

EXPERT INSIGHT

# Kubernetes – An Enterprise Guide

Master containerized application deployments, integrate enterprise systems, and achieve scalability

**Third Edition**

**Foreword by:**

*Ed Price, Architecture Portfolio Lead, Google Cloud*



**Marc Boorshtein  
Scott Surovich**

**packt**

# **Kubernetes – An Enterprise Guide**

Third Edition

Master containerized application deployments, integrate enterprise systems, and achieve scalability

**Marc Boorshtein**

**Scott Surovich**



# Kubernetes – An Enterprise Guide

Third Edition

Copyright © 2024 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.

**Senior Publishing Product Manager:** Rahul Nair

**Acquisition Editor – Peer Reviews:** Tejas Mhasvekar, Jane D’Souza and Swaroop Singh

**Project Editor:** Namrata Katare

**Content Development Editor:** Shikha Parashar

**Copy Editor:** Safis Editing

**Technical Editor:** Simanta Rajbangshi

**Proofreader:** Safis Editing

**Indexer:** Subalakshmi Govindhan

**Presentation Designer:** Ajay Patule

**Developer Relations Marketing Executive:** Priyadarshini Sharma

First published: October 2020

Second edition: December 2021

Third edition: August 2024

Production reference: 1270824

Published by Packt Publishing Ltd.

Grosvenor House

11 St Paul’s Square

Birmingham

B3 1RB, UK.

ISBN 978-1-83508-695-7

[www.packt.com](http://www.packt.com)

# Foreword

Kubernetes came from Google. That was back in 2014. It's open source and is now maintained by contributors from all over the world, with the Cloud Native Computing Foundation as the official developers. As the logo of a captain's wheel implies, the term is Greek for "navigator." Now, all the major companies use and provide support for it, including Google (of course), Microsoft, Amazon, Apple, Meta/Facebook, Reddit, Docker, VMware, and others. Simply, Kubernetes is a container orchestration system that allows you to automate how you deploy, scale, and maintain your applications, based on specific metrics that you determine.

For me, I first got into Kubernetes right after it was created (late 2015). I was helping run customer feedback programs for developers who were building solutions on Microsoft Azure. We needed our services to work with container orchestration systems, and that especially included Kubernetes. By the time **Azure Kubernetes Services (AKS)** launched in 2018, I was publishing architectural content on the Azure Architecture Center, as well as whitepapers, eBooks, and blog posts. Naturally, I worked with Microsoft's top solution architects to help publish AKS architecture design guidance and reference architectures. For example, I was heavily involved in a series of content called **AKS for Amazon EKS professionals** in 2022. And I later helped publish content about **Google Kubernetes Engine (GKE)** on Google Cloud Architecture Center. What I love about Kubernetes can be summed up in a quote that's about 300 years old:



*"Small minds are concerned with the extraordinary, great minds with the ordinary."*

*- Blaise Pascal*

---

All you have to do is find an ordinary problem and solve it. This is what Kubernetes does so well. It simplifies and automates the deployment, scaling, networking, availability, monitoring, and management of your container-based applications. In other words, containers help bundle and run your applications, but you need to manage them, right? Otherwise, you might run into downtime issues. You need a management system, and that's where Kubernetes comes in.

You're in for a treat! What happens when you write the same but different book three times? It keeps getting better! Authors Marc Boorshtein and Scott Surovich have over 40 years of combined experience that they're passing on to you in this book! You're going to learn about Docker and containers, you'll ramp up on Kubernetes, you'll get to know how to authenticate your clusters, you'll learn about the Kubernetes Dashboard, you'll learn how to secure your clusters with policies, and well, I'm going to stop there. As the name of the book clearly explains, this is your one-stop shop for Kubernetes in enterprise environments.

Please dig into each chapter and learn from Marc and Scott like a sponge. The more you engage with the content of this book, the more you'll remember and apply it when you sit down and realize that it's time to put Kubernetes to use! You'll be glad, because you'll have prepared for success.

*Ed Price*

*Architecture Portfolio Lead, Google Cloud Former Senior Program Manager of Architectural Publishing, Microsoft Azure Co-Author of 7 Books, including Meg the Mechanical Engineer and, from Packt: Hands-On Microservices with C# 8 and .NET Core 3, The Azure Cloud Native Architecture Mapbook, and ASP.NET Core 5 for Beginners.*

# Contributors

## About the authors

**Marc Boorshtein** has been a software engineer and consultant for twenty years and is currently the CTO of Tremolo Security, Inc. Marc has spent most of his career building identity management solutions for large enterprises, US government civilian agencies, and local government public safety systems. Marc has recently focused on applying identity to DevOps and Kubernetes, building open-source tools for automating the security of infrastructure. Marc is a CKAD and can often be found in the Kubernetes Slack channels answering questions about authentication and authorization.

*To my wife, for supporting me building Tremolo Security and giving up a full-time salary in order to do so.  
To my sons for keeping me on my toes, my mom for raising me and giving me my persistence, and in memory of my dad who pushed me to be my own boss and start my own company.*

**Scott Surovich** has been in information technology for over 25 years and currently works at a Global Tier 1 bank as the Global Container Engineering Lead and is the product owner for hybrid cloud Kubernetes deployments. Throughout his career he has worked on multiple engineering teams in large enterprises and government agencies. He also holds the CKA, CKAD, and Mirantis Kubernetes certifications, and was one of the first people to receive Google's premier certification, Google's Certified Hybrid Multi-Cloud Fellow as part of the certification pilot group

*I would like to thank my wife, Kim, for always being supportive and understanding, for my technology addiction. To my mother, Adele, and in memory of my father, Gene, for teaching me to never give up, and that I can do anything that I set my mind to. To my brother, Chris, for the friendly competitions that always kept me striving to be better. Finally, a special thank you to my colleagues for not only supporting the book, but my role and the platform offering.*

## About the reviewers

**Alessandro Vozza** is a community leader and has covered several roles at Microsoft, Red Hat and Solo.io ranging from Engineering Manager to Principal Software Engineer and Developer Advocate. With a scientific background in chemistry, he has been occupied professionally and contributing to open source for almost 3 decades. He runs the largest European cloud native community and organizes several conferences in The Netherlands around AI/ML, DevOps and of course, Kubernetes and cloud native software. He has a varied range of expertise including service meshes, distributed systems and multi-cloud/hybrid clouds. He is now starting a new exciting startup project around namespaces-as-a-service together with other long-time industry experts, Kubespaces.io. He is currently working at on the second edition of *Hands-on Microservices with Kubernetes* together with Gigi Sayfan for Packt Publishing.

**Seth McCombs** is a Senior Infrastructure Engineer at AcuityMD, currently residing in Alameda, CA. In past, he has worked as a member of teams supporting Kubernetes-based infrastructure at Crunchyroll and Workday. His background centers around containers, deployment pipelines, and building efficient developer experiences and platforms. Seth has been a member of prior Kubernetes release teams, focusing on documentation, as well as working with other open-source projects such as Jenkins and Midpoint. His free time is spent drumming with one of his bands, playing video games, and hanging out with his cat, Misha.

**Rom Adams** (né Romuald Vandepoel) is an open source and C-Suite advisor with 20 years of experience in the IT industry. He is a cloud-native expert who helps organizations to modernize and transform with open source solutions. He is advising companies and lawmakers on their open and inner-source strategies. He has previously worked as a principal architect at Ondat, a cloud-native storage company acquired by Akamai, where he designed products and hybrid cloud solutions. He has also held roles at Tyco, NetApp, and Red Hat, becoming a subject matter expert in hybrid cloud. Adams has been a moderator and speaker for several events, sharing his insights on culture, process, and technology adoption, as well as his passion for open innovation.

*To my grandmother for her kindness, my grandfather for his wisdom, and my partner and best friend, Mercedes Adams, for her love, patience, and continuous support.*

**Werner Dijkerman** is a freelance platform, Kubernetes (certified), and Dev(Sec)Ops engineer. He currently focuses on and works with cloud-native solutions and tools, including AWS, Ansible, Kubernetes, Argo CD and Terraform. He focuses on infrastructure as code and monitoring the correct “thing,” with tools such as Zabbix, Prometheus, Wazuh and the ELK stack. He has a passion for automating everything and avoiding doing anything that resembles manual work. He is an active reader of comics, self-care/psychology, and IT-related books. He is also a technical reviewer of various books about DevOps, CI/CD, and Kubernetes.

**Abdel Sghiouar** is a senior cloud developer advocate, a co-host of the Kubernetes Podcast, and a CNCF Ambassador. His focused areas are Kubernetes, Service Mesh, and Serverless. Abdel started his career in data centers and infrastructure in Morocco, where he is originally from, before moving to Google’s largest EU data center in Belgium. Then, in Sweden, he joined Google Cloud Professional Services and spent five years working with Google Cloud customers on architecting and designing large-scale distributed systems before turning to advocacy and community work.

Abdel has worked on *Building Serverless Applications with Google Cloud Run* (O'Reilly, 2020).

## **Join our book's Discord space**

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# Table of Contents

---

**Preface****xxvii**

<b>Chapter 1: Docker and Container Essentials</b>	<b>1</b>
Technical requirements .....	2
Understanding the need for containerization .....	2
Understanding why Kubernetes removed Docker .....	3
Introducing Docker • 4	
Docker versus Moby • 4	
Understanding Docker .....	5
Containers are ephemeral • 5	
Docker images • 7	
Image layers • 7	
Persistent data • 8	
Accessing services running in containers • 8	
Installing Docker .....	8
Preparing to install Docker • 10	
Installing Docker on Ubuntu • 10	
Granting Docker permissions • 12	
Using the Docker CLI .....	13
docker help • 13	
docker run • 13	
docker ps • 15	
docker start and stop • 16	
docker attach • 16	
docker exec • 18	
docker logs • 18	
docker rm • 19	

docker pull/run • 20	
docker build • 20	
<b>Summary .....</b>	<b>20</b>
<b>Questions .....</b>	<b>21</b>
<hr/>	
<b>Chapter 2: Deploying Kubernetes Using KinD</b>	<b>23</b>
<hr/>	
<b>Technical requirements .....</b>	<b>24</b>
<b>Introducing Kubernetes components and objects .....</b>	<b>24</b>
Interacting with a cluster • 26	
<b>Using development clusters .....</b>	<b>26</b>
Why did we select KinD for this book? • 27	
Working with a basic KinD Kubernetes cluster • 28	
Understanding the node image • 32	
KinD and Docker networking • 32	
<i>Keeping track of the nesting dolls</i> • 33	
<b>Installing KinD .....</b>	<b>35</b>
Installing KinD – prerequisites • 35	
<i>Installing kubectl</i> • 35	
Installing the KinD binary • 36	
<b>Creating a KinD cluster .....</b>	<b>37</b>
Creating a simple cluster • 37	
Deleting a cluster • 38	
Creating a cluster config file • 38	
Multi-node cluster configuration • 40	
Customizing the control plane and Kubelet options • 42	
Creating a custom KinD cluster • 43	
<b>Reviewing your KinD cluster .....</b>	<b>44</b>
KinD storage objects • 44	
Storage drivers • 45	
KinD storage classes • 45	
Using KinD's Storage Provisioner • 46	
<b>Adding a custom load balancer for Ingress .....</b>	<b>47</b>
Creating the KinD cluster configuration • 48	
The HAProxy configuration file • 49	
Understanding HAProxy traffic flow • 51	
Simulating a kubelet failure • 52	

Summary .....	54
Questions .....	54
<b>Chapter 3: Kubernetes Bootcamp</b>	<b>57</b>
Technical requirements .....	58
An overview of Kubernetes components .....	58
Exploring the control plane .....	59
The Kubernetes API server • 59	
The etcd database • 60	
kube-scheduler • 62	
kube-controller-manager • 62	
cloud-controller-manager • 63	
Understanding the worker node components .....	63
kubelet • 63	
kube-proxy • 63	
Container runtime • 63	
Interacting with the API server .....	64
Using the Kubernetes kubectl utility • 64	
Understanding the verbose option • 65	
General kubectl commands • 66	
Introducing Kubernetes resources .....	67
Kubernetes manifests • 67	
What are Kubernetes resources? • 68	
Reviewing Kubernetes resources • 70	
<i>Apiservices</i> • 70	
<i>CertificateSigningRequests</i> • 71	
<i>ClusterRoles</i> • 71	
<i>ClusterRoleBindings</i> • 71	
<i>ComponentStatus</i> • 71	
<i>ConfigMaps</i> • 71	
<i>ControllerRevisions</i> • 73	
<i>CronJobs</i> • 73	
<i>CSI drivers</i> • 74	
<i>CSI nodes</i> • 74	
<i>CSISorageCapacities</i> • 74	
<i>CustomResourceDefinitions</i> • 74	
<i>DaemonSets</i> • 75	

<i>Deployments</i>	• 75
<i>Endpoints</i>	• 75
<i>EndPointSlices</i>	• 75
<i>Events</i>	• 76
<i>FlowSchemas</i>	• 76
<i>HorizontalPodAutoscalers</i>	• 76
<i>IngressClasses</i>	• 77
<i>Ingress</i>	• 77
<i>Jobs</i>	• 77
<i>LimitRanges</i>	• 77
<i>LocalSubjectAccessReview</i>	• 78
<i>MutatingWebhookConfiguration</i>	• 78
<i>Namespaces</i>	• 78
<i>NetworkPolicies</i>	• 79
<i>Nodes</i>	• 79
<i>PersistentVolumeClaims</i>	• 80
<i>PersistentVolumes</i>	• 80
<i>PodDisruptionBudgets</i>	• 80
<i>Pods</i>	• 80
<i>PodTemplates</i>	• 81
<i>PriorityClasses</i>	• 81
<i>PriorityLevelConfigurations</i>	• 81
<i>ReplicaSets</i>	• 82
<i>Replication controllers</i>	• 82
<i>ResourceQuotas</i>	• 82
<i>RoleBindings</i>	• 84
<i>Roles</i>	• 84
<i>RuntimeClasses</i>	• 84
<i>Secrets</i>	• 85
<i>SelfSubjectAccessReviews</i>	• 86
<i>SelfSubjectRulesReviews</i>	• 86
<i>Service accounts</i>	• 86
<i>Services</i>	• 87
<i>StatefulSets</i>	• 88
<i>Storage classes</i>	• 90
<i>SubjectAccessReviews</i>	• 91

<i>TokenReviews</i> • 91	
<i>ValidatingWebhookConfigurations</i> • 91	
<i>VolumeAttachments</i> • 92	
<b>Summary</b> .....	<b>92</b>
<b>Questions</b> .....	<b>92</b>
<hr/>	
<b>Chapter 4: Services, Load Balancing, and Network Policies</b>	<b>95</b>
<hr/>	
<b>Technical requirements</b> .....	<b>96</b>
<b>Exposing workloads to requests</b> .....	<b>96</b>
Understanding how Services work • 96	
<i>Creating a Service</i> • 98	
<i>Using DNS to resolve services</i> • 100	
Understanding different service types • 101	
<i>The ClusterIP service</i> • 101	
<i>The NodePort service</i> • 102	
<i>The LoadBalancer service</i> • 105	
<i>The ExternalName service</i> • 105	
<b>Introduction to load balancers</b> .....	<b>107</b>
Understanding the OSI model • 107	
<b>Layer 7 load balancers</b> .....	<b>109</b>
Name resolution and layer 7 load balancers • 109	
Using nip.io for name resolution • 111	
Creating Ingress rules • 112	
Resolving Names in Ingress Controllers • 114	
Using Ingress Controllers for non-HTTP traffic • 115	
<b>Layer 4 load balancers</b> .....	<b>116</b>
Layer 4 load balancer options • 116	
Using MetalLB as a layer 4 load balancer • 117	
<i>Installing MetalLB</i> • 117	
<i>Understanding MetalLB's custom resources</i> • 118	
<i>MetalLB components</i> • 120	
Creating a LoadBalancer service • 122	
Advanced pool configurations • 123	
<i>Disabling automatic address assignments</i> • 123	
<i>Assigning a static IP address to a service</i> • 123	
<i>Using multiple address pools</i> • 125	

<i>IP pool scoping</i> • 127	
<i>Handling buggy networks</i> • 128	
Using multiple protocols • 129	
<b>Introducing Network Policies .....</b>	<b>130</b>
Network policy object overview • 131	
<i>The podSelector</i> • 131	
<i>The policyTypes</i> • 131	
Creating a Network Policy • 132	
Tools to create network policies • 135	
<b>Summary .....</b>	<b>136</b>
<b>Questions .....</b>	<b>136</b>
<b>Chapter 5: External DNS and Global Load Balancing</b>	<b>139</b>
<b>Technical requirements .....</b>	<b>140</b>
<b>Making service names available externally .....</b>	<b>140</b>
Setting up ExternalDNS • 141	
Integrating ExternalDNS and CoreDNS • 141	
Adding an ETCD zone to CoreDNS • 142	
ExternalDNS configuration options • 144	
Creating a LoadBalancer service with ExternalDNS integration • 145	
<i>Integrating CoreDNS with an enterprise DNS server</i> • 147	
<b>Exposing CoreDNS to external requests .....</b>	<b>149</b>
Configuring the primary DNS server • 150	
Testing DNS forwarding to CoreDNS • 150	
<b>Load balancing between multiple clusters .....</b>	<b>152</b>
Introducing the Kubernetes Global Balancer • 154	
Requirements for K8GB • 155	
Deploying K8GB to a cluster • 156	
<i>Understanding K8GB load balancing options</i> • 156	
<i>Customizing the Helm chart values</i> • 158	
<i>Using Helm to install K8GB</i> • 160	
<i>Delegating our load balancing zone</i> • 160	
Deploying a highly available application using K8GB • 162	
<i>Adding an application to K8GB using custom resources</i> • 163	
<i>Adding an application to K8GB using Ingress annotations</i> • 164	
Understanding how K8GB provides global load balancing • 164	
<i>Keeping the K8GB CoreDNS servers in sync</i> • 164	

Summary .....	168
Questions .....	168
<b>Chapter 6: Integrating Authentication into Your Cluster</b>	<b>171</b>
Technical requirements .....	172
Getting Help .....	172
Understanding how Kubernetes knows who you are .....	172
External users • 172	
Groups in Kubernetes • 173	
Service accounts • 173	
Understanding OpenID Connect .....	174
The OpenID Connect protocol • 175	
Following OIDC and the API's interaction • 176	
<i>id_token</i> • 179	
Other authentication options • 180	
<i>Certificates</i> • 180	
<i>Service accounts</i> • 181	
<i>TokenRequest API</i> • 183	
<i>Custom authentication webhooks</i> • 185	
Configuring KinD for OpenID Connect .....	185
Addressing the requirements • 185	
<i>Using LDAP and Active Directory with Kubernetes</i> • 185	
<i>Mapping Active Directory groups to RBAC RoleBindings</i> • 186	
<i>Kubernetes Dashboard access</i> • 186	
<i>Kubernetes CLI access</i> • 186	
<i>Enterprise compliance requirements</i> • 187	
<i>Pulling it all together</i> • 187	
<i>Deploying OpenUnison</i> • 188	
<i>Configuring the Kubernetes API to use OIDC</i> • 191	
<i>Verifying OIDC integration</i> • 192	
<i>Using your tokens with kubectl</i> • 194	
Introducing impersonation to integrate authentication with cloud-managed clusters .....	196
What is Impersonation? • 197	
Security considerations • 199	
Configuring your cluster for impersonation .....	199
Testing Impersonation • 200	
Using Impersonation for Debugging • 201	

