in the IoT space
- Broaden your skills in the multiple engineering domains necessary for the IoT architect
- Implement best practices to ensure reliability, scalability, and security in your IoT infrastructure

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

A

access control webhook

- authentication webhook, using 564-566
- authorization webhook, using 566-568
- custom metrics, providing for horizontal pod autoscaling 570
- employing 563
- Kubernetes, extending with custom storage 571

adapter pattern 13

admission control webhook

- using 568

advanced scheduling 150

- anti-affinity 153
- node affinity 153
- node selector 150
- pod affinity 153
- taints 151, 152
- tolerations 151, 152

alerting 448

AlertManager

- reference link 396

alerts

- versus dashboards 480

Alibaba ACK

- URL 579

Alibaba Cloud 57

Alibaba container service for Kubernetes (ACK) 57

Amazon EBS 198, 199

Amazon EFS 199, 200

Amazon EKS

- URL 579

Amazon Virtual Private Cloud (Amazon VPC) 55

Amazon Web Services (AWS) 4

ambassador pattern 13

annotations 8

Anthos

- reference link 582

anti-affinity 153

API builder alpha

- reference link 550

APIs

- deprecating 87

API server

- accessing 105
- admission control plugins, using 110, 111
- requests, authorizing 108, 109
- users, authenticating 106, 107

AppArmor

- pod security 114, 115
- profiles, writing 115, 116
- requisites 114
- used, for protecting cluster 114

AppDash

- reference link 473

Application Container Image (ACI) 27

application error reporting 447

Argo CD

- URL 584

Attribute-Based Access Control (ABAC) 109

authentication webhook

- using 564-566

authorization webhook

- using 566-568

AutoContainerSource

- reference link 421

autoscaling

- used, for performing rolling updates 254-257

AWS App Mesh 488

URL 488

AWS EKS 55, 410, 411**AWS Fargate 55****AWS PrivateLink 55****Azure 56****Azure AKS 409, 410****Azure Container**

Instances (ACI) 409, 410, 583

Azure data disk 202, 203**Azure file storage 203****Azure Kubernetes Service (AKS) 4, 56, 272**

benefits 56

B**bane tool**

reference link 115

bare-metal 579**bare-metal cluster**

building, with KRIB 60

building, with Kubespray 60

building, with RKE 61

creating, considerations 59

creating, from scratch 58

creating, process 59

use cases 58

virtual private cloud infrastructure, using 60

blue-green deployments 84, 85**Bookinfo 492**

installing 495-499

Bootkube 61**bridge plugin**

reviewing 359, 361

bridges 325**broker 422****buildpacks**

URL 439

build.sh script

exploring 232

built-in objects

embedding 311

Buoyant

URL 487

C**cAdvisor 454****Calico**

reference link 582

Calico project 337**canary deployments 85, 86****Candy 543****capacity planning 77, 78****Cassandra 229**

configuration file 238, 239

connecting 238

custom seed provider 239

headless service, creating 241

Cassandra cluster

executing, in Kubernetes 228

reference link 228

Cassandra Docker image 230, 231

build.sh script, exploring 231

run.sh script, exploring 233-238

cattle versus pets 5**cbr0 326****CD foundation**

reference link 584

centralized logging 450

cluster-level central logging 452

remote central logging 452

sensitive log information, dealing with 453

Ceph

connecting, with CephFS 210

connecting, with RBD 208-210

using 208

CephFS

used, for connecting Ceph 210

Ceph volumes

in Kubernetes 204

Certified Kubernetes Administrator (CKA)

reference link 577

Certified Kubernetes Application Developer (CKAD)

reference link 577

channels 422**chart dependencies 313, 314**

managing 304

managing, with requirements.yaml 305

special fields, utilizing in requirements.yaml

306, 307

charts

- creating 302
- metadata files 304
- templates and values, using 307
- testing 309, 310
- troubleshooting 309, 310

chart scope 313, 314

chart values 313, 314

Chart.yaml file 303

- appVersion field 303
- charts, deprecating 304
- version field 303

CI/CD pipeline 29

- deploying, for Kubernetes 30

Cillium

- reference link 581

Citadel 491

- workflow, in Kubernetes 491, 492

Classless Inter-Domain Routing (CIDR) 325

client IP addresses

- preservation, specifying 345
- preserving 345

cloud

- clusters 52

cloud controller manager

- used, for extending Kubernetes 540

Cloud Controller Manager (CCM) 540

- URL 540

Cloud Native Buildpacks

- URL 584

Cloud Native Computing Foundation (CNCF)

- certification 577
- community and education 578
- project curation 576, 577
- significance 576
- training 578
- URL 576

cloud-provider interface 52

cloud providers 56

- Chinese Alibaba Cloud 57
- Huawei 57
- IBM Kubernetes Service 57
- Oracle Container Service 58
- Tencent 57

Cloud Run

- reference link 583

cluster

- exploring, with Kubernetes dashboard 457

cluster autoscaler 408

- installing 78, 80

cluster autoscaler (CA) 78

cluster capacity

- container-native solutions, considering 272
- elastic cloud resources, benefiting 270
- managing 268
- multiple node configurations, using 270
- node types, selecting 268
- off cost and response time, trading 269, 270
- scalable storage solution, selecting 269
- selecting 268

cluster federation 366, 367

- history, on Kubernetes 366

cluster federation, use cases 368

- capacity overflow 368
- Geo-distributing high availability 369
- sensitive workloads 368, 369
- vendor lock-in, avoiding 369

cluster-level central logging 452

cluster management commands 136

clusters 6

CNCF landscape

- reference link 576

CNCFs CloudEvents specification

- reference link 420

CNI plugin 329-331

- skeleton, building 356-359
- writing 352

compute resource quota 258

condition field 306

ConfigMap

- consuming, as environment variable 224, 225
- creating 224

containerd 26

container-native solutions

- considering 272

container networking

interface (CNI) 327, 539

- container runtime 328
- third-party plugin 328

container orchestration 3

container runtime 328

container runtime interface (CRI) 23-25

containers

- benefits 3, 4
- coupled connectivity, with data stores 323
- coupled connectivity, with queues 323
- in cloud 4
- interacting 322
- registration service 322
- self-registration 322

ContainerSource

- reference link 421

container storage

- interface (CSI) 10, 216, 217, 539, 571

continuous integration and deployment 28

Contiv 332

Contiv net plugin

- capabilities 332

CoreV1API group

- dissecting 532, 533

CRD components

- admission plugin 80
- recommender 80
- updater 80

CRI-O 27

cron jobs

- scheduling 164, 166

cross-cluster scheduling 375

curl 46

customization commands 136

custom container runtimes

- used, for extending Kubernetes 542

custom devices 582

custom hardware 582

custom metrics

- providing, for horizontal pod autoscaling 570

custom metrics API server

- reference link 570

custom resources 542, 543

- custom printer columns, adding 548, 549
- finalizing 548
- integrating 545, 546
- unknown fields, dealing with 546, 547

custom resources definitions (CRD) 490, 543

- developing 543-545

custom scheduler

- pod, assigning 557
- preparing 556

custom storage

- used, for extending Kubernetes 571

D

DaemonSet

- using, for redundant persistent storage 226

DaemonSet pods

- sharing 171

Dapper

- reference link 473

dashboards

- versus alerts 480

data

- migrating 86

data-contract changes

- managing 86

deployment

- updating 143

deployment commands 136

device plugins 540

Digital Ocean

- URL 579

Digital Rebar Provision (DRP) 60

directed acyclic graph (DAG) 446

direct logging approach 450

directory structure

- configuring 157, 158

distributed data-intensive apps 222

distributed hash table (DHT) 205, 229

distributed system design patterns 12

- adapter pattern 13
- ambassador pattern 13
- multi-node patterns 14
- sidecar pattern 13

distributed tracing 446

- used, for detecting performance 482
- used, for detecting root cause 482
- with Jaeger 470, 471