Configuring Impersonation without OpenUnison • 202	
Impersonation RBAC policies • 202	
Default groups • 203	
Inbound Impersonation • 203	
Privileged Access to Clusters • 205	
Using a Privileged User Account • 205	
Impersonating a Privileged User • 206	
Temporarily Authorizing Privilege • 206	
<b>Authenticating from pipelines .....</b>	<b>207</b>
Using tokens • 208	
Using certificates • 210	
Using a pipeline's identity • 212	
Avoiding anti-patterns • 215	
<b>Summary .....</b>	<b>216</b>
<b>Questions .....</b>	<b>216</b>
<b>Answers .....</b>	<b>217</b>
<b>Chapter 7: RBAC Policies and Auditing</b>	<b>219</b>
<b>Technical requirements .....</b>	<b>219</b>
<b>Introduction to RBAC .....</b>	<b>220</b>
What's a Role? • 220	
Identifying a Role • 221	
Roles versus ClusterRoles • 222	
Negative Roles • 223	
Aggregated ClusterRoles • 224	
RoleBindings and ClusterRoleBindings • 225	
<i>Combining ClusterRoles and RoleBindings</i> • 227	
Mapping enterprise identities to Kubernetes to authorize access to resources .....	228
Implementing namespace multi-tenancy .....	229
Kubernetes auditing .....	230
Creating an audit policy • 231	
Enabling auditing on a cluster • 232	
Using audit2rbac to debug policies .....	235
Summary .....	239
Questions .....	239
Answers .....	240

<b>Chapter 8: Managing Secrets</b>	<b>241</b>
Technical Requirements .....	241
Getting Help .....	242
Examining the difference between Secrets and Configuration Data .....	242
Managing Secrets in an Enterprise • 244	
<i>Threats to Secrets at Rest</i> • 245	
<i>Threats to Secrets in Transit</i> • 246	
<i>Protecting Secrets in Your Applications</i> • 247	
Understanding Secrets Managers .....	248
Storing Secrets as Secret Objects • 248	
<i>Sealed Secrets</i> • 249	
<i>External Secrets Managers</i> • 250	
<i>Using a Hybrid of External Secrets Management and Secret Objects</i> • 254	
Integrating Secrets into Your Deployments .....	256
Volume Mounts • 257	
<i>Using Kubernetes Secrets</i> • 258	
<i>Using Vault's Sidecar Injector</i> • 260	
Environment Variables • 263	
<i>Using Kubernetes Secrets</i> • 263	
<i>Using the Vault Sidecar</i> • 265	
Using the Kubernetes Secrets API • 266	
Using the Vault API • 267	
Summary .....	268
Questions .....	268
Answers .....	269
<b>Chapter 9: Building Multitenant Clusters with vClusters</b>	<b>271</b>
Technical requirements .....	271
Getting Help .....	272
The Benefits and Challenges of Multitenancy .....	272
Exploring the Benefits of Multitenancy • 272	
The Challenges of Multitenant Kubernetes • 273	
Using vClusters for Tenants .....	275
Deploying vClusters • 277	
Securely Accessing vClusters • 280	
Accessing External Services from a vCluster • 283	

Creating and Operating High-Availability vClusters • 286	
<i>Understanding vCluster High Availability</i> • 286	
<i>Upgrading vClusters</i> • 288	
<b>Building a Multitenant Cluster with Self Service .....</b>	<b>288</b>
Analyzing Requirements • 289	
Designing the Multitenant Platform • 290	
Deploying Our Multitenant Platform • 292	
<b>Summary .....</b>	<b>297</b>
<b>Questions .....</b>	<b>297</b>
<b>Answers .....</b>	<b>298</b>
<b>Chapter 10: Deploying a Secured Kubernetes Dashboard</b>	<b>299</b>
Technical requirements .....	300
Getting help • 300	
How does the dashboard know who you are? .....	300
Dashboard architecture • 301	
Authentication methods • 302	
Understanding dashboard security risks .....	303
Exploring Dashboard Security Issues • 304	
Using a token to log in • 304	
Unencrypted Connections • 304	
Deploying the dashboard with a reverse proxy .....	305
Local dashboards • 306	
Other cluster-level applications • 308	
Integrating the dashboard with OpenUnison .....	308
What's changed in the Kubernetes Dashboard 7.0 .....	310
Summary .....	310
Questions .....	310
Answers .....	311
<b>Chapter 11: Extending Security Using Open Policy Agent</b>	<b>313</b>
Technical requirements .....	313
Introduction to dynamic admission controllers .....	314
What is OPA and how does it work? .....	315
OPA architecture • 315	
Rego, the OPA policy language • 317	

Gatekeeper • 318	
<i>Deploying Gatekeeper</i> • 318	
Automated testing framework • 319	
<b>Using Rego to write policies .....</b>	<b>319</b>
Developing an OPA policy • 320	
Testing an OPA policy • 321	
Deploying policies to Gatekeeper • 323	
Building dynamic policies • 326	
Debugging Rego • 329	
Using existing policies • 330	
<b>Enforcing Ingress policies .....</b>	<b>330</b>
Enabling the Gatekeeper cache • 331	
Mocking up test data • 332	
Building and deploying our policy • 333	
<b>Mutating objects and default values .....</b>	<b>334</b>
<b>Creating policies without Rego .....</b>	<b>337</b>
Using Kubernetes' validating admission policies • 338	
<b>Summary .....</b>	<b>340</b>
<b>Questions .....</b>	<b>340</b>
<b>Answers .....</b>	<b>341</b>
<b>Chapter 12: Node Security with Gatekeeper</b>	<b>343</b>
<b>Technical requirements .....</b>	<b>343</b>
<b>What is node security? .....</b>	<b>344</b>
Understanding the difference between containers and VMs • 344	
Container breakouts • 345	
Properly designing containers • 347	
Using and Debugging Distroless Images • 348	
Scanning Images for Known Exploits • 350	
<b>Enforcing node security with Gatekeeper .....</b>	<b>351</b>
What about Pod Security Policies? • 352	
What are the differences between PSPs, PSA, and Gatekeeper? • 352	
Authorizing node security policies • 353	
Deploying and debugging node security policies • 354	
<i>Generating security context defaults</i> • 355	
<i>Enforcing cluster policies</i> • 355	

<i>Debugging constraint violations</i> • 356	
<i>Scaling policy deployment in multi-tenant clusters</i> • 360	
<b>Using Pod Security Standards to enforce Node Security .....</b>	<b>365</b>
<b>Summary .....</b>	<b>366</b>
<b>Questions .....</b>	<b>366</b>
<b>Answers .....</b>	<b>367</b>
<b>Chapter 13: KubeArmor Securing Your Runtime</b>	<b>369</b>
<b>Technical requirements .....</b>	<b>370</b>
<b>What is runtime security? .....</b>	<b>370</b>
<b>Introducing KubeArmor .....</b>	<b>372</b>
Introduction to Linux Security • 372	
Welcome to KubeArmor • 374	
<i>Container security</i> • 374	
<i>Inline mitigation versus post-attack mitigation</i> • 374	
<i>Zero-day vulnerability</i> • 376	
<i>CI/CD pipeline integration</i> • 376	
<i>Robust auditing and logging</i> • 377	
<i>Enhanced container visibility</i> • 377	
<i>Least privilege tenet adherence</i> • 377	
<i>Policy enforcement</i> • 377	
<i>Staying in compliance</i> • 377	
<i>Policy impact testing</i> • 377	
<i>Multi-tenancy support</i> • 377	
Cluster requirements for the exercises • 378	
<b>Deploying KubeArmor .....</b>	<b>378</b>
<b>Enabling KubeArmor logging .....</b>	<b>380</b>
<b>KubeArmor and LSM policies .....</b>	<b>382</b>
<b>Creating a KubeArmorSecurityPolicy .....</b>	<b>384</b>
<b>Using karmor to interact with KubeArmor .....</b>	<b>389</b>
karmor install and uninstall • 390	
karmor probe • 390	
karmor profile • 390	
karmor recommend • 392	
karmor logs • 397	
karmor vm • 400	

Summary .....	400
Questions .....	400
Answers .....	401
<b>Chapter 14: Backing Up Workloads</b>	<b>403</b>
Technical requirements .....	404
Understanding Kubernetes backups .....	404
Performing an etcd backup .....	404
Backing up the required certificates • 405	
Backing up the etcd database • 405	
Introducing and setting up VMware’s Velero .....	407
Velero requirements • 408	
Installing the Velero CLI • 408	
Installing Velero • 408	
<i>Backup storage location</i> • 409	
<i>Deploying MinIO</i> • 410	
<i>Exposing MinIO and the console</i> • 410	
<i>Installing Velero</i> • 412	
Using Velero to back up workloads and PVCs .....	414
Backing up PVCs • 414	
<i>Using the opt-out approach</i> • 415	
<i>Using the opt-in approach</i> • 415	
<i>Limitations of backing up data</i> • 416	
Running a one-time cluster backup • 416	
Scheduling a cluster backup • 421	
Creating a custom backup • 423	
Managing Velero using the CLI .....	424
Using common Velero commands • 425	
<i>Listing Velero objects</i> • 425	
<i>Retrieving details for a Velero object</i> • 426	
<i>Creating and deleting objects</i> • 426	
Restoring from a backup .....	428
Restoring in action • 429	
<i>Restoring a deployment from a backup</i> • 429	
<i>Simulating a failure</i> • 429	
Restoring a namespace • 430	

Using a backup to create workloads in a new cluster • 431	
<i>Backing up the cluster</i> • 431	
<i>Building a new cluster</i> • 433	
Restoring a backup to the new cluster • 433	
<i>Installing Velero in the new cluster</i> • 434	
<i>Restoring a backup in a new cluster</i> • 434	
<i>Deleting the new cluster</i> • 436	
<b>Summary</b> .....	<b>436</b>
<b>Questions</b> .....	<b>436</b>
<b>Answers</b> .....	<b>437</b>
<hr/> <b>Chapter 15: Monitoring Clusters and Workloads</b> <span style="float: right;"><b>439</b></span>	
Technical Requirements .....	439
Getting Help • 440	
Managing Metrics in Kubernetes .....	440
How Kubernetes Provides Metrics .....	440
Deploying the Prometheus Stack .....	441
Introduction to Prometheus • 442	
<i>How Does Prometheus Collect Metrics?</i> • 444	
<i>Common Kubernetes Metrics</i> • 445	
<i>Querying Prometheus with PromQL</i> • 447	
Alerting with Alertmanager • 450	
<i>How Do You Know Whether Something Is Broken?</i> • 451	
<i>Alerting Your Team Based on Metrics</i> • 453	
<i>Silencing Alerts</i> • 456	
Visualizing Data with Grafana • 457	
<i>Creating Your Own Graphs</i> • 457	
Monitoring Applications • 458	
<i>Why You Should Add Metrics to Your Applications</i> • 458	
<i>Adding Metrics to OpenUnison</i> • 459	
<i>Securing Access to the Metrics Endpoint</i> • 461	
Securing Access to Your Monitoring Stack • 462	
<b>Log Management in Kubernetes</b> .....	<b>464</b>
Understanding Container Logs • 464	
Introducing OpenSearch • 464	
Deploying OpenSearch • 465	

Tracing Logs from Your Container to Your Console • 466	
Viewing Log Data in Kibana • 469	
<b>Summary .....</b>	<b>474</b>
<b>Questions .....</b>	<b>475</b>
<b>Answers .....</b>	<b>475</b>
 <b>Chapter 16: An Introduction to Istio</b> <span style="float: right;"><b>477</b></span>	
<b>Technical requirements .....</b>	<b>478</b>
<b>Understanding the Control Plane and Data Plane .....</b>	<b>479</b>
The Control Plane • 479	
The Data Plane • 479	
<b>Why should you care about a Service mesh? .....</b>	<b>479</b>
Workload observability • 480	
Traffic management • 480	
<i>Blue/green deployments</i> • 480	
<i>Canary deployments</i> • 481	
Finding issues before they happen • 481	
Security • 481	
<b>Introduction to Istio concepts .....</b>	<b>482</b>
Understanding the Istio components • 482	
<i>Making the Control Plane simple with istiod</i> • 482	
<i>Understanding istio-ingressgateway</i> • 484	
<i>Understanding istio-egressgateway</i> • 485	
<b>Installing Istio .....</b>	<b>485</b>
Downloading Istio • 486	
Installing Istio using a profile • 486	
Exposing Istio in a KinD cluster • 489	
<b>Introducing Istio resources .....</b>	<b>489</b>
Authorization policies • 489	
<i>Example 1: Denying and allowing all access</i> • 492	
<i>Example 2: Allowing only GET methods to a workload</i> • 493	
<i>Example 3: Allowing requests from a specific source</i> • 494	
<i>Gateways</i> • 495	
<i>Virtual services</i> • 497	
Destination rules • 498	
<i>Peer authentication</i> • 499	
<i>Request authentication and authorization policies</i> • 500	

<i>Service entries</i> • 501	
<i>Sidecars</i> • 502	
<i>Envoy filters</i> • 503	
<i>WASM plugins</i> • 503	
<b>Deploying add-on components to provide observability .....</b>	<b>504</b>
Installing Istio add-ons • 504	
Installing Kiali • 505	
<b>Deploying an application into the Service mesh .....</b>	<b>505</b>
Deploying your first application into the mesh • 505	
Using Kiali to observe mesh workloads • 506	
<i>The Kiali overview screen</i> • 506	
<i>Using the Graph view</i> • 507	
<i>Using the Applications view</i> • 511	
<i>Using the Workloads view</i> • 513	
<i>Using the Services view</i> • 515	
<i>The Istio Config view</i> • 516	
The future: Ambient mesh .....	519
Summary .....	520
Questions .....	520
Answers .....	521
<b>Chapter 17: Building and Deploying Applications on Istio</b>	<b>523</b>
Technical requirements .....	523
Comparing microservices and monoliths .....	524
My history with microservices versus monolithic architecture • 524	
Comparing architectures in an application • 525	
<i>Monolithic application design</i> • 525	
<i>Microservices design</i> • 526	
<i>Choosing between monoliths and microservices</i> • 528	
<i>Using Istio to help manage microservices</i> • 528	
Deploying a monolith .....	529
Exposing our monolith outside our cluster • 530	
Configuring sticky sessions • 532	
Integrating Kiali and OpenUnison • 533	
Building a microservice .....	535
Deploying Hello World • 536	
Integrating authentication into our service • 537	

Authorizing access to our service • 540	
Telling your service who's using it • 542	
Authorizing user entitlements • 545	
<i>Authorizing in service</i> • 545	
<i>Using OPA with Istio</i> • 545	
<i>Creating an OPA Authorization Rule</i> • 547	
Calling other services • 549	
<i>Using OAuth2 Token Exchange</i> • 549	
<i>Passing tokens between services</i> • 559	
<i>Using simple impersonation</i> • 559	
<b>Do I need an API gateway?</b> .....	<b>560</b>
Summary .....	561
Questions .....	561
<b>Chapter 18: Provisioning a Multitenant Platform</b>	<b>563</b>
Technical requirements .....	563
Designing a pipeline .....	564
Opinionated platforms • 565	
Securing your pipeline • 566	
Building our platform's requirements • 566	
Choosing our technology stack • 568	
Designing our platform architecture .....	570
Securely managing a remote Kubernetes cluster • 571	
Securely pushing and pulling images • 576	
Using Infrastructure as Code for deployment .....	576
Automating tenant onboarding .....	579
Designing a GitOps strategy • 582	
Considerations for building an Internal Developer Platform .....	585
Summary .....	586
Questions .....	587
Answers .....	588
<b>Chapter 19: Building a Developer Portal</b>	<b>589</b>
Technical Requirements .....	590
Fulfilling Compute Requirements • 590	
<i>Using Cloud-Managed Kubernetes</i> • 590	
<i>Building a Home Lab</i> • 591	

---

Customizing Nodes • 592	
Accessing Services on Your Nodes • 593	
Deploying Pulumi • 594	
<b>Deploying our IDP .....</b>	<b>595</b>
Setting Up Pulumi • 596	
Initial Deployment • 597	
Unsealing Vault • 599	
Completing the Harbor Configuration • 600	
Completing the GitLab Configuration • 601	
<i>Generating a GitLab Runner • 601</i>	
<i>Generating a GitLab Personal Access Token • 604</i>	
Finishing the Control Plane Rollout • 607	
Integrating Development and Production • 609	
Bootstrapping GitOps with OpenUnison • 610	
<b>Onboarding a Tenant .....</b>	<b>613</b>
<b>Deploying an Application .....</b>	<b>616</b>
Promoting to Production • 619	
<b>Adding Users to a Tenant .....</b>	<b>620</b>
<b>Expanding Our Platform .....</b>	<b>620</b>
Different Sources of Identity • 621	
Integrating Monitoring and Logging • 621	
Integrating Policy Management • 621	
Replacing Components • 622	
<b>Summary .....</b>	<b>622</b>
<b>Questions .....</b>	<b>622</b>
<b>Answers .....</b>	<b>623</b>
<b>Other Books You May Enjoy</b>	<b>627</b>
<hr/>	
<b>Index</b>	<b>631</b>

# Preface

Kubernetes has taken the world by storm, becoming the standard infrastructure for DevOps teams to develop, test, and run applications. Most enterprises are either running it already or are planning to run it in the next year. A look at job postings on any of the major job sites shows that just about every big-name company has Kubernetes positions open. The fast rate of adoption has led to Kubernetes-related positions growing by over 2,000% in the last 4 years.

One common problem that companies are struggling to address is the lack of enterprise Kubernetes knowledge. While Kubernetes isn't new anymore (it just turned 10!), companies have had issues with trying to build teams to run clusters reliably. They've also struggled to understand how to integrate Kubernetes workloads across the multiple silos and technology stacks that are common in the enterprise world. Finding people with basic Kubernetes skills is becoming easier, but finding people with knowledge on topics that are required for enterprise clusters is still a challenge.

## Who this book is for

We created this book to help DevOps teams to expand their skills beyond the basics of Kubernetes. It was created from the years of experience we have working with clusters in multiple enterprise environments.

There are many books available that introduce Kubernetes and the basics of installing clusters, creating Deployments, and using Kubernetes objects. Our plan was to create a book that would go beyond a basic cluster, and to keep the book at a reasonable length, we have not re-hashed the basics of Kubernetes. Readers should have some experience with Kubernetes and DevOps before reading this book.

While the primary focus of the book is on extending clusters with enterprise features, the first section of the book will provide a refresher on key Docker topics and Kubernetes objects. It is important that you have a solid understanding of Kubernetes objects in order to get the most out of the more advanced chapters.

## What this book covers

*Chapter 1, Docker and Container Essentials*, covers the problems Docker and Kubernetes address for developers. You will be introduced to Docker, including the Docker daemon, data, installation, and using the Docker CLI.

*Chapter 2, Deploying Kubernetes Using KinD*, helps with creating development clusters using KinD, a powerful tool that allows you to create clusters ranging from a single-node cluster to a full multi-node cluster. The chapter goes beyond a basic KinD cluster, explaining how to use a load balancer running HAProxy to load-balance worker nodes. By the end of the chapter, you will understand how KinD works and how to create a custom multi-node cluster, which will be used for the exercises in the chapters.

*Chapter 3, Kubernetes Bootcamp*, provides a refresher on Kubernetes. This chapter will cover most of the objects that a cluster includes, which will be helpful if you are new to Kubernetes. It will explain each object with a description of what each object does and its function in a cluster. It is meant to be a refresher, or a “pocket guide” to objects. It does not contain exhaustive details for each object (that would require a second book).

*Chapter 4, Services, Load Balancing, and Network Policies*, explains how to expose a Kubernetes Deployment using services. Each service type will be explained with examples, and you will learn how to expose them using both a layer 7 and layer 4 load balancer. In this chapter, you will go beyond the basics of a simple Ingress controller, installing MetalLB, to provide layer 4 access to services. Finally, you will learn how to provide fine-grained control over the communication between pods, enhancing security and compliance within your cluster by using Kubernetes network policies.

*Chapter 5, External DNS and Global Load Balancing*, will make you learn about two add-ons that benefit enterprise clusters by installing an incubator project called external-dns to provide dynamic name resolution for the services exposed by MetalLB. You will also learn how to add a **Global Load Balancer** to your cluster, using a project called K8GB, which provides native Kubernetes Global Load Balancing.

*Chapter 6, Integrating Authentication into Your Cluster*, answers the question, “Once your cluster is built, how will users access it?” In this chapter we’ll detail how OpenID Connect works and why you should use it to access your cluster. You’ll also learn how to authenticate your pipelines, and finally, we’ll also cover several anti-patterns that should be avoided and explain why they should be avoided.

*Chapter 7, RBAC Policies and Auditing*, explains that once users have access to a cluster, you need to know how to limit their access. Whether you are providing an entire cluster to your users or just a namespace, you’ll need to know how Kubernetes authorizes access via its **role-based access control (RBAC)** system. In this chapter, we’ll detail how to design RBAC policies, how to debug them, and different strategies for multi-tenancy.

*Chapter 8, Managing Secrets*, puts the focus on one of the hardest to implement issues in the Kubernetes world: how to manage secret data. First, we’ll look at the challenges of managing Secrets in Kubernetes. Then we’ll learn about HashiCorp’s **Vault** for secret management. Finally, we’ll integrate our clusters with Vault using both the **Vault sidecar** and the popular **External Secrets Operator**.

*Chapter 9, Building Multitenant Clusters with vClusters*, moves out of a single cluster toward breaking up clusters into tenants using the **vCluster** project from Loft. You’ll learn how vClusters work, how they interact with host clusters, how to securely access them, and how to automate their rollout for your tenants. We’ll also build off what we learned in *Chapter 8* to integrate managed Secrets into our vClusters too!

*Chapter 10, Deploying a Secured Kubernetes Dashboard*, covers Kubernetes Dashboard, which is often the first thing users try to launch once a cluster is up and running. There's quite a bit of mythology around security (or lack thereof). Your cluster will be made of other web applications too, such as network dashboards, logging systems, and monitoring dashboards. This chapter looks at how the dashboard is architected, how to properly secure it, and examples of how not to deploy it with details as to why.

*Chapter 11, Extending Security Using Open Policy Agent*, provides you with the guidance you need to deploy **Open Policy Agent** and **GateKeeper** to enable policies that can't be implemented using RBAC. We'll cover how to deploy Gatekeeper, how to write policies in Rego, and how to test your policies using OPA's built-in testing framework.

*Chapter 12, Node Security with Gatekeeper*, deals with the security of the nodes that run your pods. We will discuss how to securely design your containers so they are harder to abuse and how to build policies using GateKeeper that prevent your containers from accessing resources they don't need.

*Chapter 13, KubeArmor Securing Your Runtime*, presents security, which is the job of everyone, and providing tools to address attack vectors is key to running a secure and resilient cluster. In this chapter, you will learn how to secure your containers runtime by using a CNCF project called **KubeArmor**. KubeArmor provides an easy way to lock down containers using easy to understand policies.

*Chapter 14, Backing Up Workloads*, explains how to create a backup of your cluster workloads for disaster recovery or cluster migrations, using **Velero**. You will work hands-on to create an S3-compatible storage location using MinIO to create a backup of example workloads and persistent storage. You will then restore the backup to a brand-new cluster to simulate a cluster migration.

*Chapter 15, Monitoring Clusters and Workloads*, explores how to know how healthy your cluster is using **Prometheus** and **OpenSearch**. You'll start with understanding how Kubernetes and Prometheus handle metrics, then we'll deploy the Prometheus stack with **Alertmanager** and **Grafana**. You'll learn how to secure the stack and how to extend it to monitor additional workloads. After we're done with monitoring, we will move on to log aggregation with OpenSearch. We will start with exploring how logging in Kubernetes works, move on to integrating OpenSearch, and wrap up with securing access to OpenSearch with **OpenUnison**.

*Chapter 16, An Introduction to Istio*, explains that many enterprises use a service mesh to provide advanced features such as security, traffic routing, authentication, tracing, and observability to a cluster. This chapter will introduce you to Istio, a popular open-source mesh, and its architecture, along with the most commonly used resources it provides. You will deploy Istio to your KinD cluster with an example application and learn how to observe the behavior of an application using an observability tool called Kiali.

*Chapter 17, Building and Deploying Applications on Istio*, acknowledges that once you've deployed Istio, you'll want to develop and deploy applications that use it! This chapter starts with a walk-through of the differences between monoliths and microservices and how they're deployed. Next, we'll step through building a micro-service to run in Istio and get into advanced topics like authentication, authorization, and service-to-service authentication for your services. You will also learn how to secure Kiali access by leveraging existing roles in Kubernetes using an OIDC provider and JSON Web Tokens. You'll also learn how to secure Istio services using JWTs, along with how to use token exchanges to gain access to different services, securely moving from one service to another. Finally, we use OPA to create a custom authorization rule we configure with Istio.

*Chapter 18, Provisioning a Multitenant Platform*, explores how to build pipelines, how to automate their creation, and how they relate to GitOps. We'll explore how the objects that are used to drive pipelines are related to each other, how to build relationships between systems, and finally, design a self-service workflow for automating the Deployment of pipelines.

*Chapter 19, Building a Developer Portal*, builds off of our designs in *Chapter 18* to build out a multitenant platform with many of the tools we used throughout this book. We'll start with talking about building a lab to run our multitenant cluster in. Next we'll roll out Kubernetes to three clusters and integrate them with GitLab, Vault, Argo CD, Harbor, and OpenUnison. Finally we'll walk through onboarding a new vCluster based tenant using OpenUnison's self-service portal.

## To get the most out of this book

- You should have a basic understanding of Linux, basic commands, tools like Git, and a text editor like Vi.
- The book chapters contain both theory and hands-on exercises. We feel that the exercises help to reinforce the theory, but they are not required to understand each topic. If you want to do the exercises in the book, you will need to meet the requirement in the table below:

Requirement for the chapter exercises	Version
Ubuntu Server	22.04 or higher

All exercises use Ubuntu, but most of them will work on other Linux installations.

## Download the example code files

The code bundle for the book is hosted on GitHub at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-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://packt.link/gbp/9781835086957>.

## Supplementary content

Here's a link to the YouTube channel (created and managed by the authors Marc Boorshtein and Scott Surovich) that contains videos of the labs from this book, so you can see them in action before you start on your own: <https://packt.link/N5qjd>.

## Conventions used

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

**CodeInText:** Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. For example: “The `--name` option will set the name of the cluster to `cluster01`, and `--config` tells the installer to use the `cluster01-kind.yaml` config file.”

A block of code is set as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: grafana
  name: grafana
  namespace: monitoring
```

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: apps/v1
kind: Deployment
metadata:
  labels:
    app: grafana
  name: grafana
  namespace: monitoring
```

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

```
PS C:\Users\mlb> kubectl create ns not-going-to-work
namespace/not-going-to-work created
```

**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: “Hit the **Finish Login** button at the bottom of the screen.”



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](mailto: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](mailto: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>, select your book, click on the Errata Submission Form link, and enter 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](mailto: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>.

## Share your thoughts

Once you've read *Kubernetes - An Enterprise Guide, Third Edition*, we'd love to hear your thoughts! Please [click here](#) to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

## Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835086957>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.



# 1

## Docker and Container Essentials

Containers have become an incredibly popular and influential technology that brings significant changes from legacy applications. Everyone, from tech companies to big corporations and end users, now has widely embraced containers to handle their day-to-day tasks. It's worth noting that the conventional method of installing ready-made commercial applications is gradually transforming into fully containerized setups. Considering the sheer magnitude of this technological shift, it becomes essential for people working in the field of information technology to gain knowledge and understand the concept of containers.

This chapter will provide an overview of the issues that containers aim to solve. We will begin by highlighting the significance of containers. Then, we will introduce **Docker**, the runtime that played a pivotal role in the rise of containerization, and discuss its relationship with **Kubernetes**.

This chapter intends to provide you with an understanding of running containers in Docker. One common question you may have heard is: "What is the relationship of Docker to Kubernetes?" Well, in today's world, Docker is not tied to Kubernetes at all – you do not need Docker to run Kubernetes and you don't need it to create containers. We are discussing Docker in this chapter to provide you with the skills to run containers locally and test your images before you deploy them to a Kubernetes cluster.

By the end of this chapter, you will have a clear understanding of how to install Docker and how to effectively utilize the commonly used **Docker command-line interface (CLI)** commands.

In this chapter, we will cover the following main topics:

- Understanding the need for containerization
- Understanding why Kubernetes removed Docker
- Understanding Docker
- Installing Docker
- Using the Docker CLI

## Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, though 8 GB is suggested.
- Scripts from the `chapter1` folder from the repository, which you can access by using this link: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Understanding the need for containerization

You may have experienced a conversation like this at your office or school:

**Developer:** “*Here’s the new application. It went through weeks of testing and you are the first to get the new release.*”

.... A little while later ....

**User:** “*It’s not working. When I click the submit button, it shows an error about a missing dependency.*”

**Developer:** “*That’s weird; it’s working fine on my machine.*”

Encountering such issues can be incredibly frustrating for developers when they’re deploying an application. Oftentimes, these problems occur due to a missing library in the final package that the developer had on their own machine. One might think that a simple solution would be to include all the libraries in the release, but what if this release includes a newer version of a library that replaces an older version, which another application may still rely on?

Developers have to carefully consider their new releases and the potential conflicts they may cause with existing software on users’ workstations. It becomes a delicate balancing act that often requires larger deployment teams to thoroughly test the application on various system configurations. This situation can result in additional work for the developer or, in extreme cases, render the application completely incompatible with an existing one.

Over the years, there have been several attempts to simplify application delivery. One solution is VMware’s **ThinApp**, which aims to virtualize an application (not to be confused with virtualizing the entire operating system (OS)). It allows you to bundle the application and its dependencies into a single executable package. By doing so, all the application’s dependencies are contained within the package, eliminating conflicts with other application dependencies. This not only ensures application isolation but also enhances security and reduces the complexities of OS migrations.

You might not have come across terms like application packaging or application-on-a-stick until now, but it seems like a great solution to the infamous “it worked on my machine” problem. However, there are reasons why it hasn’t gained widespread adoption as anticipated. Firstly, most solutions in this space are paid offerings that require a significant investment. Additionally, they require a “clean PC,” meaning that for each application you want to virtualize, you need to start with a fresh system. The package you create captures the differences between the base installation and any changes made afterward. These differences are then packaged into a distribution file, which can be executed on any workstation.

We've mentioned application virtualization to highlight that application issues such as "it works on my machine" have had different solutions over the years. Products such as **ThinApp** are just one attempt at solving the problem. Other attempts include running the application on a server using **Citrix**, **Remote Desktop**, **Linux containers**, **chroot jails**, and even **virtual machines**.

## Understanding why Kubernetes removed Docker

Kubernetes removed all support for Docker in version 1.24 as a supported container runtime. While it has been removed as a runtime engine option, you can create new containers using Docker and they will run on any runtime that supports the **Open Container Initiative (OCI)** specification. OCI is a set of standards for containers and their runtimes. These standards ensure that containers remain portable, regardless of the container platform or the runtime used to execute them.

When you create a container using Docker, you are creating a container that is fully OCI compliant, so it will still run on Kubernetes clusters that are running any Kubernetes-compatible container runtime.

To fully explain the impact and the supported alternatives, we need to understand what a container runtime is. A high-level definition would be that a container runtime is the software layer that runs and manages containers. Like many components that make up a Kubernetes cluster, the runtime is not included as part of Kubernetes – it is a pluggable module that needs to be supplied by a vendor, or by you, to create a functioning cluster.

There are many technical reasons that led to the decision to deprecate and remove Docker, but at a high level, the main concerns were as follows:

- Docker contains multiple pieces inside of the Docker runtime to support its own remote API and **user experience (UX)**. Kubernetes only requires one component in the executable, dockerd, which is the runtime process that manages containers. All other pieces of the executable contribute nothing to using Docker in a Kubernetes cluster. These extra components make the binary bloated and can lead to additional bugs, security, or performance issues.
- Docker does not conform to the **Container Runtime Interface (CRI)** standard, which was introduced to create a set of standards to easily integrate container runtimes in Kubernetes. Since it doesn't comply, the Kubernetes team has had extra work that only caters to supporting Docker.

When it comes to local container testing and development, you can still use Docker on your workstation or server. Considering the previous statement, if you build a container on Docker and the container successfully runs on your Docker runtime system, it will run on a Kubernetes cluster that does not use Docker as the runtime.

Removing Docker will have very little impact on most users of Kubernetes in new clusters. Containers will still run using any standard method, as they would with Docker as the container runtime. If you happen to manage a cluster, you may need to learn new commands when you troubleshoot a Kubernetes node – you will not have a Docker command on the node to look at running containers, clean up volumes, and so on.

Kubernetes supports a number of runtimes in place of Docker. Two of the most commonly used runtimes are as follows:

- containerd
- CRI-O

While these are the two commonly used runtimes, there are a number of other compatible runtimes available. You can always view the latest supported runtimes on the Kubernetes GitHub page at <https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md>.



For more details on the impact of deprecating and removing Docker, refer to the article called *Don't Panic: Kubernetes and Docker* on the Kubernetes.io site at <https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/>.

## Introducing Docker

Both the industry and end users were seeking a solution that was both convenient and affordable, and this is where Docker containers came in. While containers have been utilized in different ways over time, Docker has brought about a transformation by providing a runtime and tools for everyday users and developers.

Docker brought an abstraction layer to the masses. It was easy to use and didn't require a clean PC for every application before creating a package, thus offering a solution for dependency issues, but most attractive of all, it was free. Docker became a standard for many projects on GitHub, where teams would often create a Docker container and distribute the Docker image or Dockerfile to team members, providing a standard testing or development environment. This adoption by end users is what eventually brought Docker to the enterprise and, ultimately, what made it the standard it has become today.

Within the scope of this book, we will be focusing on what you will need to know when trying to use a local Kubernetes environment. Docker has a long and interesting history of how it evolved into the standard container image format that we use today. We encourage you to read about the company and how they ushered in the container world we know today.

While our focus is not to teach Docker inside out, we feel that those of you who are new to Docker would benefit from a quick primer on general container concepts.

If you have some Docker experience and understand terminology such as ephemeral and stateless, you can jump to the *Installing Docker* section.

## Docker versus Moby

When the Docker runtime was developed, it was a single code base. The single code base contained every function that Docker offered, whether you have used them or not. This led to inefficiencies, and it started to hinder the progression of Docker and containers in general.

The following table shows the differences between the Docker and Moby projects.

Feature	Docker	Moby
Development	The primary contributor is Docker, with some community support	It is open-source software with heavy community development and support
Project scope	The complete platform that includes all components to build and run containers	It is a modular platform for building container-based components and solutions
Ownership	It is a branded product, offered by Docker, Inc.	It is an open-source project that is used to build various container solutions
Configuration	A full default configuration is included to make it easy for users to use it quickly	It has more available customizations, providing users with the ability to address their specific requirements
Commercial support	It offers full support, including enterprise support	It is offered as open-source software; there is no support direct from the Moby project

Table 1.1: Docker versus Moby features

To recap – **Moby** is a project that was started by Docker, but it is not the complete Docker runtime. The Docker runtime uses the components from Moby to create the Docker runtime, which includes the Moby open-source components and Docker’s own open-sourced components.

Now, let’s move on to understanding Docker a little more and how you can use it to create and manage containers.

## Understanding Docker

This book assumes that you have a foundational understanding of Docker and container concepts. However, we know that not everyone will have prior experience with Docker or containers. Therefore, we have included this crash course to introduce you to container concepts and guide you through the usage of Docker.

If you are new to containers, we suggest reading the documentation that can be found on Docker’s website for additional information: <https://docs.docker.com/>.

## Containers are ephemeral

The first thing to understand is that containers are ephemeral.

The term “ephemeral” means something that exists for a short period. **Containers** can be intentionally terminated, or automatically restarted without any user involvement or consequences. To better understand this concept, let’s look at an example – imagine someone interactively adds files to a web server running within a container. The uploaded files are temporary because they were not originally part of the base image.

This means that once a container is built and running, any changes that are made to the container will not be saved once it is removed, or destroyed, from the Docker host. Let's look at a full example:

1. You start a container running a web server using **NGINX** on your host without any base **HTML** pages.
2. Using a Docker command, you execute a `copy` command to copy some web files into the container's filesystem.
3. To test that the copy was successful, you go to the website and confirm that it is serving the correct web pages.
4. Happy with the results, you stop the container and remove it from the host. Later that day, you want to show a coworker the website and you start your **NGINX** container. You go to the website again, but when the site opens, you receive a **404** error (page not found error).

What happened to the files you uploaded before you stopped and removed the container from the host?

The reason your web pages cannot be found after the container was restarted is that all containers are ephemeral. Whatever is in the base container image is all that will be included each time the container is initially started. Any changes that you make inside a container are short-lived.

If you need to add permanent files to an existing image, you need to rebuild the image with the files included or, as we will explain in the *Persistent data* section later in this chapter, you could mount a Docker volume in your container.

At this point, the main concept to understand is that containers are **ephemeral**.

But wait! You may be wondering, "If containers are ephemeral, how did I add web pages to the server?" **Ephemeral** just means that changes will not be saved; it doesn't stop you from making changes to a running container.

Any changes made to a running container will be written to a temporary layer, called the **container layer**, which is a directory on the localhost filesystem. Docker uses a **storage driver**, which is in charge of handling requests that use the container layer. The storage driver is responsible for managing and storing images and containers on your Docker host. It controls the mechanisms and processes involved in their storage and management.

This location will store all changes in the container's filesystem so that when you add the HTML pages to the container, they will be stored on the local host. The container layer is tied to the **container ID** of the running image and it will remain on the host system until the container is removed from Docker, either by using the CLI or by running a Docker **prune job** (see *Figure 1.1* on the next page).

Considering that containers are temporary and are read only, you might wonder how it's possible to modify data within a container. Docker addresses this by utilizing **image layering**, which involves creating interconnected layers that collectively function as a single filesystem. Through this, changes can be made to the container's data, even though the underlying image remains **immutable**.