DNS Provider 387

DNS records

- versus shared environment variables 223

Docker 25, 26

Docker networking

- versus Kubernetes networking
- model 320, 321

Docker networking model

- versus Kubernetes networking model 321

Domain Name System (DNS) 370

durable node storage

with local volumes 180, 181

dynamic host path provisioner

reference link 293

E

eksctl

URL 55

Elastic Block Store (EBS) 198

elastic cloud resources

benefiting 270

cloud quotas 271

instance autoscaling 270

regions, managing 271

Elastic container instances (ECIs) 57

Elastic Container Service (ECS) 53

Elastic File System (EFS) 199

Elastic Kubernetes Service (EKS) 4, 53, 272

Elasticsearch, Kibana, and Fluentd (EFK) 515

emptyDir

using, for intra-pod communication 176-178

enterprise storage

integrating, into Kubernetes 212

Envoy 487, 490

URL 487

error reports

versus logs 481

versus metrics 481

etcd 19

etcd3 276

gRPC, using instead of REST 276

leases, using instead of TTLs 276

optimizations 277

state storage 276

watch implementation 276

etcd cluster

about 69

creating 72, 73

verifying 73

etcd operator

installing 70-72

etcd-operator

reference link 541

event consumer 421

event consumer, types

Addressable consumer 421

Callable consumer 421

event delivery, modes

fan-out delivery 423

simple delivery 422

event registry 422

event source 421

event types 422

extended Berkeley Packet Filter (eBPF) 581

applications 581

external data stores

accessing, via DNS 223

accessing, via environment variables 223

External DNS Controller 387

external load balancer 343, 346

client IP addresses, preserving 345

configuring 344

configuring, via kubectl command 344

configuring, via service configuration file 344

IP addresses, finding 344

external service

exposing 148, 149

separating 144

F

fan-out delivery 423

Fargate 410, 411

limitations 412

reference link 410

federation API server 372

federation controller manager 372

Fiber Channel (FC) 189

FireCracker

reference link 581

firecracker-containerd

reference link 581

Fission 429, 430

experimenting 432, 433

URL 429

workflows 430-432

Flannel 335, 336

backends 336

FlexVolume approach

advantages 216

out-of-tree volume plugins, using 215, 216

Flocker

as clustered container data volume manager 211, 212

Fluentbit

URL 454

Fluentd

URL 453

using, for log collection 453

Frakti 28

Function as a Service (FaaS) 584

characteristics 408

executing, on serverless computing 407

functions

using 308

G

Galley 492

Gardener

extending 397-401

Gardener architecture 394

clusters, monitoring 395, 396

clusters, networking 395

cluster state, managing 394

control plane, managing 395

gardenctl CLI 396

infrastructure, preparing 395

machine controller manager, using 395

Gardener project

conceptual model 393, 394

terminology 392

URL 392

Gardener ring 401

GCE persistent disk 201, 202

generic commands 136

Giant Swarm

reference link 580

gibibyte (GiB) 183

GlusterFS

endpoints, creating 205

Pods, creating 207

using 205

GlusterFS Kubernetes service

adding 206

GlusterFS volumes

in Kubernetes 204

Google Anthos for GKE

reference link 579

Google Cloud Platform (GCP) 4, 53

Google Cloud Run 412

reference link 580

Google GKE

URL 579

Google Kubernetes Engine (GKE) 53, 272

Google Kubernetes Engine (GKE), on Coursera

reference link 578

Grafana

reference link 396

URL 468

Grafana Loki

reference link 470

gvisor-containerd-shim

reference link 581

H

HAProxy

executing, in Kubernetes cluster 350

NodePort, utilizing 349

using, in load balancer provider 349

Heapster 67, 454

Helm 283

chart, creating 302

chart, customizing 296, 297

charts, finding 287, 288

charts, managing 301

installation link 70

installation options 298

installation status, checking 292-296

installing 286

package, installing on Kubernetes

cluster 290

release, deleting 299, 300

release, rolling back 298, 299

release, upgrading 298, 299

repositories, adding 288-290

repositories, working with 300, 301

use cases 284

using 285

Helm 2

Tiller server, installing 286

used, for installing riff 439-442

Helm 2 architecture 284

Helm 2 components 284

- Helm client 285
- Tiller server 284
- Helm 3 285**
- Helm client**
 - installing 286
- hierarchical cluster structures**
 - with kustomization 156
- high availability 77, 78**
- high availability, best practices**
 - about 66
 - data, protecting 73
 - etcd cluster 69
 - Kubernetes cluster, creating 67, 68
 - Kubernetes cluster state, protecting 69
 - leader election, executing with
 - Kubernetes 74, 75
 - nodes performance, creating 68, 69
 - staging environment, creating 75
 - testing 76, 77
- high availability, concepts**
 - about 64
 - hot swapping 64
 - idempotency 66
 - leader election 65
 - redundancy 64
 - self-healing 66
 - smart load balancing 65
- High-Availability Proxy (HAProxy) 349**
 - reference link 349
- Higher-Order Behavior**
 - employing 387
- highly available (HA) clusters 44**
- horizontal pod autoscaler (HPA) 78, 248**
 - autoscaling, with Kubectl 251-253
 - custom metrics 251
 - deploying 248-250
- host cluster**
 - configuring 379, 381
 - registering, with Kubernetes federation 381
- HostPath**
 - using, for intra-node communication 178-180
- hot swapping 64**
- Httpie**
 - installation link 46
- URL 529
 - used, for filtering output 529, 530
 - using 529
- Huawei 57**
- Hue**
 - advanced science 173
 - utilizing, for education 173
 - utilizing, in enterprise 172
- Hue components**
 - about 131
 - authorizer 132
 - external service 132
 - generic actuator 132
 - generic sensor 132
 - identity 132
 - user graph 131
 - user learner 133
 - user profile 131
- Hue microservices**
 - about 133
 - data stores 134
 - plugins 133
 - queue-based interactions 134, 135
 - serverless functions 134
 - stateless microservices 134
- Hue platform**
 - designing 129
 - evolving, with Kubernetes 172
 - identity 130, 131
 - managing, with Kubernetes 167
 - notifications 130
 - privacy 130, 131
 - scope, defining 130
 - security 130, 131
 - smart reminders 130
- Hue-reminders service**
 - creating 146, 147
- Hue workflows**
 - automatic workflows 135
 - budget-aware workflows 135
 - human workflows 135
 - planning 135
- Hyper Containers 28**
 - Frakti 28
 - Stackube 28