## Docker images

A Docker image is composed of multiple image layers, each accompanied by a **JavaScript Object Notation (JSON)** file that stores metadata specific to the layer. When a container image is launched, these layers are combined to form the application that users interact with.

You can read more about the contents of an image on Docker's GitHub at <https://github.com/moby/blob/master/image/spec/v1.1.md>.

## Image layers

As we mentioned in the previous section, a running container uses a **container layer** that is “on top” of the base **image layer**, as shown in the following diagram:

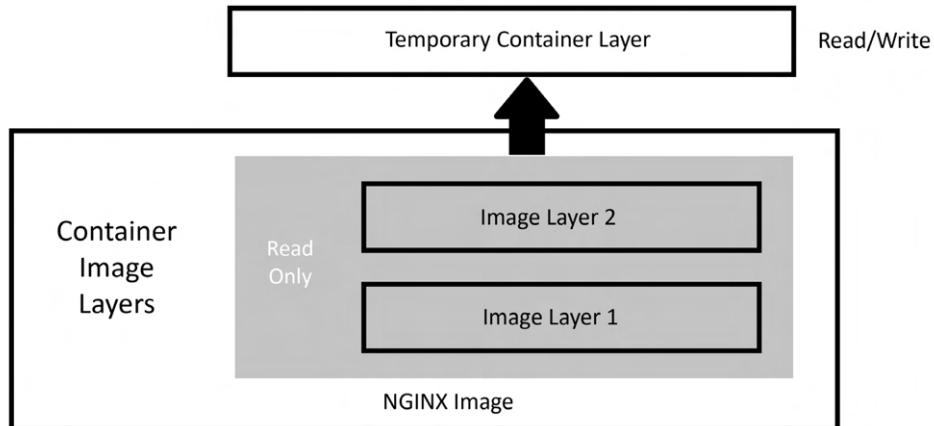


Figure 1.1: Docker image layers

The image layers cannot be written to since they are in a read-only state, but the temporary container layer is in a writeable state. Any data that you add to the container is stored in this layer and will be retained as long as the container is running.

To deal with multiple layers efficiently, Docker implements **copy-on-write**, which means that if a file already exists, it will not be created. However, if a file is required that does not exist in the current image, it will be written. In the container world, if a file exists in a lower layer, the layers above it do not need to include it. For example, if layer 1 had a file called `/opt/nginx/index.html` in it, layer 2 does not need the same file in its layer.

This explains how the system handles files that either exist or do not exist, but what about a file that has been modified? There will be times when you'll need to replace a file that is in a lower layer. You may need to do this when you are building an image or as a temporary fix to a running container issue. The copy-on-write system knows how to deal with these issues. Since images read from the top down, the container uses only the highest layer file. If your system had a `/opt/nginx/index.html` file in layer 1 and you modified and saved the file, the running container would store the new file in the container layer. Since the container layer is the topmost layer, the new copy of `index.html` would always be read before the older version in the image layer.

## Persistent data

Being limited to ephemeral-only containers would severely limit the use cases for Docker. You will probably encounter use cases where persistent storage is needed or data must be retained even if a container is stopped.

Remember, when you store data in the container image layer, the base image does not change. When the container is removed from the host, the container layer is also removed. If the same image is used to start a new container, a new container image layer is created. While containers themselves are ephemeral, you can achieve data persistence by incorporating a Docker volume. By utilizing a **Docker volume**, data can be stored externally in the container, enabling it to persist beyond the container's lifespan.

## Accessing services running in containers

Unlike a physical machine or a virtual machine, containers do not connect to a network directly. When a container needs to send or receive traffic, it goes through the Docker host system using a bridged **network address translation (NAT)** connection. This means that when you run a container and you want to receive incoming traffic requests, you need to expose the ports for each of the containers that you wish to receive traffic on. On a Linux-based system, `iptables` has rules to forward traffic to the Docker daemon, which will service the assigned ports for each container. There is no need to worry about how the `iptables` rules are created, as Docker will handle that for you by using the port information provided when you start the container. If you are new to Linux, `iptables` may be new to you.



At a high level, `iptables` is used to manage network traffic and keep it secure within a cluster. It controls the flow of network connections between components in the cluster, deciding which connections are allowed and which ones are blocked.

That concludes the introduction to container fundamentals and Docker concepts. In the next section, we will guide you through the process of installing Docker on your host.

## Installing Docker

The hands-on exercises in this book will require that you have a working Docker host. To install Docker, we have included a script located in this book's GitHub repository, in the `chapter1` directory, called `install-docker.sh`.

Today, you can install Docker on just about every hardware platform out there. Each version of Docker acts and looks the same on each platform, making development and using Docker easy for people who need to develop cross-platform applications. By making the functions and commands the same between different platforms, developers do not need to learn a different container runtime to run images.

The following is a table of Docker's available platforms. As you can see, there are installations for multiple OSs, as well as multiple architectures:

Desktop Platform	x86_64/amd64	arm64 (Apple Silicon)			
Docker Desktop (Linux)	✓				
Docker Desktop (macOS)	✓	✓			
Docker Desktop (Windows)	✓				
Server Platform	x86_64/amd64	arm64/aarch64	arm (32-bit)	ppcc64le	s390x
CentOS	✓	✓		✓	
Debian	✓	✓	✓	✓	
Fedora	✓	✓		✓	
Raspberry Pi OS			✓		
RHEL (s390)	✓				✓
SLES	✓	✓	✓	✓	✓
Ubuntu	✓	✓	✓		

Table 1.2: Available Docker platforms

Images that are created using one architecture cannot run on a different architecture. This means that you cannot create an image based on x86 hardware and expect that same image to run on your Raspberry Pi running an ARM processor. It's also important to note that while you can run a Linux container on a Windows machine, you cannot run a Windows container on a Linux machine.

While images, by default, are not cross-architecture compatible, there are new tools to create what's known as a multi-platform image. Multi-platform images are images that can be used across different architectures or processors in a single container, rather than having multiple images, such as one for NGINX on x86, another one for ARM, and another one for PowerPC. This will help you simplify your management and deployment of containerized applications. Since multi-platform images contain various versions, one for each architecture you include, you need to specify the architecture when deploying the image. Luckily, the container runtime will help out and automatically select the correct architecture from the image manifest.

The use of multi-platform images provides portability, flexibility, and scalability for your containers across cloud platforms, edge deployments, and hybrid infrastructure. With the use of ARM-based servers growing in the industry and the heavy use of Raspberry Pi by people learning Kubernetes, cross-platform images will help make consuming containers quicker and easier.

For example, in 2020, Apple released the M1 chip, ending the era of Apple running Intel processors in favor of the ARM processor. We're not going to get into the details of the difference, only that they are different and this leads to important challenges for container developers and users. Docker does have **Docker Desktop**, a macOS tool for running containers that lets you use the same workflows that you used if you had a Docker installation on Linux, Windows, or x86 macOS. Docker will try to match the architecture of the underlying host when pulling or building images. On ARM-based systems, if you are attempting to pull an image that does not have an ARM version, Docker will throw an error due to the architecture incompatibilities. If you are attempting to build an image, it will build an ARM version on macOS, which cannot run on x86 machines.

Multi-platform images can be complex to create. If you want additional details on creating multi-platform images, visit the *Multi-platform images* page on Docker's website: <https://docs.docker.com/build/building/multi-platform/>.

The installation procedures that are used to install Docker vary between platforms. Luckily, Docker has documented many of them on their website: <https://docs.docker.com/install/>.

In this chapter, we will install Docker on an **Ubuntu 22.04** system. If you do not have an Ubuntu machine to install on, you can still read about the installation steps, as each step will be explained and does not require that you have a running system to understand the process. If you have a different Linux installation, you can use the installation procedures outlined on Docker's site at <https://docs.docker.com/>. Steps are provided for CentOS, Debian, Fedora, and Ubuntu, and there are generic steps for other Linux distributions.

## Preparing to install Docker

Now that we have introduced Docker, the next step is to select an installation method. Docker's installation changes between not only different Linux distributions but also versions of the same Linux distribution. Our script is based on using an Ubuntu 22.04 server, so it may not work on other versions of Ubuntu. You can install Docker using one of two methods:

- Add the Docker repositories to your host system
- Install using Docker scripts

The first option is considered the best option since it allows for easy installation and updates to the Docker engine. The second option is designed for installing Docker on testing/development environments and is not recommended for deployment in production environments.

Since the preferred method is to add Docker's repository to our host, we will use that option.

## Installing Docker on Ubuntu

Now that we have added the required repositories, the next step is to install Docker.

We have provided a script in the `chapter1` folder of the Git repository called `install-docker.sh`. When you execute the script, it will automatically install all of the necessary binaries required for Docker to run.

To provide a brief summary of the script, it begins by modifying a specific value in the `/etc/needrestart/needrestart.conf` file. In Ubuntu 22.04, there was a change in how daemons are restarted, where users might be required to manually select which system daemons to restart. To simplify the exercises described in the book, we alter the `restart` value in the `needsrestart.conf` file to “automatic” instead of prompting for each changed service.

Next, we install a few utilities like `vim`, `ca-certificates`, `curl`, and `GnuPG`. The first three utilities are fairly common, but the last one, `GnuPG`, may be newer to some readers and might need some explaining. `GnuPG`, an acronym for **GNU Privacy Guard**, enhances Ubuntu with a range of cryptographic capabilities such as **encryption, decryption, digital signatures, and key management**.

In our Docker deployment, we need to add Docker’s **GPG public key**, which is a cryptographic key pair that secures communication and maintains data integrity. GPG keys use asymmetrical encryption, which involves the use of two different, but related, keys, known as a **public key** and a **private key**. These keys are generated together as a pair, but they provide different functions. The private key, which remains confidential, is used to generate the digital signatures on the downloaded files. The public key is publicly available and is used to verify digital signatures created by the private key.

Next, we need to add Docker’s repository to our local repository list. When we add the repository to the list, we need to include the Docker certificate. The `docker.gpg` certificate was downloaded by the script from Docker’s site and stored on the local server under `/etc/apt/keyrings/docker.gpg`. When we add the repository to the repository list, we add the key by using the `signed-by` option in the `/etc/apt/sources.list.d/docker.list` file. The full command is shown here:

```
deb [arch=amd64 signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu jammy stable
```

By including the Docker repository in our local `apt` repository list, we gain the ability to install the Docker binaries effortlessly. This process entails using a straightforward `apt-get install` command, which will install the five essential binaries for Docker: `docker-ce`, `docker-ce-cli`, `containerd.io`, `docker-buildx-plugin`, and `docker-compose-plugin`. As previously stated, all these files are signed with Docker’s GPG key. Thanks to the inclusion of Docker’s key on our server, we can be confident that the files are safe and originate from a reliable source.

Once Docker is successfully installed, the next step involves enabling and configuring the Docker daemon to start automatically during system boot using the `systemctl` command. This process follows the standard procedure applied to most system daemons installed on Linux servers.



Rather than go over each line of code in each script, we have included comments in the scripts to help you understand how what each command and step is executing. Where it may help with some topics, we will include some section of code in the chapters for reference.

After installing Docker, let’s get some configuration out of the way. First, you will rarely execute commands as root in the real world, so we need to grant permissions to use Docker to your user.

## Granting Docker permissions

In a default installation, Docker requires root access, so you will need to run all Docker commands as **root**. Rather than using `sudo` with every Docker command, you can add your user account to a new group on the server that provides Docker access without requiring `sudo` for every command.

If you are logged on as a standard user and try to run a Docker command, you will receive an error:

```
Got permission denied while trying to connect to the Docker daemon socket at
unix:///var/run/docker.sock: Get http://%2Fvar%2Frun%2Fdocker.sock/v1.40/
images/json: dial unix /var/run/docker.sock: connect: permission denied
```

To allow your user, or any other user you may want to add, to execute Docker commands, you need to add the users to a new group called `docker` that was created during the installation of Docker. The following is an example command you can use to add the currently logged-on user to the group:

```
sudo usermod -aG docker $USER
```

To add the new members to your account, you can either log off and log back into the Docker host, or activate the group changes using the `newgrp` command:

```
newgrp docker
```

Now, let's test that Docker is working by running the standard `hello-world` image (note that we do not require `sudo` to run the Docker command):

```
docker run hello-world
```

You should see the following output, which verifies that your user has access to Docker:

```
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest: sha256:37a0b92b08d4919615c3ee023f7ddb068d12b8387475d64c622ac30f45c29c51
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
```

This message shows that your installation is working correctly – congratulations!

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the `hello-world` image from Docker Hub (amd64).
3. The Docker daemon created a new container from the image that runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious – you can run an Ubuntu container with the following:

```
$ docker run -it ubuntu bash
```

For more examples and ideas, visit <https://docs.docker.com/get-started/>.

Now that we've granted Docker permission, we can start unlocking the most common Docker commands by learning how to use the Docker CLI.

## Using the Docker CLI

You used the Docker CLI when you ran the `hello-world` container to test your installation. The Docker command is what you will use to interact with the Docker daemon. Using this single executable, you can do the following, and more:

- Start and stop containers
- Pull and push images
- Run a shell in an active container
- Look at container logs
- Create Docker volumes
- Create Docker networks
- Prune old images and volumes

This chapter is not meant to include an exhaustive explanation of every Docker command; instead, we will explain some of the common commands that you will need to use to interact with the Docker daemon and containers.

You can break down Docker commands into two categories: general Docker commands and Docker management commands. The standard Docker commands allow you to manage containers, while management commands allow you to manage Docker options such as managing volumes and networking.

### **docker help**

It is quite common to forget the syntax or options of a command, and Docker acknowledges this. If you ever find yourself in a situation where you can't recall a command, you can always depend on the `docker help` command. It will help you by providing what the command can do and how to use it.

### **docker run**

To run a container, use the `docker run` command with the provided image name. But, before executing a `docker run` command, you should understand the options you can supply when starting a container.

In its simplest form, an example command you can use to run an NGINX web server would be `docker run bitnami/nginx:latest`. This will start a container running NGINX, and it will run in the foreground, showing logs of the application running in the container. Pressing `Ctrl + C` will stop the running container and terminate the NGINX server:

```
nginx 22:52:27.42
```

```
nginx 22:52:27.42 Welcome to the Bitnami nginx container
nginx 22:52:27.43 Subscribe to project updates by watching https://github.com/
bitnami/bitnami-docker-nginx
nginx 22:52:27.43 Submit issues and feature requests at https://github.com/
bitnami/bitnami-docker-nginx/issues
nginx 22:52:27.44
nginx 22:52:27.44 INFO ==> ** Starting NGINX setup **
nginx 22:52:27.49 INFO ==> Validating settings in NGINX_* env vars
nginx 22:52:27.50 INFO ==> Initializing NGINX
nginx 22:52:27.53 INFO ==> ** NGINX setup finished! **

nginx 22:52:27.57 INFO ==> ** Starting NGINX **
```

As you saw, when you used *Ctrl + C* to stop the container, NGINX also stopped. In most cases, you want a container to start and continue to run without being in the foreground, allowing the system to run other tasks while the container also continues to run. To run a container as a background process, you need to add the `-d`, or `--detach` option to your Docker command, which will run your container in detached mode. Now, when you run a detached container, you will only see the container ID, instead of the interactive or attached screen:

```
[root@localhost ~]# docker run -d bitnami/nginx:latest
13bdde13d0027e366a81d9a19a56c736c28feb6d8354b363ee738d2399023f80
[root@localhost ~]#
```

By default, containers will be given a random name once they are started. In our previous detached example, if we list the running containers, we will see that the container has been given the name `silly_keldysh`, as shown in the following output:

CONTAINER ID	IMAGE	NAMES
13bdde13d002	bitnami/nginx:1	

If you do not assign a name to your container, it can quickly get confusing as you start to run multiple containers on a single host. To make management easier, you should always start your container with a name that will make it easier to manage. Docker provides another option with the `run` command: the `--name` option. Building on our previous example, we will name our container `nginx-test`. Our new `docker run` command will be as follows:

```
docker run --name nginx-test -d bitnami/nginx:latest
```

Just like running any detached image, this will return the container ID, but not the name you provided. In order to verify that the container ran with the name `nginx-test`, we can list the containers using the `docker ps` command, which we will explain next.

## docker ps

Often, you will need to retrieve a list of running containers or a list of containers that have been stopped. The Docker CLI has a flag called `ps` that will list all running and stopped containers, by adding the extra flag to the `ps` command. The output will list the containers, including their container ID, image tag, entry command, creation date, status, ports, and container name. The following is an example of containers that are currently running:

CONTAINER ID	IMAGE	COMMAND	CREATED
13bdde13d002	bitnami/nginx:latest	"/opt/bitnami/script..."	Up 4 hours
3302f2728133	registry:2	"/entrypoint.sh /etc..."	Up 3 hours

This is helpful if the container you are looking for is currently running, but what if the container has stopped, or even worse, what if the container failed to start and then stopped? You can view the status of all containers, including previously run containers, by adding the `-a` flag to the `docker ps` command. When you execute `docker ps -a`, you will see the same output from a standard `ps` command, but you will notice that the list may include additional containers.

How can you tell which containers are running versus which ones have stopped? If you look at the `STATUS` field of the list, the running containers will show a running time; for example, `Up xx hours`, or `Up xx days`. However, if the container has been stopped for any reason, the status will show when it stopped; for example, `Exited (0) 10 minutes ago`.

IMAGE	COMMAND	CREATED	STATUS
bitnami/nginx:latest	"/opt/bitnami/script..."	10 minutes ago	Up 10 minutes
bitnami/nginx:latest	"/opt/bitnami/script..."	12 minutes ago	Exited (0) 10 minutes ago

A stopped container does not mean there was an issue with running the image. There are containers that may execute a single task and, once completed, the container may stop gracefully. One way to determine whether an exit was graceful or whether it was due to a failed startup is to look at the exited status code. There are a number of exit codes that you can use to find out why a container has exited.

Exit Code	Description
0	The command was executed successfully without any issues.
1	The command failed due to an unexpected error.
2	The command was unable to find the specified resource or encountered a similar issue.
125	The command failed due to a Docker-related error.
126	The command failed because the Docker binary or script could not be executed.
127	The command failed because the Docker binary or script could not be found.
128+	The command failed due to a specific Docker-related error or exception.

Table 1.3: Docker exit codes

## docker start and stop

You may need to stop a container due to limited system resources, limiting you to running a few containers simultaneously. To stop a running container and free up resources, use the `docker stop` command with the name of the container, or the container ID, you want to stop.

If you need to start that container at a future time for additional testing or development, execute `docker start <name>`, which will start the container with all of the options that it was originally started with, including any networks or volumes that were assigned.

## docker attach

In order to troubleshoot an issue or inspect a log file, it may be necessary to interact with a container. One way to connect to a container that is currently running is by using the `docker attach <container ID/name>` command. When you perform this action, you establish a connection with the active process of the running container. If you attach to a container that is executing a process, it is unlikely that you will see any prompt. In fact, it's likely that you will see a blank screen for a period of time until the container starts producing output that is displayed on the screen.

You should always be cautious when attaching to a container. It's easy to accidentally stop the running process and, in turn, stop the container. Let's use an example of attaching to a web server running NGINX. First, we need to verify that the container is running using `docker ps`:

CONTAINER ID	IMAGE	COMMAND	STATUS
4a77c14a236a	nginx	/docker-entrypoint ..."	Up 33 seconds

Using the `attach` command, we execute `docker attach 4a77c14a236a`.