I

IBM Cloud Kubernetes service

URL 579

IBM Kubernetes Service 57

idempotency 66

Ingress

Ingress DNS controller 387

Ingress DNS Record 387

init containers

employing, for orderly pod bring-up 169

inside-the-cluster-network components 167

internal service

deploying 145, 146

separating 144

Internet of Things (IoT) 583

intra-node communication

with HostPath 178-180

intra-pod communication

with emptyDir 176-178

IP addresses 324

IP Address Management (IPAM) 329

Istio 488

distributed tracing 519-522

incorporating, into Kubernetes cluster 489

installing 493-495

logs 513-516

metrics 516-519

minikube cluster, preparing 492

monitoring and observability 512

policies 509-511

security 502, 503

traffic management 499-502

URL 488

Istio architecture 489, 490

Citadel 491

Envoy 490

Galley 492

Mixer 491

Pilot 490

Istio authentication 504

origin authentication 505

transport authentication 504

Istio authorization 505-508

istio-certs 504

Istio identity 503

Istio PKI 504

J

Jaeger 472

distributed tracing 470, 471

installing 475-478

URL 472

Jaeger agent 474

Jaeger architecture 473, 474

Jaeger client 474

Jaeger collector 474

Jaeger Query 474

Java Management Extensions (JMX) 237

Jenkins

URL 584

Jenkins X

URL 584

jobs

cleaning up 164

executing, in parallelism 163, 164

launching 162

jq

URL 529

used, for filtering output 529, 530

using 529

jsonpatch

URL 385

JSON Web Tokens (JWTs) 505

K

k3d 48

installing 48

used, for creating cluster 49-51

used, for creating multi-node cluster 47

versus Minikube 51, 52

k3s 48

k9s tools

reference link 559

Keepalived Virtual IP (Keepalived VIP) 351

Kiali

used, for visualizing service mesh 522

Kind

about 42

echo service, deploying with 46

installing 42

used, for creating multi-node cluster 42-45

versus k3d 51

versus Minikube 51

Knative 413-439

installing 424, 425

Knative, components

Knative Eventing 413

Knative Serving 413

Knative Configuration object 417, 418**Knative Eventing 413, 420**

architecture 422, 423

Knative Eventing terminology

broker 422

channel 422

defining 420

event consumer 421

event registry 422

event source 421

event types 422

subscriptions 422

trigger 422

Knative, installing

reference link 424

Knative Revision object 420**Knative Route object 416****Knative service**

deploying 426

invoking 426, 427

scale-to-zero option, checking in 427, 428

Knative Service object 414, 416**Knative Serving 413****Krew**

reference link 560

used, for managing Kubectl plugins 560

krew-plugin-template

reference link 562

KRIB

reference link 61

used, for building bare-metal cluster 60

kubeadm

reference link 59

kubebuilder

reference link 541, 549

Kube controller manager 19**Kubectl 32**

executing, with Python

subprocesses 536-538

reference link 33

used, for autoscaling HPA 251, 253

Kubectl commands

overriding 562

Kubectl effectively

using 136

Kubectl plugins 562

creating 561, 562

implementing 559

managing, with Krew 560

namespace, for Krew plugins 563

naming 562

shebangs 562

writing 559

Kubectl programmatically

invoking 536

kubectl resource, configuration files

about 137

ApiVersion 138

container spec 138, 139

kind 138

metadata 138

spec 138

KubeEdge

reference link 580, 584

KubeFed control plane 372

federation API server 372

federation controller manager 372

kubefedctl

installing 377-379

Kubeless 434

implementing with 435, 437

working, with serverless framework 438

Kubeless architecture 434

Kubeless function 434

Kubeless runtime 434

Kubeless triggers 435

Kubeless function 434**Kubeless runtime 434****Kubeless triggers 435****Kubeless UI**