When you attach to a process, you will only be able to interact with the running process, and the only output you will see is data being sent to standard output. In the case of the NGINX container, the `attach` command has been attached to the NGINX process. To show this, we will leave the attachment and `curl` to the web server from another session. Once we `curl` to the container, we will see logs outputted to the attached console:

```
[root@astra-master manifests]# docker attach 4a77c14a236a
172.17.0.1 - - [15/Oct/2021:23:28:31 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/7.61.1" "-"
172.17.0.1 - - [15/Oct/2021:23:28:33 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/7.61.1" "-"
172.17.0.1 - - [15/Oct/2021:23:28:34 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/7.61.1" "-"
172.17.0.1 - - [15/Oct/2021:23:28:35 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/7.61.1" "-"
172.17.0.1 - - [15/Oct/2021:23:28:36 +0000] "GET / HTTP/1.1" 200 615 "-"
"curl/7.61.1" "-"
```

We mentioned that you need to be careful once you attach to the container. Those who are new to Docker may attach to the NGINX image and assume that nothing is happening on the server or a process appears to be hung so they may decide to break out of the container using the standard *Ctrl + C* keyboard command. This will stop the container and send them back to a Bash prompt, where they may run `docker ps` to look at the running containers:

```
root@localhost:~# docker ps
CONTAINER ID        IMAGE       COMMAND       CREATED      STATUS
root@localhost:~#
```

What happened to the NGINX container? We didn't execute a `docker stop` command, and the container was running until we attached to the container. Why did the container stop after we attached to it?

As we mentioned, when an attachment is made to a container, you are attached to the running process. All keyboard commands will act in the same way as if you were at a physical server that was running NGINX in a regular shell. This means that when the user used *Ctrl + C* to return to a prompt, they stopped the running NGINX process.

If we press *Ctrl + C* to exit the container, we will receive an output that shows that the process has been terminated. The following output shows an example of what happens in our NGINX example:

```
2023/06/27 19:38:02 [notice] 1#1: signal 2 (SIGINT) received, exiting2023/06/27
19:38:02 [notice] 31#31: exiting2023/06/27 19:38:02 [notice] 30#30:
exiting2023/06/27 19:38:02 [notice] 29#29: exiting2023/06/27 19:38:02 [notice]
31#31: exit2023/06/27 19:38:02 [notice] 30#30: exit2023/06/27 19:38:02 [notice]
29#29: exit2023/06/27 19:38:02 [notice] 32#32: exiting2023/06/27 19:38:02
[notice] 32#32: exit2023/06/27 19:38:03 [notice] 1#1: signal 17 (SIGCHLD)
received from 312023/06/27 19:38:03 [notice] 1#1: worker process 29 exited
with code 02023/06/27 19:38:03 [notice] 1#1: worker process 31 exited with
code 02023/06/27 19:38:03 [notice] 1#1: worker process 32 exited with code
02023/06/27 19:38:03 [notice] 1#1: signal 29 (SIGIO) received2023/06/27
19:38:03 [notice] 1#1: signal 17 (SIGCHLD) received from 292023/06/27 19:38:03
[notice] 1#1: signal 17 (SIGCHLD) received from 302023/06/27 19:38:03 [notice]
1#1: worker process 30 exited with code 02023/06/27 19:38:03 [notice] 1#1: exit
```

If a container's running process stops, the container will also stop, and that's why the `docker ps` command does not show a running NGINX container.

To exit an attachment, rather than use *Ctrl + C* to return to a prompt, you should have used *Ctrl + P*, followed by *Ctrl + Q*, which will exit the container without stopping the running process.

There is an alternative to the `attach` command: the `docker exec` command. The `exec` command differs from the `attach` command since you supply the process to execute on the container.

## docker exec

A better option when it comes to interacting with a running container is the `exec` command. Rather than attach to the container, you can use the `docker exec` command to execute a process in the container. You need to supply the container name and the process you want to execute in the image. Of course, the process must be included in the running image – if you do not have the Bash executable in the image, you will receive an error when trying to execute Bash in the container.

We will use an NGINX container as an example again. We will verify that NGINX is running using `docker ps` and then, using the container ID or the name, we execute into the container. The command syntax is `docker exec <options> <container name> <command>`:

```
root@localhost:~# docker exec -it nginx-test bash  
I have no name!@a7c916e7411:/app$
```

The option we included is `-it`, which tells the `exec` to run in an interactive TTY session. Here, the process we want to execute is Bash.

Notice how the prompt name changed from the original user and hostname. The hostname is `localhost`, while the container name is `a7c916e7411`. You may also have noticed that the current working directory changed from `~` to `/app` and that the prompt is not running as a root user, as shown by the `$` prompt.

You can use this session the same way you would a standard `SSH` connection; you are running Bash in the container and since we are not attached to the running process in the container, `Ctrl + C` will not stop any process from running.

To exit an interactive session, you only need to type in `exit`, followed by `Enter`, which will exit the container. If you then run `docker ps`, you will notice that the container is still in a running state.

Next, let's see what we can learn about Docker log files.

## docker logs

The `docker logs` command allows you to retrieve logs from a container using the container name or container ID. You can view the logs from any container that is listed in your `ps` command; it doesn't matter if it's currently running or stopped.

Log files are often the only way to troubleshoot why a container may not be starting up, or why a container is in an exited state. For example, if you attempt to run an image and the image starts and suddenly stops, you may find the answer by looking at the logs for that container.

To look at the logs for a container, you can use the `docker logs <container ID or name>` command.

To view the logs for a container with a container ID of `7967c50b260f`, you would use the following command:

```
docker logs 7967c50b260f
```

This will output the logs from the container to your screen, which may be very long and verbose. Since many logs may contain a lot of information, you can limit the output by supplying the `logs` command with additional options. The following table lists the options available for viewing logs:

Log Options	Description
<code>-f</code>	Follow the log output (can also use <code>--follow</code> ).
<code>--tail xx</code>	Show the log output starting from the end of the file and retrieve <code>xx</code> lines.
<code>--until xxx</code>	Show the log output before the <code>xxx</code> timestamp. <code>xxx</code> can be a timestamp; for example, <code>2020-02-23T18:35:13</code> . <code>xxx</code> can be a relative time; for example, <code>60m</code> .
<code>--since xxx</code>	Show the log output after the <code>xxx</code> timestamp. <code>xxx</code> can be a timestamp; for example, <code>2020-02-23T18:35:13</code> . <code>xxx</code> can be a relative time; for example, <code>60m</code> .

Table 1.4: Log options

Checking log files is a process you will find yourself doing often, and since they can be very lengthy, knowing options like `tail`, `until`, and `since` will help you to find the information in a log quicker.

## docker rm

Once you assign a name to a container, the assigned name cannot be used on a different container unless you remove it using the `docker rm` command. If you had a container running called `nginx-test` that was stopped and you attempted to start another container with the name `nginx-test`, the Docker daemon would return an error, stating that the name is in use:

Conflict. The container name "/nginx-test" is already in use

The original `nginx-test` container is not running, but the daemon knows that the container name was used previously and that it's still in the list of previously run containers.

When you want to reuse a specific name, you must first remove the existing container before launching a new one with the same name. This scenario commonly occurs during container image testing. You may initially start a container but encounter issues with the application or image. In such instances, you would stop the container, resolve the problems with the image or application, and wish to redeploy it using the same name. However, since the previous container with that name still exists in the Docker history, it becomes necessary to remove it before reutilizing the name.



You can also add the `--rm` option to your Docker command to automatically remove the image after it is stopped.

To remove the `nginx-test` container, simply execute `docker rm nginx-test`:

```
root@localhost ~:# docker rm nginx-test
nginx-test
root@localhost ~:#
```

Assuming the container name is correct and it's not running, the only output you will see is the name of the image that you have removed.

We haven't discussed Docker volumes, but when removing a container that has a volume, or volumes, attached, it's a good practice to add the `-v` option to your remove command. Adding the `-v` option to the `docker rm` command will remove any volumes that were attached to the container.

## **docker pull/run**

When running a `pull`, make sure to specify the architecture. `docker pull` and `run` are used to either pull an image or run an image. If you try to run a container that doesn't exist on the Docker host already, it will initiate a `pull` request to get the container and then run it.

When you attempt to `pull` or `run` a container, Docker will download a container that is compatible with the host's architecture. If you want to download a different image that is based on a different architecture, you can add the `--platform` tag to the `build` command. For example, if you are on a system that is running on `arm64` architecture and you want to pull an `x86` image, you would add `linux/arm64` as your platform. When running a `pull`, make sure to specify the architecture:

```
root@localhost ~:# docker pull --platform=linux/amd64 ubuntu:22.04
22.04: Pulling from library/ubuntu6b851dcae6ca: Pull completeDigest:
sha256:6120be6a2b7ce665d0cbddc3ce6eae60fe94637c6a66985312d1f02f63cc0bcdStatus:
Downloaded newer image for ubuntu:22.04WARNING: image with reference ubuntu was
found but does not match the specified platform: wanted linux/amd64, actual:
linux/arm64/v8docker.io/library/ubuntu:22.04
```

Adding `--platform=linux/amd64` is what told Docker to get the right platform. You can use the same parameter for `docker run` to make sure that the right container image platform is used.

## **docker build**

Similar to `pull` and `run`, Docker will attempt to build the image based on the host's architecture: `arm64`. Assuming you are building on an `arm64`-based image system, you can tell Docker to create an `x86` image by using the `buildx` sub-command:

```
root@localhost ~:# docker buildx build --platform linux/amd64 --tag docker.
io/mlbiam/openunison-kubernetes-operator --no-cache -f ./src/main/docker/
Dockerfile .
```

This addition tells Docker to generate the `x86` version, which will run on any `x86`-based hardware.

## **Summary**

In this chapter, you learned how Docker can be used to solve common development issues, including the dreaded "it works on my machine" problem. We also presented an introduction to the most commonly used Docker CLI commands that you will use on a daily basis.

In the next chapter, we will start our Kubernetes journey with an introduction to **KinD**, a utility that provides an easy way to run multi-node Kubernetes test servers on a single workstation.

## Questions

1. A single Docker image can be used on any Docker host, regardless of the architecture used.

- a. True
- b. False

Answer: b



2. What does Docker use to merge multiple image layers into a single filesystem?

- a. Merged filesystem
- b. NTFS filesystem
- c. EXT4 filesystem
- d. Union filesystem

Answer: d

3. Kubernetes is only compatible with the Docker runtime engine.

- a. True
- b. False

Answer: b

4. When you edit a container's filesystem interactively, what layer are the changes written to?

- a. OS layer
- b. Bottom-most layer
- c. Container layer
- d. Ephemeral layer

Answer: c

5. Assuming the image contains the required binaries, what Docker command allows you to gain access to a running container's bash prompt?

- a. docker shell -it <container> /bin/bash
- b. docker run -it <container> /bin/bash
- c. docker exec -it <container> /bin/bash
- d. docker spawn -it <container> /bin/bash

Answer: c

6. If you start a container with a simple `run` command, without any flags, and the container is stopped, the Docker daemon will delete all traces of the container.
  - a. True
  - b. False

Answer: b

7. What command will show you a list of all containers, including any stopped containers?
  - a. `docker ps -all`
  - b. `docker ps -a`
  - c. `docker ps -list`
  - d. `docker list all`

Answer: b

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 2

## Deploying Kubernetes Using KinD

Like many IT professionals, having a Kubernetes cluster on our laptops is highly beneficial for showcasing and testing products. In certain cases, you may require running a cluster with multiple nodes or clusters for intricate demonstrations or testing, such as a multi-cluster service mesh. These scenarios require multiple servers to create the required clusters, which, in turn, call for substantial RAM and a **hypervisor** to run virtual machines.

To do full testing on a multiple-cluster scenario, you would need to create multiple nodes for each cluster. If you created the clusters using virtual machines, you would need to have enough resources to run multiple virtual machines. Each of the machines would have an **overhead**, including disk space, memory, and CPU utilization.

Imagine if it were possible to establish a cluster solely using containers. By utilizing containers instead of full virtual machines, you gain the advantage of running additional nodes due to the reduced system requirements. This approach enables you to quickly create and delete clusters within minutes using a single command. Additionally, you can employ scripts to facilitate cluster creation and even run multiple clusters on a single host.

Using containers to run a Kubernetes cluster provides you with an environment that would be difficult for most people to deploy using virtual machines or physical hardware, due to resource constraints. Lucky for us, there is a tool to accomplish this called **KinD (Kubernetes in Docker)**, which allows us to run Kubernetes cluster(s) on a single machine. KinD is a tool that provides a simple, quick, and easy method to deploy a development Kubernetes cluster. It is smaller when compared to other alternatives like Minikube, and even K3s, making it ideal for most users to run on their own systems.

We will use KinD to deploy a multi-node cluster that you will use in future chapters to test and deploy components, such as Ingress controllers, authentication, RBAC (Role-Based Access Control), security policies, and more.

In this chapter, we will cover the following main topics:

- Introducing Kubernetes components and objects
- Using development clusters
- Installing KinD
- Creating a KinD cluster
- Reviewing your KinD cluster
- Adding a custom load balancer for Ingress

Let's get started!

## Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, although 8 GB is recommended
- Scripts from the `chapter2` folder from the GitHub repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

We consider it essential to highlight that this chapter will mention various Kubernetes objects, some of which may lack extensive context. However, in *Chapter 3, Kubernetes Bootcamp*, we will dive into Kubernetes objects in depth, providing numerous example commands to enhance your understanding. To ensure a practical learning experience, we recommend having a cluster while reading the bootcamp chapter.

Most of the basic Kubernetes topics covered in this chapter will be discussed in future chapters, so if some topics become a bit foggy after you've read this chapter, never fear! They will be discussed in detail in later chapters.

## Introducing Kubernetes components and objects

Since this chapter will discuss various common Kubernetes objects and components, we've included a table with brief definitions for each term. This will provide you with the necessary context and help ensure you understand the terminology as you read through the chapter.

In *Chapter 3, Kubernetes Bootcamp*, we will go over the components of Kubernetes and the basic set of objects that are included in a cluster. Since we will have to use some basic objects in this module, we have provided some common Kubernetes components and resources in the table below.

Component	Description
Control Plane	API-Server: Frontend of the control plane that accepts requests from clients. kube-scheduler: Assigns workloads to nodes. etcd: Database that contains all cluster data. kube-controller-manager: Watches for node health, pod replicas, endpoints, service accounts, and tokens.
Node	kubelet: The agent that runs a pod based on instructions from the control plane. kube-proxy: Creates and deletes network rules for pod communication. Container runtime: Component responsible for running a container.
Object	Description
Container	A single immutable image that contains everything needed to run an application.
Pod	The smallest object that Kubernetes can control. A pod holds a container, or multiple containers. All containers in a pod are scheduled on the same server in a shared context (that is, each container in a pod can address other pods using 127.0.0.1).
Deployment	Used to deploy an application to a cluster based on a desired state, including the number of pods and rolling update configuration.
Storage Class	Defines storage providers and presents them to the cluster.
Persistent Volume (PV)	Provides a storage target that can be claimed by a Persistent Volume Request.
Persistent Volume Claim (PVC)	Connects (claims) a Persistent Volume so that it can be used inside a pod.
Container Network Interface (CNI)	Provides the network connection for pods. Common CNI examples include Flannel and Calico.
Container Storage Interface (CSI)	Provides the connection between pods and storage systems.

Figure 2.1: Kubernetes components and objects

While these are only a few of the objects that are available in a Kubernetes cluster, they are the objects we will discuss in this chapter. Knowing what each resource is and having basic knowledge of its functionality will help you to understand this chapter.

## Interacting with a cluster

To interact with a cluster, you use the `kubectl` executable. We will go over `kubectl` in *Chapter 3, Kubernetes Bootcamp*, but since we will be using a few commands in this chapter, we wanted to provide the basic commands we will use in a table with an explanation of what the options provide:

Kubectl command	Description
<code>kubectl get &lt;object&gt;</code>	Retrieves a list of the requested object.  Example: <code>kubectl get nodes</code> .
<code>kubectl create -f &lt;manifest-name&gt;</code>	Creates the objects in the <code>include</code> manifest that is provided. <code>create</code> can only create the initial objects; it cannot update the objects.
<code>kubectl apply -f &lt;manifest-name&gt;</code>	Deploys the objects in the <code>include</code> manifest that is provided. Unlike the <code>create</code> option, the <code>apply</code> command can update objects, as well as create objects.
<code>kubectl patch &lt;object-type&gt; &lt;object-name&gt; -p {patching options}</code>	Patches the supplied <code>object-type</code> with the options provided.

Figure 2.2: Basic `kubectl` commands

In this chapter, you will use these basic commands to deploy parts of the cluster that we will use throughout this book.

Next, we will introduce the concept of development clusters and then focus on one of the most popular tools used to create the development clusters, KinD.

## Using development clusters

Over time, several solutions have been developed to facilitate the installation of development Kubernetes clusters, enabling administrators and developers to conduct testing on local systems. While these tools have proven effective for basic Kubernetes testing, they often possess certain limitations that render them suboptimal for more advanced scenarios.

Some of the most common solutions available are as follows:

- Docker Desktop
- K3s
- KinD
- kubeadm
- minikube
- Rancher Desktop

Each solution has benefits, limitations, and use cases. Some solutions limit you to a single node that runs both the control plane and worker nodes. Others offer multi-node support but require additional resources to create multiple virtual machines. Depending on your development or testing requirements, these solutions may not meet your needs completely.

To truly get into Kubernetes, you need to have a cluster that has at least a control plane and a single worker node. You may want to test scenarios where you drop a worker node suddenly to see how a workload reacts; in this case, you would need to create a cluster that has a control plane node and three worker nodes. To create these various cluster types, we can use a project from the Kubernetes Special Interest Group (SIG), called **KinD**.

**KinD (Kubernetes in Docker)** offers the capability to create multiple clusters on a single host, where each cluster can have multiple control planes and worker nodes. This feature facilitates advanced testing scenarios that would have otherwise required additional resource allocation, using alternative solutions. The community response to KinD has been highly positive, as shown by its active GitHub community at <https://github.com/kubernetes-sigs/kind> and the availability of a dedicated Slack channel (#kind).



While KinD is a great tool to create development clusters, do not use KinD as a production cluster or expose a KinD cluster to the Internet. Although KinD clusters offer most of the same features you would want in a production cluster, it has not been designed for production environments.

## Why did we select KinD for this book?

When we started this book, our objective was to combine theoretical knowledge with practical hands-on experience. KinD emerged as an important tool in achieving this goal by enabling us to include scripts to quickly set up and tear down clusters. While alternative solutions may offer comparable functionality, KinD stands out by its ability to establish multi-node clusters within a matter of minutes. We intended to provide a cluster that has both a control plane and worker nodes to emulate a more “real-world” cluster environment. In order to reduce hardware demands and streamline Ingress configuration, we have chosen to limit most of the exercise scenarios in this book to a single control plane node and a single worker node.

Some of you may be asking yourselves why we didn't use kubeadm or some other tool to deploy a cluster that has multiple nodes for both the control plane and worker nodes. As we have said, while KinD is not meant to be used in production, it requires fewer resources to simulate a multi-node cluster, allowing most of the readers to work on a cluster that will act like a standard enterprise-ready Kubernetes cluster.

A multi-node cluster can be created in a few minutes, and once testing has been completed, clusters can be torn down in a few seconds. The ability to spin up and down clusters makes KinD the perfect platform for our exercises. KinD's requirements are simple: you only need a running Docker daemon to create a cluster. This means that it is compatible with most operating systems, including the following:

- Linux
- macOS running Docker Desktop
- Windows running Docker Desktop
- Windows running WSL2

At the time of writing, KinD does not offer official support for Chrome OS. There are a number of posts in the KinD Git repository on the required steps to make KinD work on a system running Chrome OS; however, it's not officially supported by the team.

While KinD supports most operating systems, we have selected **Ubuntu 22.04** as our host system. Some of the exercises in this book require files to be in specific directories and commands; selecting a single Linux version helps us make sure the exercises work as designed. If you do not have access to a Ubuntu server at home, you can create a compute instance in a cloud provider such as **Google Cloud Platform (GCP)**. Google offers \$300 in credit, which is more than enough to run a single Ubuntu server for a few weeks. You can view GCP's free options at <https://cloud.google.com/free/>.

Finally, the scripts used to deploy the exercises are all pinned to specific versions of KinD and other dependencies. Kubernetes and the cloud-native world move very quickly, and we can't guarantee that everything will work as expected with the latest versions of the systems that exist when you read our book.

Now, let's explain how KinD works and what a basic KinD Kubernetes cluster looks like. Before we move on to creating the cluster, we will use it for the book exercises.

## Working with a basic KinD Kubernetes cluster

From a broader perspective, a KinD cluster can be seen as comprising a single Docker container, responsible for running both a control plane node and a worker node, creating a Kubernetes cluster. To ensure a straightforward and resilient deployment, KinD packages all Kubernetes objects into a unified image, referred to as a node image. This node image includes all the necessary Kubernetes components required to create either a single-node or multi-node cluster(s).

To show what's running in a KinD container, we can utilize Docker to execute commands within a control plane node container and examine the process list. Within the process list, you will observe the standard Kubernetes components that are active on the control plane nodes. If we were to execute the following command: `docker exec cluster01-worker ps -ef`.

```
TIME CMD
00:00:00 /sbin/init
00:00:00 /lib/systemd/systemd-journald
00:00:17 /usr/local/bin/containerd
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 2079ca3d203
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 9ee0fe46c62
00:00:00 /pause
00:00:00 /pause
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 476635887b0
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 483ff964102
00:00:00 /pause
00:00:00 /pause
00:00:21 kube-apiserver --advertise-address=172.18.0.4 --allow-privileged=true --
00:00:06 kube-controller-manager --allocate-node-cidrs=true --authentication-kube
00:00:06 etcd --advertise-client-urls=https://172.18.0.4:2379 --cert-file=/etc/ku
00:00:03 kube-scheduler --authentication-kubeconfig=/etc/kubernetes/scheduler.con
00:00:03 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kubele
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id f2d236f617b
00:00:00 /pause
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id e15d3f05f78
00:00:00 /pause
00:00:00 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf --hos
00:00:00 /bin/kindnetd
00:00:00 bash
00:00:00 ps -cf
```

Figure 2.3: Host process list showing control plane components

If you were to exec into a worker node to check the components, you would see all the standard worker node components:

```
TIME CMD
00:00:00 /sbin/init
00:00:00 /lib/systemd/systemd-journald
00:00:08 /usr/local/bin/containerd
00:00:12 /usr/bin/kubelet --bootstrap-kubeconfig=/etc/kubernetes/bootstrap-kub
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 0b4d6d0c
00:00:00 /usr/local/bin/containerd-shim-runc-v2 -namespace k8s.io -id 48699dfc
00:00:00 /pause
00:00:00 /pause
00:00:00 /usr/local/bin/kube-proxy --config=/var/lib/kube-proxy/config.conf --
00:00:00 /bin/kindnetd
00:00:00 bash
00:00:00 ps -ef
```

Figure 2.4: Host process list showing worker components

We will cover the standard Kubernetes components in *Chapter 3, Kubernetes Bootcamp*, including `kube-apiserver`, `kubelets`, `kube-proxy`, `kube-scheduler`, and `kube-controller-manager`.

In addition to standard Kubernetes components, both KinD nodes (the control plane node and worker node) have an additional component that is not part of most standard Kubernetes installations, referred to as **Kindnet**. Kindnet is a **Container Network Interface (CNI)** solution that is included in a default KinD deployment and provides networking to a Kubernetes cluster.

The Kubernetes CNI is a specification that allows Kubernetes to utilize a large list of network software solutions, including **Calico**, **Flannel**, **Cilium**, **Kindnet**, and more.

Although Kindnet serves as the default CNI, it is possible to deactivate it and opt for an alternative, such as Calico, which we will utilize for our KinD cluster. While Kindnet would work for most tasks we need to run, it isn't a CNI that you will see in the real world running a Kubernetes cluster. Since this book is meant to help you along your Kubernetes enterprise journey, we wanted to replace the CNI with a more commonly used CNI like Calico.

Now that you have discussed each of the nodes and the Kubernetes components, let's take a look at what's included with a base KinD cluster. To show the complete cluster and all the components that are running, we can run the `kubectl get pods --all` command. This will list all the running components on the cluster, including the base components, which we will discuss in *Chapter 3, Kubernetes Bootcamp*. In addition to the base cluster components, you may notice a running pod in a namespace called `local-path-storage`, along with a pod named `local-path-provisioner`. This pod runs one of the add-ons included with KinD, providing the cluster with the ability to auto-provision `PersistentVolumeClaims`:

NAMESPACE	NAME	READY	STATUS
kube-system	coredns-558bd4d5db-qbfgl	1/1	Running
kube-system	coredns-558bd4d5db-tdl7g	1/1	Running
kube-system	etcd-temp-control-plane	1/1	Running
kube-system	kindnet-hjjr6	1/1	Running
kube-system	kindnet-jc4p8	1/1	Running
kube-system	kube-apiserver-temp-control-plane	1/1	Running
kube-system	kube-controller-manager-temp-control-plane	1/1	Running
kube-system	kube-proxy-4hf6n	1/1	Running
kube-system	kube-proxy-x7lp7	1/1	Running
kube-system	kube-scheduler-temp-control-plane	1/1	Running
local-path-storage	local-path-provisioner-547f784dff-wthkp	1/1	Running

Figure 2.5: `kubectl get pods` showing `local-path-provisioner`

Each development cluster option typically provides similar functionalities essential for testing deployments. These options typically include a Kubernetes control plane, worker nodes, and a default **Container Networking Interface (CNI)** for networking requirements. While most offerings meet these fundamental needs, some go beyond and offer additional capabilities. As your Kubernetes workloads progress, you may find the need for supplementary add-ons like the `local-path-provisioner`. In this book, we heavily depend on this component for various exercises, as it plays a pivotal role in deploying many of the examples featured throughout the book. Without it, completing the exercises would become considerably more challenging.

Why should the use of persistent volumes in your development cluster be significant? It's all about knowledge that you will run into when using most enterprise Kubernetes clusters. As Kubernetes matures, numerous organizations have transitioned stateful workloads to containers, requiring persistent storage for their data. Being equipped with the capability to interact with storage resources within a KinD cluster offers an opportunity to acquire knowledge about working with storage, all accomplished without the need for additional resources.

The local provisioner is great for development and testing, but it should not be used in a production environment. Most production clusters running Kubernetes will provide persistent storage to developers. Usually, the storage will be backed by storage systems based on block storage, **S3 (Simple Storage Service)**, or **NFS (Network File System)**.

Aside from NFS, most home labs rarely have the resources to run a full-featured storage system. **local-path-provisioner** removes this limitation from users by providing all the functions to your KinD cluster that an expensive storage solution would provide using local disk resources.

In *Chapter 3, Kubernetes Bootcamp*, we will discuss a few API objects that are part of Kubernetes storage. We will discuss the **CSIdrivers**, **CSInodes**, and **StorageClass** objects. These objects are used by the cluster to provide access to the backend storage system. Once installed and configured, pods consume the storage using the **PersistentVolumes** and **PersistentVolumeClaims** objects. Storage objects are important to understand, but when they were first released, they were difficult for most people to test, since they weren't included in most Kubernetes development offerings.

KinD recognized this limitation and chose to bundle a project from Rancher Labs, now part of SUSE, called **local-path-provisioner**, which is built upon the Kubernetes local persistent volumes framework, initially introduced in **Kubernetes 1.10**.

You may be wondering why anyone would need an add-on, since Kubernetes has native support for local host persistent volumes. While support may have been added for local persistent storage, Kubernetes has not added auto-provisioning capabilities. While the **CNCF (Cloud Native Computing Foundation)** does offer an auto-provisioner, it must be installed and configured as a separate Kubernetes component. KinD's provisioner removes this configuration, so you can use persistent volumes easily on development clusters. Rancher's project provides the following to KinD:

- Auto-creation of **PersistentVolumes** when a **PersistentVolumeClaim** request is created.
- A default **StorageClass** named standard.

When the auto-provisioner sees a **PersistentVolumeClaim** (PVC) request hit the API server, a **PersistentVolume** will be created, and the pod's PVC will be bound to the newly created **PersistentVolume**. The PVC can then be used by a pod that requires persistent storage.

The **local-path-provisioner** adds a feature to KinD clusters that greatly expands the potential test scenarios that you can run. Without the ability to auto-provision persistent disks, it would be a challenge to test deployments that require persistent disks.

With the help of Rancher, KinD provides you with a solution so that you can experiment with dynamic volumes, storage classes, and other storage tests that would otherwise be impossible to run outside of an expensive home lab or a data center.

We will use the provisioner in multiple chapters to provide volumes to different deployments. Knowing how to use persistent storage in a Kubernetes cluster is a great skill to have, and in future chapters, you will see the provisioner in action.

Now, let's move on to explain the KinD node image, which is used to deploy both the control plane and the worker nodes.

## Understanding the node image

The node image is what gives KinD the magic to run Kubernetes inside a Docker container. This is an impressive accomplishment, since Docker relies on a `systemd` running system and other components that are not included in most container images.

KinD starts with a base image, which is an image the team has developed that contains everything required for Docker, Kubernetes, and `systemd`. Since the node image is based on a base Ubuntu image, the team removes services that are not required and configures `systemd` for Docker.

If you want to know the details of how the base image is created, you can look at the Docker file in the KinD team's GitHub repository at <https://github.com/kubernetes-sigs/kind/blob/main/images/base/Dockerfile>.

## KinD and Docker networking

When employing KinD, which relies on Docker or Red Hat's **Podman** as the container engine to run cluster nodes, it's important to note that the clusters have the same network constraints typically associated with standard Docker containers. While these limitations don't hinder testing the KinD Kubernetes cluster from the local host, they may introduce complications when attempting to test containers from other machines on your network.



Podman is outside of the scope of this book; it is mentioned as an alternative that KinD now supports. At a high level, it's an open source offering from Red Hat that is meant to replace Docker as a runtime engine. It offers advantages over Docker for many Enterprise use cases, such as enhanced security, not requiring a system daemon, and more. While it has advantages, it can also add complexity for people who are new to the container world

When you install KinD on a Docker host, a new Docker bridge network will be created, called `kind`. This network configuration was introduced in **KinD v0.8.0**, which resolved multiple issues from previous versions that used the default Docker bridge network. Most users will not notice this change, but it's important to know this; as you start to create more advanced KinD clusters with additional containers, you may need to run on the same network as KinD. If you have the requirement to run additional containers on the KinD network, you will need to add `--net=kind` to your `docker run` command.

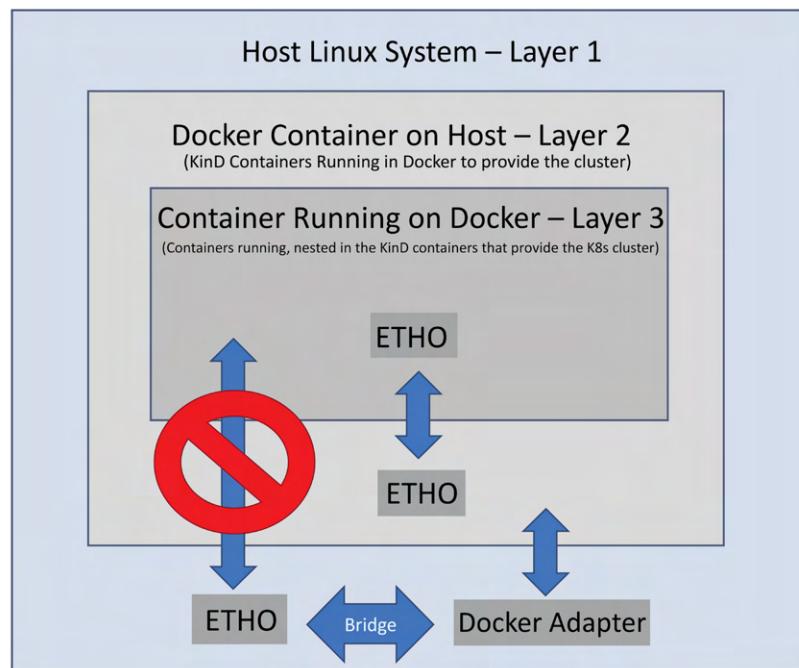
Along with the Docker networking considerations, we must consider the Kubernetes CNI as well. KinD supports multiple different CNIs, including Kindnet, Calico, Cilium, and others. Officially, Kindnet is the only CNI they will support, but you do have the option to disable the default Kindnet installation, which will create a cluster without a CNI installed. After the cluster has been deployed, you need to deploy a CNI such as Calico. Since many Kubernetes installations for both small development clusters and enterprise clusters use Tigera's Calico for the CNI, we have elected to use it as our CNI for the exercises in this book.

## Keeping track of the nesting dolls

The deployment of a solution like KinD, which involves a container-in-a-container approach, can become perplexing. We compare this to the concept of Russian nesting dolls, where one doll fits inside another, and so on. As you use KinD for your own cluster, it's possible to lose track of the communication paths between your host, Docker, and the Kubernetes nodes. To maintain clarity and sanity, it is crucial to have a thorough understanding of the location of each container and how you can interact with them.

*Figure 2.6* shows the three tiers and the network flow for a KinD cluster. It is crucial to recognize that each tier can solely interact with the layer immediately above it so the KinD container within the third layer can solely communicate with the Docker image running within the second layer, while the Docker image can only access the Linux host operating in the first layer. If you wish to establish direct communication between the host and a container operational within your KinD cluster, you will be required to traverse through the Docker layer before reaching the Kubernetes container in the third layer.

This is important to understand so that you can use KinD effectively as a testing environment:



*Figure 2.6: KinD network flow*

Suppose you intend to deploy a web server to your Kubernetes cluster as an example. After successfully deploying an Ingress controller within the KinD cluster, you want to test the website using Chrome on your Docker host or another workstation on the network. However, when you attempt to access the host on port 80, the browser fails to establish a connection. Why does this issue arise?

The reason behind this failure is that the web server's pod operates at layer 3 and cannot directly receive traffic from the host or network machines. To access the web server from your host, you must forward the traffic from the Docker layer to the KinD layer. Specifically, you need to enable port forwarding for port 80 and port 443. When a container is initiated with port specifications, the Docker daemon assumes the responsibility of routing incoming traffic from the host to the running Docker container.

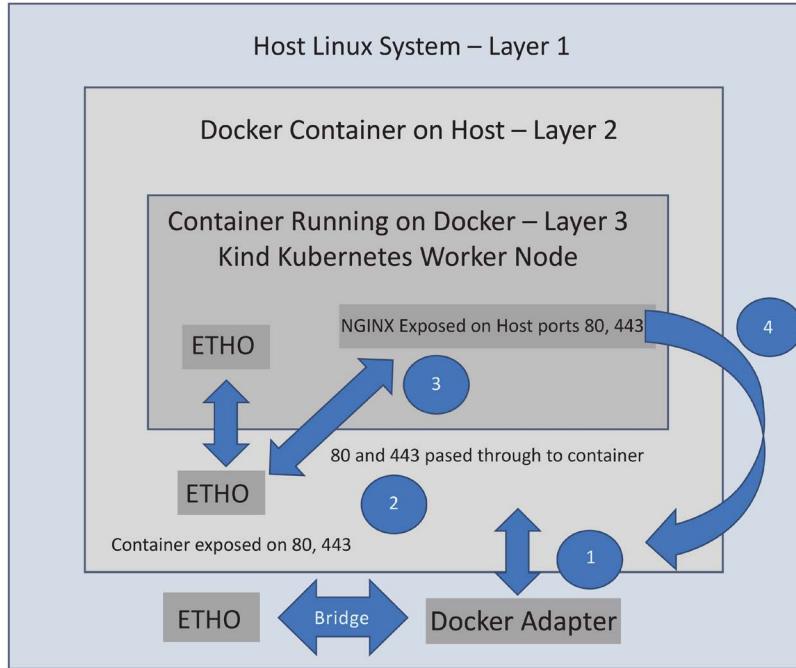


Figure 2.7: Host communicates with KinD via an Ingress controller

With ports 80 and 443 exposed on the Docker container, the Docker daemon will now accept incoming requests for 80 and 443, allowing the NGINX Ingress controller to receive the traffic. This works because we have exposed ports 80 and 443 in two places, first on the Docker layer and then on the Kubernetes layer by running our NGINX controller on the host, using ports 80 and 443.

Now, let's look at the traffic flow for this example.

On the host, you make a request for a web server that has an Ingress rule in your Kubernetes cluster:

1. The request looks at the IP address that was requested (in this case, the local IP address) and the traffic is sent to the Docker container running on the host.
2. The NGINX web server on the Docker container running our Kubernetes node listens on the IP address for ports 80 and 443, so the request is accepted and sent to the running container.
3. The NGINX pod in your Kubernetes cluster has been configured to use the host ports 80 and 443, so the traffic is forwarded to the pod.
4. The user receives the requested web page from the web server via the NGINX Ingress controller.

This is a little confusing, but the more you use KinD and interact with it, the easier it becomes.

In order to utilize a KinD cluster to meet your development needs, it's important to have an understanding of how KinD operates. So far, you have acquired knowledge about the node image and its role in cluster creation. You have also familiarized yourself with the flow of network traffic between the Docker host and the containers that run the cluster within KinD. With this foundational knowledge, we will now proceed to the first step in creating our Kubernetes cluster, installing KinD.

## Installing KinD

At the time of writing, the current version of KinD is `0.22.0`, supporting Kubernetes clusters up to `1.30.x`.

The files required to deploy KinD and all of the components for the cluster that we will use for the chapters are located in the repository, under the `chapter2` folder. The script, `create-cluster.sh`, located in the root of the `chapter2` directory, will execute all of the steps discussed in the remainder of the chapter. You do not need to execute the commands as you read the chapter; you are welcome to follow the steps, but before executing the install script in the repo, you must delete any KinD clusters that may have been deployed.

The deployment script contains in-line remarks to explain each step; however, we will explain each step of the installation process in the next section.

### Installing KinD – prerequisites

There are multiple methods available for installing KinD, but the simplest and fastest approach to begin building KinD clusters is to download the KinD binary along with the standard Kubernetes `kubectl` executable, which enables interaction with the cluster.