using 437, 438

kubelet 22**Kubemark cluster**

comparing, to real-world cluster 281

reference link 281

setting up 281

Kubemark tool 281**Kubenet 324, 326**

MTU, setting 327

requisites 326, 327

kubens tool

reference link 128

kube-prometheus

reference link 461

Kubernetes 576

API objects, serializing with protocol
buffers 276

API responsiveness, measuring 277, 278

capabilities 2

centralized logging 450

Ceph volumes in 204

CI/CD pipeline, deploying for 30

cluster federation, history 366

component logs 449

configuration and deployment challenges 100

connecting 238

container logs 448, 449

cultural challenges 102

end-to-end pod startup time, measuring 279

enterprise storage, integrating 212

etcd3 276

extending, with cloud controller manager 540

extending, with controller pattern 541

extending, with custom container
runtimes 542

extending, with custom storage 571

extending, with operator pattern 541

extending, with plugins 539

features 54

Fluentd, using for log collection 453

GlusterFS in 204

image challenges 99, 100

limits 273, 274

logging with 448

monitoring, with metrics server 455, 456

network challenges 97, 98

network policies, managing 118

node challenges 96, 97

on EC2 54

organizational challenges 102

overview 103

performance and scalability, improving 274

performance and scalability, measuring 277

Platform as a Service (PaaS) 580

pod and container challenges 101

pod lifecycle event generator (PLEG) 274, 275

Pods security 112

process challenges 102

reads, caching in API server 274

scheduling, extension 542

secrets, storing in 122

security challenges 96

service accounts 103, 104

service accounts, managing 105

SLOs 277

stateful applications 221

stateless applications 221

state, managing in 222

state, managing outside 222, 223

testing, at scale 280

upcoming trends 580

used, for collecting metrics 454, 455

used, for evolving Hue platform 172

used, for managing Hue platform 167

using, to build Hue platform 136

Kubernetes API 14

accessing, via Python client 531, 532

CoreV1API group, dissecting 532, 533

exploring 526, 527

exploring, with Postman 528

extending 538, 539

objects, creating 534, 535

objects, listing 534

objects, watching 535

Python subprocesses, used for executing
Kubectrl 536-538

resource categories 15

used, for creating pod 530, 531

working with 525

Kubernetes API server

aggregating 549, 550

Kubernetes architecture 6, 12

distributed system design patterns 12

Kubernetes Certified Service Provider (KCSP)

reference link 577

Kubernetes cli-runtime

reference link 562

Kubernetes cluster

APIs, deprecating 87

availability requisites 88

bare-metal cluster 58

best effort 88

blue-green deployments 84, 85

- canary deployments 85, 86
- cost 88
- creating 67, 68
- data consistency 93
- data-contract changes, managing 86
- data, migrating 86
- design trade-offs 88
- HAProxy, executing in 350
- Helm package, installing on 290
- Istio, incorporating into 489
- live updates 80
- maintenance windows 89
- multi-node cluster 42
- overview 31
- performance 88, 93
- quick recovery 90
- rolling updates 81, 82
- single-node cluster 32
- Site reliability engineering (SRE) 92
- zero downtime 90

Kubernetes cluster federation

- creating 379
- Higher-Order Behavior, employing 387
- managing 377
- overrides, using 385
- placement field, using to control 385, 386
- propagation failures, debugging 387

Kubernetes, components

- master components 18
- node components 21

Kubernetes, concepts 5

- annotations 8
- cluster 6
- labels 8
- label selectors 9
- master 7
- names 11
- namespaces 11, 12
- nodes 6
- Pods 7
- replica sets 10
- replication controllers 10
- secrets 11
- services 9
- StatefulSet 10, 11
- volume 10

Kubernetes contrib

- reference link 351

Kubernetes dashboard

- used, for exploring cluster 457

Kubernetes extensions patterns 539

Kubernetes extensions points 539

Kubernetes FaaS frameworks 428, 429

- Fission 429, 430
- Knative 439
- Kubeless 434
- riff 439

Kubernetes federation

- API types, working with 381, 382
- auto-scaling 376, 377
- basic concepts, defining 370
- basics, learning 370
- building blocks 370
- data access 376
- features 372
- namespace 384
- overview 373
- resources 382, 383
- resources status, checking 384
- unit of work 374
- used, for registering host cluster 381

Kubernetes federation, elements

- policy 371
- scheduling 371
- status 371

Kubernetes GitHub repository

- reference link 576

Kubernetes incubator project

- reference link 182

Kubernetes ingress 324

Kubernetes networking model 318

- external access 319
- inter-pod communication (pod to pod) 318
- intra-pod communication (container to container) 318
- pod to service communication 319
- versus Docker networking 320, 321
- versus Docker networking model 321, 322

Kubernetes networking solutions 332

- bridging, on bare metal clusters 332
- Calico project 337
- Contiv 332, 333
- Flannel 335, 336
- Open vSwitch (OVS) 333-335

- Romana 337-339
- Weave net 340
- Kubernetes network plugin 324**
 - bridges 325
 - CIDRs 325
 - Container Networking Interface (CNI) 327
 - IP addresses 324
 - Kubenet 326
 - Linux networking 324
 - maximum transmission unit (MTU) 326
 - netmasks 325
 - network namespaces 325
 - pod networking 326
 - ports 324
 - routing 325
 - subnets 325
 - Virtual Ethernet (veth) devices 325
- Kubernetes network policy 341**
 - CNI plugin 341
 - configuring 341
 - design 340
 - implementing 342
 - using 340
- Kubernetes on Raspberry PI**
 - reference link 580
- Kubernetes plugins**
 - custom scheduler, writing 552
 - writing 552
- Kubernetes runtimes 22**
 - container runtime interface (CRI) 23, 25
 - CRI-O 27
 - Docker 25, 26
 - Hyper Containers 28
 - rkt 27
- Kubernetes scheduler**
 - design 552-554
- Kubernetes services**
 - accessing, locally through proxy 46, 47
- kube scheduler 21**
- Kubespary 67**
 - used, for building bare-metal cluster 60
- kube-state-metrics**
 - reference link 396
- Kuma 488**
 - URL 488
- kustomization**
 - applying 158, 159
 - patching 160
 - staging namespace, kustomizing 160, 161
 - using, for hierarchical cluster structures 156
- kustomize**
 - basics 156
 - URL 156

L

- labels 8**
- label selectors 9**
- leader election 65**
- limit ranges**
 - using, for default compute quotas 267, 268
- Linen CNI plugin**
 - reference link 333
- Linkerd 487**
- Linkerd 2 487**
 - URL 487
- Linux networking 324**
- liveness probe**
 - using, to ensure containers 167
 - using, to manage dependencies 168
- load balancer provider**
 - with HAProxy 349
- load balancing options 342, 343**
 - external load balancer 343
 - ingress 347
- local volumes**
 - using, for durable node storage 180, 181
- location affinity 374**
- location affinity, requirements**
 - loosely coupled 375
 - preferentially coupled 375
 - strictly coupled 374
 - strictly decoupled 375
 - uniformly spread 375
- log aggregation 445, 448, 450**
- log collection strategy**
 - direct logging approach 450
 - node agent approach 451
 - selecting 450
 - sidecar container 451
- log format 445**
- logging 444**
 - with Kubernetes 448
- logs**

versus error reports 481

versus metrics 481

logs, key attributes

log aggregation 445

log format 445

log storage 445

log storage 445

long-running microservices

deploying, in pods 139

long-running processes

deploying, with deployments 142

long-running services

characteristics 407

executing, on serverless computing 406

loopback plugin 352-356

reference link 353

M

macOS

single-node cluster, creating 34, 35

Maesh

URL 488

managed Kubernetes platforms 579

master 7

master components

API server 449

controller manager 449

scheduler 449

master components, Kubernetes

API server 18

cloud controller managers 19, 20

DNS 21

etcd 19

Kube controller manager 19

kube scheduler 21

maximum transmission unit (MTU) 326

MetalLB 351

reference link 351

metrics 445

collecting, with Kubernetes 454, 455

types 516

versus error reports 481

versus logs 481

Microsoft AKS

URL 579

Minikube 33

reference link 33

used, for creating single-node cluster 32

versus k3d 52

versus KinD 51

minikube cluster

preparing, for Istio 492

minions 6

misc commands 136

Mixer 491

multi-cluster Ingress DNS

utilizing 387

multi-cluster scheduling

utilizing 389-392

multi-cluster Service DNS

utilizing 388, 389

multi-container pod challenges 101

multi-node cluster

creating, with k3d 47-51

creating, with KinD 42-45

multi-node patterns 14

multiple node configurations

using 270

multi-user cluster

executing 125

namespace pitfalls, avoiding 127, 128

namespace, using for safe

multi-tenancy 126, 127

use cases 126

N

names 11

namespaces 11, 12

using, to limit access 154, 156

namespace-specific context

using 262

native CI/CD 584

Network Address Translation (NAT) 318

networking 581

network namespaces 325

network policies

cross-namespace policies 122

defining 119-121

egress network policy, limiting to external

networks 121

managing 118

networking solution, selecting 119

- secrets, using 122
- node affinity 153**
 - advantages 153
- node agent approach 451**
- node components, Kubernetes**
 - kubelet 22
 - proxy 21
- node-exporter**
 - reference link 396
- NodePort**
 - utilizing 349
- node-problem-detector**
 - reference link 479
- nodes 6**
- node selector 150**
- node types**
 - selecting 268, 269
- non-cluster components**
 - inside-the-cluster-network components 167
 - mixing 166
 - outside-the-cluster-network components 166
- Nuage networks VCS 335**

O

- object count quota 260**
- objects**
 - creating 534, 535
 - Kubectl programmatically, invoking 536
 - listing 534
 - watching 535
- observability 444**
 - alerting 448
 - application error reporting 447
 - dashboards 447
 - distributed tracing 446
 - logging 444
 - metrics 445
 - visualization 447
- octant tools**
 - reference link 559
- off cost and response time**
 - trading 269, 270
- OpenAPI 526**
- OpenAPI V3**
 - reference link 544
- Open Container Initiative (OCI) 26**

- Open Service Broker API**
 - reference link 550
- OpenShift**
 - reference link 580
- OpenStack**
 - reference link 579
- OpenTracing 471**
 - URL 471
- OpenTracing, concepts 472**
 - Span 472
 - Trace 472
- Open Virtualization Network (OVN) 333**
- Open Virtual Networking (OVN) 328**
 - reference link 333
- Open vSwitch (OVS) 328, 333**
 - key features 334, 335
- operator framework**
 - reference link 541
- operator pattern 584**
 - used, for extending Kubernetes 541
- Oracle Cloud**
 - URL 579
- Oracle Container Service 58**
- origin authentication 505**
- out-of-tree volume plugins**
 - using, with FlexVolume 215, 216
- outside-the-cluster-network components 166**
- overrides**
 - using 385