### Installing kubectl

Since KinD is a single executable, it does not install `kubectl`. If you do not have `kubectl` installed and you use an Ubuntu 22.04 system, you can install it by running `sudo snap install`, or you can download it directly from Google.

To install `kubectl` using `snap`, you only need to run a single command:

```
sudo snap install kubectl --classic
```

To install `kubectl` from Google directly, you need to download the binary, give it the execute permission, and move it to a location in your system's path. This can be completed using the steps outlined below:

```
curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

Looking at the `curl` command above, you can see that the initial URL is used to find the current release, which at the time of writing was `v1.30.0`. Using the value returned from the `curl` command, we download that release from Google storage.

Now that you have `kubectl`, we can move on to downloading the KinD executable so that we can start to create clusters.

## Installing the KinD binary

Now that we have `kubectl`, we need to download the KinD binary, which is a single executable that we use to create and delete clusters. The binary for KinD v0.22.0 can be downloaded directly using the following URL: <https://github.com/kubernetes-sigs/kind/releases/download/v0.22.0/kind-linux-amd64>. The `create-cluster.sh` script will download the binary, rename it `kind`, mark it as executable, and then move it to `/usr/bin`. To manually download KinD and move it to `/usr/bin`, as the script does for you, you would execute the commands below:

```
curl -Lo ./kind https://github.com/kubernetes-sigs/kind/releases/download/v0.22.0/kind-linux-amd64  
chmod +x ./kind  
sudo mv ./kind /usr/bin
```

The KinD executable provides all of the options you need to maintain a cluster's life cycle. Of course, the KinD executable can create and delete clusters, but it also provides the following capabilities:

- Can create custom a build base and node images
- Can export `kubeconfig` or log files
- Can retrieve clusters, nodes, or `kubeconfig` files
- Can load images into nodes

With the KinD binary successfully installed, you are now on the verge of creating your first KinD cluster.

Since we need a few other executables for some of the exercises in the book, the script also downloads Helm and `jq`. To manually download these utilities, you would execute the commands below:

```
curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3  
chmod 700 get_helm.sh  
../get_helm.sh  
sudo snap install jq --classic
```

If you are new to these tools, Helm is a Kubernetes package manager designed to streamline the deployment and administration of applications and services. It simplifies the process of creating, installing, and managing applications within a cluster. Alternatively, `jq` allows you to extract, filter, transform, and format JSON data sourced from files, command outputs, and APIs. It provides a set of functionalities to work with JSON data, enabling streamlined data manipulation and analysis.

Now that we have the required prerequisites, we can move on to creating clusters. However, before we create our first cluster, it is important to understand the various creation options offered by KinD. Knowing the options will ensure a smooth cluster creation process.

## Creating a KinD cluster

The KinD utility offers the flexibility to create both single-node clusters and more intricate setups with multiple control plane nodes and worker nodes. In this section, we will dive into the various options provided by the KinD executable. By the conclusion of this chapter, you will have a fully operational two-node cluster comprising a control plane node and a worker node.



### Note

Kubernetes cluster concepts, including the control plane and worker nodes will be covered in detail in the next chapter, *Chapter 3: Kubernetes Bootcamp*.

For the exercises covered in this book, we will be setting up a multi-node cluster. The simple cluster configuration provided in the next section serves as an introductory example and should not be employed for the exercises presented in the book.

## Creating a simple cluster

We will create a cluster later in the chapter, but before we do that, let's explain how we can use KinD to create different cluster types.

To create a simple cluster that runs the control plane and a worker node in a single container, you only need to execute the KinD executable with the `create cluster` option.

By executing this command, a cluster named `kind` will be created, encompassing all the necessary Kubernetes components within a single Docker container. The Docker container itself will be assigned the name `kind-control-plane`. If you prefer to assign a custom cluster name instead of using the default name, you can include the `--name <cluster name>` option within the `create cluster` command, for example, `kind create cluster --name custom-cluster`:

```
Creating cluster "kind" ...
✓ Ensuring node image (kindest/node:v1.30.0) 
✓ Preparing nodes 
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing CNI 
✓ Installing StorageClass 
Set kubectl context to "kind-kind"
You can now use your cluster with:
```

```
kubectl cluster-info --context kind-kind
```

Have a question, bug, or feature request? Let us know! <https://kind.sigs.k8s.io/#community> ☺

The `create` command will create the cluster and modify the `kubectl config` file. KinD will add the new cluster to your current `kubectl config` file, and it will set the new cluster as the default context. If you are new to Kubernetes and the concept of context, it is the configuration that will be used to access a cluster and namespace with a set of credentials.

Once the cluster has been deployed, you can verify that the cluster was created successfully by listing the nodes using the `kubectl get nodes` command. The command will return the running nodes in the cluster, which, for a basic KinD cluster, is a single node:

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	Ready	control-plane,master	32m	v1.30.0

The main point of deploying this single-node cluster was to show you how quickly KinD can create a cluster that you can use for testing. For our exercises, we want to split up the control plane and worker node, so we can delete this cluster using the steps in the next section.

## Deleting a cluster

When you no longer need the cluster, you can delete it using the KinD `delete cluster` command. The `delete` command will quickly delete the cluster, including any entries related to the KinD cluster in your `kubeconfig` file.

If you execute the `delete` command without providing a cluster name, it will only attempt to delete a cluster called `kind`. In our previous example, we did not provide a cluster name when we created the cluster, so the default name of `kind` was used. If you did name the cluster when you created it, the `delete` command would require the `--name` option to delete the correct cluster. For example, if we created a cluster named `cluster01`, we would need to execute `kind delete cluster --name cluster01`, to delete it.

While a quick, single-node cluster, is useful for many use cases, you may want to create a multi-node cluster for various testing scenarios. Creating a more complex cluster, with multiple nodes, requires that you create a config file.

## Creating a cluster config file

When creating a multi-node cluster, such as a two-node cluster with custom options, we need to create a cluster config file. The config file is a YAML file and the format should look familiar. Setting values in this file allows you to customize the KinD cluster, including the number of nodes, API options, and more. The config file we'll use to create the cluster for the book is shown here – it is included in this book's repository at `/chapter2/cluster01-kind.yaml`:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
runtimeConfig:
  "authentication.k8s.io/v1beta1": "true"
  "admissionregistration.k8s.io/v1beta1": true
featureGates:
```

```

    "ValidatingAdmissionPolicy": true
networking:
  apiServerAddress: "0.0.0.0"
  disableDefaultCNI: true
  apiServerPort: 6443
  podSubnet: "10.240.0.0/16"
  serviceSubnet: "10.96.0.0/16"
nodes:
  - role: control-plane
    extraPortMappings:
      - containerPort: 2379
        hostPort: 2379
    extraMounts:
      - hostPath: /sys/kernel/security
        containerPath: /sys/kernel/security
  - role: worker
    extraPortMappings:
      - containerPort: 80
        hostPort: 80
      - containerPort: 443
        hostPort: 443
      - containerPort: 2222
        hostPort: 2222
    extraMounts:
      - hostPath: /sys/kernel/security
        containerPath: /sys/kernel/security

```

Details about each of the custom options in the file are provided in the following table:

Config Options	Option Details
apiServerAddress	This configuration option tells the installation what IP address the API server will listen on. By default, it will use 127.0.0.1, but since we plan to use the cluster from other networked machines, we have selected to listen on all IP addresses.
disableDefaultCNI	This setting is used to enable or disable the Kindnet installation. The default value is false, but since we want to use Calico as our CNI, we need to set it to true.
podSubnet	Sets the CIDR range that will be used by pods.
serviceSubnet	Sets the CIDR range that will be used by services.
Nodes	This section is where you define the nodes for the cluster. For our cluster, we will create a single control plane node and a single worker node.

- role: control-plane	The role section allows you to set options for nodes. The first role section is for the <code>control-plane</code> .
- role: worker	This is the second node section, which allows you to configure options that the worker nodes will use. Since we will deploy an Ingress controller, we have also added additional ports that will be used by the NGINX pod.
<code>extraPortMappings</code>	To expose ports to your KinD nodes, you need to add them to the <code>extraPortMappings</code> section of the configuration. Each mapping has two values, the container port and the host port. The host port is the port you would use to target the cluster, while the container port is the port that the container listens on.

*Table 2.1: KinD configuration options*

Understanding the available options allows you to customize a KinD cluster according to your specific requirements. This includes incorporating advanced components like ingress controllers, which facilitate efficient routing of external traffic to services within the cluster. It also provides the ability to deploy multiple nodes within the cluster, allowing you to conduct testing and failure/recovery procedures, ensuring the resilience and stability of your cluster. By leveraging these capabilities, you can fine-tune your cluster to meet the exact demands of your applications and infrastructure.

Now that you know how to create a simple all-in-one container to run a cluster and create a multi-node cluster using a config file, let's discuss a more complex cluster example.

## Multi-node cluster configuration

If you only wanted a multi-node cluster without any extra options, you could create a simple configuration file that lists the number and node types you want in the cluster. The following example config file will create a cluster with three control plane nodes and three worker nodes:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: control-plane
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
```

Incorporating multiple control plane servers introduces added complexities, since our `kubectl` config file can only target a single host or IP. To make this solution work across all three control plane nodes, it is necessary to deploy a load balancer in front of our cluster. This load balancer will facilitate the distribution of control plane traffic among the control plane servers. It's important to note that, by default, HAProxy will not load balance any traffic between the worker nodes. To load balance traffic to worker nodes is more complex, and we will discuss it later in the chapter.

KinD has considered this, and if you deploy multiple control plane nodes, the installation will create an additional container running a HAProxy load balancer. During the creation of a multi-node cluster, you will see a few additional lines regarding configuring an extra load balancer, joining additional control-plane nodes and extra worker nodes – as shown in the example below, we created a cluster using the example cluster config, with three control plane and worker nodes:

```
Creating cluster "multinode" ...
✓ Ensuring node image (kindest/node:v1.30.0) 
✓ Preparing nodes 
✓ Configuring the external load balancer 
✓ Writing configuration 
✓ Starting control-plane 
✓ Installing StorageClass 
✓ Joining more control-plane nodes 
✓ Joining worker nodes 

Set kubectl context to "kind-multinode"
You can now use your cluster with:

kubectl cluster-info --context kind-multinode

Thanks for using kind! ☺
```

In the output above, you will see a line that says `Configuring the external load balancer`. This step deploys a load balancer to route the incoming traffic to the API server to the three control plane nodes.

If we look at the running containers from a multi-node control plane config, we will see six node containers running and a HAProxy container:

Container ID	Image	Port	Names
d9107c31eedb	kindest/haproxy: haproxy:v20230606- 42a2262b	0.0.0.0:6443	multinode-external-load- balancer
03a113144845	kindest/node:v1.30.0	127.0.0.1:44445->6443/tcp	multinode-control-plane3
9b078ecd69b7	kindest/node:v1.30.0		multinode-worker2
b779fa15206a	kindest/node:v1.30.0		multinode-worker
8171baafac56	kindest/node:v1.30.0	127.0.0.1:42673->6443/tcp	multinode-control-plane
3ede5e163eb0	kindest/node:v1.30.0	127.0.0.1:43547->6443/tcp	multinode-control-plane2
6a85afc27cf8	kindest/node:v1.30.0		multinode-worker3

Table 2.2: KinD configuration options

Since we have a single host for KinD, each control plane node and the HAProxy container must operate on distinct ports. To enable incoming requests, it is necessary to expose each container on a unique port, since only a single process can be bound to a network port. In this scenario, you can see that port 6443 is the assigned port of the HAProxy container. If you were to examine your Kubernetes configuration file, you would observe that it points to `https://0.0.0.0:6443`, representing the port assigned to the HAProxy container.

When a command is executed using `kubectl`, it is sent directly to the HAProxy server on port 6443. Using a configuration file that was created by KinD during the cluster's creation, the HAProxy container knows how to route traffic among the three control plane nodes, providing a highly available control plane for testing.

The included HAProxy image is not configurable. It is only provided to handle the control plane and to load balance the control plane API traffic. Due to this limitation, if you want to use a load balancer for the worker nodes, you will need to provide your own load balancer. We will explain how to deploy a second HAProxy instance that you can use to load balance incoming traffic among multiple worker nodes later in the chapter.

An example where this would be typically used is when there is a need to utilize an ingress controller across multiple worker nodes. In this scenario, a load balancer would be required to accept incoming requests on ports 80 and 443 and distribute the traffic among the worker nodes, each hosting an instance of the ingress controller. Later in this chapter, we will show a configuration that uses a customized HAProxy setup to load balance traffic across the worker nodes.

You will often find yourself creating clusters that may require additional API settings for your testing. In the next section, we will show you how to add extra options to your cluster, including adding options like **OIDC values** and enabling **feature gates**.

## Customizing the control plane and Kubelet options

You may want to go beyond simple clusters to test features such as **OIDC integration** or **Kubernetes feature gates**.



OIDC provides Kubernetes with authentication and authorization through the OpenID Connect protocol, enabling secure access to the Kubernetes cluster based on user identities.

A **feature gate** in Kubernetes serves as a tool to enable access to experimental or alpha-level features. It functions like a toggle switch, allowing administrators to activate or deactivate specific functionalities within Kubernetes as required.

This requires you to modify the startup options of components, like the API server. KinD uses the same configuration that you would use for a `kubeadm` installation, allowing you to add any optional parameter that you require. As an example, if you wanted to integrate a cluster with an OIDC provider, you could add the required options to the configuration patch section:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
```

```
kubeadmConfigPatches:
  - |
    kind: ClusterConfiguration
    metadata:
      name: config
    apiServer:
      extraArgs:
        oidc-issuer-url: "https://oidc.testdomain.com/auth/idp/k8sIdp"
        oidc-client-id: "kubernetes"
        oidc-username-claim: sub
        oidc-client-id: kubernetes
        oidc-ca-file: /etc/oidc/ca.crt
    nodes:
      - role: control-plane
      - role: control-plane
      - role: control-plane
      - role: worker
      - role: worker
      - role: worker
```

This is only a small example of the type of customization you can do when deploying a KinD cluster. For a list of available configuration options, take a look at the *Customizing control plane configuration with kubeadm* page on the Kubernetes site at <https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/control-plane-flags/>.

Now that you have created the cluster file, let's move on to how you use the configuration file to create your KinD cluster.

## Creating a custom KinD cluster

Finally! Now that you are familiar with KinD, we can move forward and create our cluster.

We need to create a controlled, known environment, so we will give the cluster a name and provide a cluster config file.

Before you begin, make sure that you are in your cloned repository in the chapter2 directory. You will create the entire cluster using our supplied script, `create-cluster.sh`.

This script will create a cluster using a configuration file called `cluster01-kind.yaml`, which will create a cluster called `cluster01` with a control plane and worker node, exposing ports 80 and 443 on the worker node for our `ingress` controller.

Rather than providing each step in the chapter, we have documented the script itself. You can read what each step does when you look at the source code for the script. Below is a high-level list of the steps that are executed by the script:

1. Downloads the KinD v 0.22.0 binary, makes it executable, and moves it to `/usr/bin`.

2. Downloads `kubectl`, make it executable, and moves it to `/usr/bin`.
3. Downloads the **Helm** installation script and executes it.
4. Installs `jq`.
5. Executes KinD to create our cluster using the config file and declaring the image to use (we do this to avoid any issues with newer releases and our chapter scripts).
6. Labels the worker node for ingress.
7. Uses the two manifests, `custom-resources.yaml` and `tigera-operator.yaml`, in the `chapter2/calico` to deploy **Calico**.
8. Deploys the NGINX Ingress using the `nginx-deploy.yaml` manifest in the `chapter2/nginx-ingress` directory.

The manifests we used in steps 7 and 8 are the standard deployment manifests from both the Calico and NGINX-Ingress projects. We will store them in the repository to make the deployments quicker, and also to avoid any issues if either deployment is updated with options that may fail on our KinD cluster.

Congratulations! You now have a fully functioning, two-node Kubernetes cluster running Calico with an Ingress controller.

## Reviewing your KinD cluster

Now that you have a fully functional Kubernetes cluster, we can dive into the realm of a few key Kubernetes objects, specifically storage objects. In the next chapter, *Kubernetes Bootcamp*, we will get deeper into the other objects that are available in a Kubernetes cluster. While that chapter will explore the large list of objects available in a cluster, it is important to introduce the storage-related objects at this point, since KinD provides storage capabilities.

In this section, we shall acquaint ourselves with the storage objects seamlessly integrated within KinD. These purpose-built storage objects extend persistent storage capabilities to your workloads within the cluster, ensuring data persistence and resilience. By familiarizing ourselves with these storage objects, we lay the foundation for seamless data management within the Kubernetes ecosystem.

## KinD storage objects

Remember that KinD includes Rancher's auto-provisioner to provide automated persistent disk management for the cluster. Kubernetes has a number of storage objects, but there is one object that the auto-provisioner does not require, since it uses a base Kubernetes feature: a `CSI``driver` object. Since the ability to use local host paths as PVCs is part of Kubernetes, we will not see any `CSI``driver` objects for local storage in our KinD cluster.

The first object in our KinD cluster we will discuss is `CSInodes`. Any node that can run a workload will have a `CSInode` object. On our KinD clusters, both nodes have a `CSInode` object, which you can verify by executing `kubectl get csinodes`:

NAME	DRIVERS	AGE
cluster01-control-plane	0	20m
cluster01-worker	0	20m

If we were to describe one of the nodes using `kubectl describe csinodes <node name>`, you would see the details of the object:

```
Name:           cluster01-worker
Labels:         <none>
Annotations:   storage.alpha.kubernetes.io/migrated-plugins:
                kubernetes.io/aws-ebs,kubernetes.io/azure-disk,kubernetes.io/azure-file,kubernetes.io/cinder,kubernetes.io/gce-pd,kubernetes.io/vsphere-vo...
CreationTimestamp: Tue, 20 Jun 2023 16:45:54 +0000
Spec:
  Drivers:
    csi.tigera.io:
      Node ID: cluster01-worker
  Events:        <none>
```

The main thing to point out is the Spec section of the output. This lists the details of any drivers that may be installed to support backend storage systems. In the driver section, you will see an entry for a driver called `csi.tigera.io`, which was deployed when we installed Calico. This driver is used by Calico to enable secure connections between Calico's **Felix**, which handles network policy enforcement, and **Dikastes**, which manages Kubernetes network policy translation and enforcement pods by mounting a shared volume.

It is important to note that this driver is not used by standard Kubernetes deployments for persistent storage.

Since the local-provisioner does not require a driver, we will not see an additional driver on our cluster for the local storage.

## Storage drivers

A storage driver in Kubernetes plays an important role in handling the communication between containerized applications and the underlying storage infrastructure. Its primary function is to control the provisioning, attachment, and management of storage resources for applications deployed in Kubernetes clusters.

As we already mentioned, your KinD cluster does not require any additional storage drivers for the local-provisioner, but we do have a driver for Calico's communication. If you execute `kubectl get csidrivers`, you will see the `csi.tigera.io` in the list.