P

- performance**
 - detecting, with distributed tracing 482
- Persistent Volume Claim (PVC) 190**
- persistent volume claims**
 - applying 226
- persistent volumes**
 - access mode 183
 - capacity 183
 - claims, creating 185, 187
 - claims, mounting 188
 - creating 182
 - dynamically, provisioning 182
 - externally, provisioning 182
 - mount options 185
 - overview 175, 176

- provisioning 181
- raw block volumes 189, 190
- reclaim policy 184
- statically, provisioning 182
- storage class 184, 191
- storage classes 192
- storage, demonstrating end to end 192-198
- volume mode 183
- volume type 185
- personally identifiable information (PII) 453**
- physical machines 3**
- Pilot 490**
- pipelines**
 - using 308
- placement field**
 - using, to control Kubernetes cluster federation 385, 386
- Platform 9 PMK**
 - reference link 580
- Platform as a Service (PaaS) 2**
- plugins**
 - used, for extending Kubernetes 539
- pod affinity 153**
- pod lifecycle event generator (PLEG) 275**
- pod networking 326**
- pod readiness 170**
- pods 7**
 - assigning, to custom scheduler 557
 - creating 139, 140
 - creating, via Kubernetes API 530, 531
 - decorating, with labels 141
 - endpoints 322
 - interacting 322
 - long-running microservices, deploying 139
 - scheduling 555
 - verifying, with correct scheduler 558
- pod security**
 - with AppArmor 114, 115
- pod security policies (PSPs) 116, 117**
 - authorizing, via RBAC 117, 118
- pods security 112**
 - cluster, protecting with AppArmor 114
 - ImagePullSecrets 112
 - private image repository, using 112
 - security context, specifying 113
- ports 324**
- Postman**
 - output, filtering with httpie and jq 529, 530
 - URL 528
 - using, to explore Kubernetes API 528
- priority classes 261**
- private clouds 579**
- Prometheus 458**
 - alertmanager 466-468
 - custom metrics, incorporating 466
 - features 458
 - installing 460, 461
 - interacting 462
 - kube-state-metrics, incorporating 462, 464
 - Loki, considering 470
 - metrics, visualizing with Grafana 468, 470
 - node exporter, utilizing 464
 - reference link 395
 - URL 458
- Prometheus operator**
 - reference link 460
- propagation failures**
 - debugging 387
- protected health information (PHI) 453**
- proxy**
 - setting up 526
- public cloud Kubernetes platforms 579**
- public cloud storage, volume types 198**
 - Amazon EBS 198, 199
 - Amazon EFS 199, 200
 - Azure data disk 202, 203
 - Azure file storage 204
 - GCE persistent disk 201, 202
- public key infrastructure (PKI) 504**
- Python client**
 - used, for accessing Kubernetes API 531, 532
- Python client library**
 - reference link 531
- Python subprocesses**
 - used, for executing Kubectl 536-538

Q

- queues**
 - benefits 323
 - downsides 323
- quotas**
 - creating 262-267
 - limit ranges, using for default compute

- quotas 267, 268
- namespace-specific context, using 262
- scopes 261
- working with 262

R

Rados Block Device (RBD) 208

- used, for connecting Ceph 208-210

Rancher k3S

- reference link 580

Rancher Kubernetes Engine (RKE)

- reference link 61
- used, for building bare metal cluster 61

Rancher RKE

- reference link 580

raw block volumes 189

- defining, with FC provider 189

readiness gates 170

ReadOnlyMany (ROX) 193

ReadWriteMany (RWX) 193

ReadWriteOnce (RWO) 193

real routable IP addresses, benefits

- performance 337
- scalability 337
- visibility 337

reclaim policy

- delete 184
- recycle 184
- retain 184

redundancy 64

redundant in-memory state

- using 225

redundant persistent storage

- DaemonSet, using 226

remote central logging 452

replica sets 10

replication controllers 10

resource categories 15

- clusters 18
- config and storage 17
- Discovery and Load Balancing 16, 17
- metadata 17
- workloads API 16

resource quotas 261

- enabling 258
- requests and limits 262

resource quotas, types 258

- compute resource quota 258
- object count quota 260
- storage resource quota 259

riff 439

- installing, with Helm 2 439-442

riff runtimes 439

- core runtime 439
- Knative runtime 439
- streaming runtime 439

rkt 27

- app container 27

role-based access control (RBAC) 55

Role-Based Access Control (RBAC) 109

rolling updates 81, 82

- complex deployments 83
- performing, with autoscaling 254-257

Romana 337, 339

Rook 213, 214

root cause

- detecting, with distributed tracing 482

routing 325

runC 26

run.sh script

- exploring 233-237

S

scalability 77, 78

scalable storage solution

- categories 269
- selecting 269

scarce resources

- handling, with limits and quotas 257, 258

Screwdriver.cd

- URL 584

seccomp

- reference link 581

secrets 11

- creating 123
- decoding 124
- storing, in Kubernetes 122
- using 122
- using, in container 124, 125

security 580

self-healing 66

sensitive log information

- dealing with 453
- sentry chart**
 - reference link 313
- serverless computing 583**
 - about 405, 406
 - FaaS, executing 407
 - long-running services, executing 406
- serverless framework**
 - used, for working with Kubeless 438
- serverless Kubernetes**
 - AWS EKS 410, 411
 - Azure AKS 409, 410
 - Azure Container Instances 409, 410
 - cluster autoscaler 408
 - Fargate 410, 411
 - Google Cloud Run 412
 - in cloud 408
- service catalog**
 - reference link 550
 - utilizing 550-552
- service-level agreements (SLAs) 92**
- service-level indicators (SLIs) 92**
- service-level objectives (SLOs)**
- service load balancer 346**
- service mesh 483-582**
 - AWS App Mesh 488
 - control plane 487
 - data plane 487
 - Envoy 487
 - Istio 488
 - Kuma 488
 - Linkerd 2 487
 - Maesh 488
 - selecting 487
 - visualizing, with Kiali 522
- services 9**
- settings commands 136**
- shared environment variables**
 - versus DNS records 223
- sidecar container 451**
- sidecar container KEP**
 - reference link 582
- sidecar pattern 13**
- simple delivery 423**
- single-node cluster**
 - checking 37, 38
 - creating 35, 36
 - creating, on macOS 34, 35
 - creating, on Windows 33, 34
 - creating, requisites 33
 - creating, with Minikube 32
 - examining, with dashboard 40, 42
 - running 38, 39
 - troubleshooting 36
- single point of failure (SPOF) 323**
- Site reliability engineering (SRE) 92**
- smart load balancing 65**
- software-defined networking (SDN)**
- Span 472**
- SPIFEE**
 - URL 504
- Spinnaker**
 - URL 584
- Stackube 28**
 - reference link 60
- starter packs**
 - advantage 302
- stateful applications**
 - in Kubernetes 221
- StatefulSet 10, 11**
 - components 226-228
 - used, to create Cassandra cluster 241
 - using 226
 - utilizing 226
- StatefulSet YAML file**
 - dissecting 241-246
- stateless applications**
 - in Kubernetes 221
- Storage Area Network (SAN) 212**
- storage class 191, 192**
- storage resource quota 259**
- subprocess 536**
- subscriptions 422**

T

- tags field 306**
- taints 151, 152**
- Tekton**
 - reference link 413
 - URL 584
- templates files**
 - pipelines and functions, using 308
 - writing 307, 308

- Tencent** 57
- Tencent Kubernetes engine (TKE)** 57
- Tencent TKE**
 - URL 579
- Terraform** 395
- Tiller server**
 - installing, for Helm 2 286
 - installing, in cluster 287
- Time to Live (TTL)** 276
- Token Controller** 105
- tolerations** 151, 152
- tooling** 578
- top-of-rack (ToR)** 338
- Trace** 472
- Traefic** 351, 352
- Traefik**
 - URL 488
- transport authentication** 504
- trigger** 422
- triggers**
 - HTTP trigger 429
 - Kubernetes watch trigger 429
 - Message queue trigger 429
 - Timer trigger 429
- troubleshooting commands** 136
- troubleshooting problems** 478
 - daemons 479, 480
 - detecting, at node level 479
 - staging environments, advantage 478

U

- unique ID (UID)** 7
- users**
 - authenticating 106, 107
 - impersonating 108

V

- values**
 - feeding, from file 312, 313
- Velero** 74
 - reference link 74
- vertical pod autoscaler (VPA)** 80
 - considering 80
 - limitations 80
- veth0** 326
- VirtualBox**

- reference link 33
- Virtual Ethernet (veth) devices** 325
- Virtualized Cloud Services (VCS)** 335
- virtual kubelet**
 - URL 410
- virtual machine (VM)** 3, 76, 408
- Virtual Redundancy Router Protocol (VRRP)** 351
- VMware Kubernetes academy**
 - reference link 578
- VMware PKS**
 - reference link 580
- volume cloning** 217, 218
- volumes** 10, 176
 - emptyDir, using for intra-pod communication 176-178
 - HostPath, using for intra-node communication 178-180
 - local volumes, using for durable node storage 180
 - projecting 214
- volume snapshots** 217, 218

W

- WdaveScope**
 - reference link 581
- Weave net** 340
- webhook admission controller**
 - configuring 568-570
- webhook**
 - used, for extending Kubernetes 541
- Windows**
 - single-node cluster, creating 33, 34
- worker node components**
 - Kubelet 449
 - Kube proxy 450
- workloads API** 16

Z

- zero downtime**
 - planning 91, 92
- Zipkin**
 - URL 473