## KinD storage classes

To attach to any cluster-provided storage, the cluster requires a `StorageClass` object. Rancher's provider creates a default storage class called `standard`. It also sets the class as the default `StorageClass`, so you do not need to provide a `StorageClass` name in your PVC requests. If a default `StorageClass` is not set, every PVC request would require a `StorageClass` name in the request. If a default class is not enabled and a PVC request fails to set a `StorageClass` name, the PVC allocation will fail, since the API server won't be able to link the request to a `StorageClass`.

In a production cluster, it is recommended to avoid setting a default `StorageClass`. This approach helps prevent potential issues that can arise when deployments forget to specify a class and the default storage system does not meet the deployment requirements. Such issues may only surface when they become critical in a production environment, impacting business revenue and reputation. By not assigning a default class, developers will encounter a failed PVC request, prompting the identification of the problem before any negative impact on the business. Additionally, this approach encourages developers to explicitly select a `StorageClass` that aligns with the desired performance requirement, enabling them to use cost-effective storage for non-critical systems or high-speed storage for critical workloads.

To list the storage classes on the cluster, execute `kubectl get storageclasses`, or use the shortened version with `sc` instead of `storageclasses`:

NAME	ATTACHREQUIRED	PODINFOOwhoNT	STORAGECAPACITY
csi.tigera.io	true	true	false

Now that you know about the objects that Kubernetes uses for storage, let's learn how to use the provisioner.

## Using KinD's Storage Provisioner

Using the included provisioner is very simple. Since it can auto-provision the storage and is set as the default class, any PVC requests that come in are seen by the provisioning pod, which then creates the `PersistentVolume` and `PersistentVolumeClaim`.

To show this process, let's go through the necessary steps. The following is the output of running `kubectl get pv` and `kubectl get pvc` on a base KinD cluster:

```
kubectl get pv  
No resources found
```

PVs are not namespaced objects, meaning they are cluster-level resources, so we don't need to add a namespace option to the command. PVCs are namespaced objects, so when we tell Kubernetes to show the PVs that are available, we need to specify the namespace with the `kubectl get pvc` command. Since this is a new cluster and none of the default workloads require a persistent disk, there are currently no PV or PVC objects.

Without an auto-provisioner, we would need to create a PV before a PVC could claim the volume. Since we have the Rancher provisioner running in our cluster, we can test the creation process by deploying a pod with a PVC request, like the one listed here, which we will name `pvc-test.yaml`:

```
kind: PersistentVolumeClaim  
apiVersion: v1  
metadata:  
  name: test-claim  
spec:  
  accessModes:
```

```

  - ReadWriteOnce
resources:
  requests:
    storage: 1Mi

```

This PVC will be named `test-claim` in the default namespace, since we didn't provide a namespace, and its volume is set at 1 MB. Again, we do need to include the `StorageClass` option, since KinD has set a default `StorageClass` for the cluster.

To generate the PVC, we can execute a `kubectl` command by using the `create` command, along with the `pvctest.yaml` file, `kubectl create -f pvctest.yaml`. Kubernetes will respond by confirming the creation of the PVC. However, it is crucial to understand that this acknowledgment does not guarantee the PVC's complete functionality. While the PVC object itself was successfully created, it is possible that certain dependencies within the PVC request may be incorrect or missing. In such cases, although the object is created, the PVC request itself will not be fulfilled and may fail.

After creating a PVC, you can check the real status using one of two options. The first is a simple `get` command – that is, `kubectl get pvc`. Since our request is in the default namespace, I don't need to include a namespace value in the `get` command (note that we had to shorten the volume's name so that it fits the page):

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
test-claim	Bound	pvc-b6ecf50...	1Mi	RWO	standard	15s

We know that we created a PVC object by submitting the PVC manifest, but we did not create a PV request. If we look at the PVs now, we can see that a single PV was created from our PVC request. Again, we shortened the PV name in order to fit the output on a single line:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM
pvc-b6ecf...	1Mi	RWO	Delete	Bound	default/test-claim

Given the increasing number of workloads that rely on persistent disks, it's important to have a clear understanding of how Kubernetes workloads integrate with storage systems. In the previous section, you gained insights into how KinD enhances the cluster with the auto-provisioner. In *Chapter 3, Kubernetes Bootcamp*, we will further strengthen our understanding of these Kubernetes storage objects.

In the next section, we will discuss the complex topic of using a load balancer with our KinD cluster to enable highly available clusters, using HAProxy.

## Adding a custom load balancer for Ingress

We added this section for anybody who may want to know more about how to load balance between multiple worker nodes.

This section discusses a complex topic that covers adding a custom HAProxy container that you can use to load balance worker nodes in a KinD cluster. You should not deploy this on the KinD cluster that we will use for the remaining chapters.

Since you will interact with load balancers in most enterprise environments, we wanted to add a section on how to configure your own HAProxy container for worker nodes, in order to load balance between three KinD nodes.

First, we will not use this configuration for any of the chapters in this book. We want to make the exercises available to everyone, so to limit the required resources, we will always use the two-node cluster that we created earlier in this chapter. If you want to test KinD nodes with a load balancer, we suggest using a different Docker host or waiting until you have finished this book and deleted your KinD cluster.

## Creating the KinD cluster configuration

We have provided a script, `create-multinode.sh`, located in the `chapter2/HAdemo` directory, that will create a cluster with three nodes for both the control plane and worker nodes. The script will create a cluster named `multimode`, which means the control plane nodes will be named `multinode-control-plane`, `multinode-control-plane2`, and `multinode-control-plane3`, while the worker nodes will be named `multinode-worker`, `multinode-worker2`, and `multinode-worker3`.

Since we will use a HAProxy container exposed on ports 80 and 443 on your Docker host, you do not need to expose any worker node ports in your KinD config file. The config file we use in the script is shown below:

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
networking:
  apiServerAddress: "0.0.0.0"
  disableDefaultCNI: true
  apiServerPort: 6443
  podSubnet: "10.240.0.0/16"
  serviceSubnet: "10.96.0.0/16"
nodes:
- role: control-plane
- role: control-plane
- role: control-plane
- role: worker
- role: worker
- role: worker
```

Notice that we do not expose the worker nodes on any ports for ingress, so there is no need to expose the ports directly on the nodes. Once we deploy the HAProxy container, it will be exposed on ports 80 and 443, and since it's on the same host as the KinD cluster, the HAProxy container will be able to communicate with the nodes using the Docker network.

At this point, you should have a working multi-node cluster, with a load balancer for the API server and another load balancer for your worker nodes. One of the worker nodes will run an NGINX ingress controller, but it could be any of the three nodes, so how does the HAProxy server know which node is running NGINX? It does a health check against all nodes, and any node that replies with 200 (a successful connection) is running NGINX and is added to the backend(s) server list.

In the next section, we will explain the configuration file that HAProxy uses to control the backend server(s) and the health checks that are executed.

## The HAProxy configuration file

HAProxy offers a container on Docker Hub that is easy to deploy, requiring only a config file to start the container.

To create the configuration file, you will need to know the IP addresses of each worker node in the cluster. The included script that creates the cluster and deploys HAProxy will find this information for you, create the config file, and start the HAProxy container.

Since configuring HAProxy may be unfamiliar to many people, we will provide a breakdown of the script, explaining the main sections that we configured. The script creates the file for us by querying the IP address of the worker node containers:

```
global log /dev/log local0 log /dev/log local1 notice daemon
defaults log global mode tcp timeout connect 5000 timeout client 50000
timeout server 50000

frontend workers_https bind *:443 mode tcp use_backend ingress_https
backend ingress_https option httpchk GET /healthz mode tcp server worker
172.18.0.8:443 check port 80 server worker2 172.18.0.7:443 check port 80
server worker3 172.18.0.4:443 check port 80
frontend stats
  bind *:8404
  mode http
  stats enable
  stats uri /
  stats refresh 10s
frontend workers_http bind *:80 use_backend ingress_http backend ingress_http
mode http option httpchk GET /healthz server worker 172.18.0.8:80 check port
80 server worker2 172.18.0.7:80 check port 80 server worker3 172.18.0.4:80
check port 80
```

The frontend sections are key to the configuration. This tells HAProxy the port to bind to and the server group to use for the backend traffic. Let's look at the first frontend entry:

```
frontend workers_https
  bind *:443
  mode tcp
  use_backend ingress_https
```

This binds a frontend called `workers_https` to TCP port 443. The last line, `use_backend`, tells HAProxy which server group will receive traffic on port 443.

Next, we declare the backend servers, or the collection of nodes that will be running the workloads for the desired port or URL. The first backend section contains the servers that are part of the `workers_https` group.

```
backend ingress_https
  option httpchk GET /healthz
  mode tcp
  server worker 172.18.0.8:443 check port 443
  server worker2 172.18.0.7:443 check port 443
  server worker3 172.18.0.4:443 check port 443
```

The first line contains the name of the rule; in our example, we have called the rule `ingress-https`. The option `httpchk` tells HAProxy how to health check each of the backend servers. If the check is successful, HAProxy will add it as a healthy backend target. If the check fails, the server will not direct any traffic to the failed node(s). Finally, we provide the list of servers; each endpoint has its own line that starts with the server, followed by the name, IP address, and the port to check – in our example, port `443`.

You can use a similar block for any other port that you want HAProxy to load balance. In our script, we configure HAProxy to listen on TCP ports `80` and `443`, using the same backend servers.

We have also added a backend section to expose the HAProxy status page. The status page must be exposed via HTTP, and it runs on port `8404`. This doesn't need to be forwarded to any group of servers, since the status page is part of the Docker container itself. We only need to add it to the configuration file, and when we execute the HAProxy container, we need to add the port mapping for port `8404` (you will see that in the `docker run` command that is described in the next paragraph). We will show the status page and how to use it in the next section.

The final step is to start a Docker container that runs HAProxy with our created configuration file containing the three worker nodes, exposed on the Docker host on ports `80` and `443`, and connected to the KinD network in Docker:

```
# Start the HAProxy Container for the Worker Nodes
docker run --name HAProxy-workers-lb --network $KIND_NETWORK -d -p 8404:8404 -p
80:80 -p 443:443 -v ~/HAProxy:/usr/local/etc/HAProxy:ro haproxy -f /usr/local/
etc/HAProxy/HAProxy.cfg
```

Now that you have learned how to create and deploy a custom HAProxy load balancer for your worker nodes, let's look at how HAProxy communication works.

## Understanding HAProxy traffic flow

The cluster will have a total of eight containers running. Six of these containers will be the standard Kubernetes components – that is, three control plane servers and three worker nodes. The other two containers are KinD's HAProxy server and your own custom HAProxy container (the `docker ps` output has been shortened due to formatting):

CONTAINER	ID	NAMES
3d876a9f8f02	Haproxy	HAProxy-workers-lb
183e86be2be3	kindest/node:v1.30.1	multinode-worker3
ce2d2174a2ba	kindest/haproxy:v20230606-42a2262b	multinode-external-load-balancer
697b2c2bef68	kindest/node:v1.30.1	multinode-control-plane
f3938a66a097	kindest/node:v1.30.1	multinode-worker2
43372928d2f2	kindest/node:v1.30.1	multinode-control-plane2
baa450f8fe56	kindest/node:v1.30.1	multinode-worker
ee4234ff4333	kindest/node:v1.30.1	multinode-control-plane3

Table 2.3: Cluster having eight containers running

The container named HAProxy-workers-lb container is exposed on the host ports 80 and 443. This means that any incoming requests to the host on port 80 or 443 will be directed to the custom HAProxy container, which will then send the traffic to the Ingress controller.

The default NGINX Ingress deployment only has a single replica, which means that the controller runs on a single node, but it could move to any of the other nodes at any time. Let's use the HAProxy status page that we mentioned in the previous section to see where the Ingress controller is running. Using a browser, we need to point to our Docker host's IP address on port 8404. For our example, the host is on 192.168.149.129, so in our browser, we would enter `http://192.168.149.129:8404`, which will bring up the HAProxy status page, similar to what is shown in *Figure 2.8* below:



Figure 2.8: HAProxy status page

In the status page details, you will see the backends that we created in our HAProxy configuration and the status of each service, including which worker node is running the ingress controller. To explain this in more detail, let's focus on the details of the incoming SSL traffic. On the status page, we will focus on the `ingress_https` section, as shown in *Figure 2.9*.



ingress_https			Session rate		
	Queue		Cur	Max	Limit
worker	0	0	-	0	0
worker2	0	0	-	0	0
worker3	0	0	-	0	0
Backend	0	0		0	0

Status	LastChk
9m1s DOWN	L4CON in 0ms
9m1s UP	L6OK in 1ms
9m1s DOWN	L4CON in 0ms
9m1s UP	

*Figure 2.9: HAProxy HTTPS Status*

In the HAProxy config, we created a backend called `ingress_https` that includes all of the worker nodes in the cluster. Since we only have a single replica running for the controller, only one node will run the ingress controller. In the list of nodes, you will see that two of them are in a DOWN state, while `worker2` is in an UP state. The DOWN state is expected, since the health check for HTTPS will fail on any node that isn't running a replica of the ingress controller.

While we would run at least three replicas in production, we only have three nodes, and we want to show how HAProxy will update the backend services when the ingress controller pod moves from the active node to a new node. So, we will simulate a node failure to prove that HAProxy provides high availability to our NGINX ingress controller.

## Simulating a kubelet failure

In our example, we want to prove that HAProxy provides HA support for NGINX. To simulate a failure, we can stop the `kubelet` service on a node, which will alert the `kube-apiserver` so that it doesn't schedule any additional pods on the node. We know that the running container is on `worker2`, so that's the node we want to take down.

The easiest way to stop `kubelet` is to send a `docker exec` command to the container:

```
docker exec multinode-worker2 systemctl stop kubelet
```

You will not see any output from this command, but if you wait a few minutes for the cluster to receive the updated node status, you can verify the node is down by looking at a list of nodes:

```
kubectl get nodes
```

You will receive the following output:

NAME	AGE	VERSION	STATUS	ROLES
multinode-control-plane	Ready		control-plane	29m v1.30.0
multinode-control-plane2	Ready		control-plane	29m v1.30.0
multinode-control-plane3	Ready		control-plane	29m v1.30.0

multinode-worker	Ready	<none>
28m v1.30.0		
multinode-worker2	NotReady	<none>
28m v1.30.0		
multinode-worker3	Ready	<none>
28m v1.30.0		

This verifies that we just simulated a kubelet failure and that worker2 is now down, in a NotReady status.

Any pods that were running before the kubelet “failure” will continue to run, but kube-scheduler will not schedule any workloads on the node until the kubelet issue is resolved. Since we know the pod will not restart on the node, we can delete the pod so that it can be rescheduled on a different node.

You need to get the pod name and then delete it to force a restart:

```
kubectl get pods -n ingress-nginx
```

This will return the pods in the namespace, such as the following:

```
nginx-ingress-controller-7d6bf88c86-r7ztq
```

Delete the ingress controller pod using kubectl:

```
kubectl delete pod nginx-ingress-controller-7d6bf88c86-r7ztq -n ingress-nginx
```

This will force the scheduler to start the container on another worker node. It will also cause the HAProxy container to update the backend list, since the NGINX controller has moved to another worker node.

To prove this, we can look at the HAproxy status page, and you see that the active node has changed to **worker3**. Since the failure that we simulated was for **worker2**, when we killed the pod, Kubernetes rescheduled the pod to start up on another, healthy, node.

ingress_https									
	Queue			Session rate					
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	
worker	0	0	-	0	0		0	0	
worker2	0	0	-	0	0		0	0	
worker3	0	0	-	0	0		0	0	
Backend	0	0		0	0		0	0	

Figure 2.10: HAproxy backend node update

If you plan to use this HA cluster for additional tests, you will want to restart the kubelet on **multinode-worker2**.

If you plan to delete the HA cluster, you can just run a KinD cluster delete, and all the nodes will be deleted. Since we called the cluster **multinode**, you would run the following command to delete the KinD cluster:

```
kind delete cluster --name multinode
```

You will also need to delete the HAProxy container that we deployed for the worker nodes, since we executed that container from Docker and it was not created by the KinD deployment. To clean up the worker nodes' HAProxy deployment, execute the commands below:

```
docker stop HAProxy-workers-lb && docker rm HAProxy-workers-lb
```

That completes this KinD chapter! We mentioned a lot of different Kubernetes services in this chapter, but we only scratched the surface of the objects included in clusters. In the next section, we will go through a bootcamp on what components make up a cluster and provide an overview of the base objects in a cluster.

## Summary

This chapter provided an overview of the KinD project, a Kubernetes SIG project. We covered the process of installing optional components in a KinD cluster, such as Calico for CNI and NGINX for Ingress control. Additionally, we explored the Kubernetes storage objects that are included with a KinD cluster.

You should now understand the potential benefits that KinD can bring to you and your organization. It offers a user-friendly and highly customizable Kubernetes cluster deployment, and the number of clusters on a single host is only limited by the available host resources.

In the next chapter, we will dive into Kubernetes objects. We've called the next chapter *Kubernetes Bootcamp*, since it will cover the majority of the basic Kubernetes objects and what each one is used for. The next chapter can be considered a "Kubernetes pocket guide." It contains a quick reference to Kubernetes objects and what they do, as well as when to use them.

It's a packed chapter and is designed to be a refresher for those of you who have experience with Kubernetes; alternatively, it's a crash course for those of you who are new to Kubernetes. Our intention with this book is to go beyond the basic Kubernetes objects, since there are many books on the market today that cover the basics of Kubernetes very well.

## Questions

1. What object must be created before you can create a `PersistentVolumeClaim`?

- a. PVC
- b. A disk
- c. `PersistentVolume`
- d. `VirtualDisk`

Answer: c

2. KinD includes a dynamic disk provisioner. Which company created the provisioner?

- a. Microsoft
- b. CNCF
- c. VMware
- d. Rancher

Answer: d

3. If you created a KinD cluster with multiple worker nodes, what would you install to direct traffic to each node?
  - a. A load balancer
  - b. A proxy server
  - c. Nothing
  - d. Replicas set to 3

Answer: a

4. True or false? A Kubernetes cluster can only have one CSIDriver installed.
  - a. True
  - b. False

Answer: b



# 3

## Kubernetes Bootcamp

The previous chapter introduced you to deploying Kubernetes clusters using KinD (Kubernetes in Docker), which is useful for creating a development cluster on a single machine using containers instead of virtual machines. This approach reduces the system resource requirements and simplifies the entire setup process. We covered the installation and configuration of KinD, creating clusters, including add-ons like Ingress controllers, Calico as the CNI, and using persistent storage.

We understand that many of you have experience with Kubernetes, whether it's running clusters in production or experimenting with tools like kubeadm, minikube, or Docker Desktop. Our intention with this book is to go beyond the fundamentals of Kubernetes, which is why we didn't want to reiterate all the basics. Instead, we've included this chapter as a bootcamp for those who are new to Kubernetes or have only had limited exposure to it.

In this chapter, we will explore the essential components of a Kubernetes cluster, including the control plane and worker nodes. We will provide detailed explanations of each Kubernetes resource and its respective use cases. If you have previous experience with Kubernetes and feel comfortable using kubectl, as well as an understanding of Kubernetes resources like **DaemonSets**, **StatefulSets**, and **ReplicaSets**, this chapter can serve as a helpful review before moving on to *Chapter 4*, where we will dive into **Services**, **Load Balancing**, **ExternalDNS**, **Global Balancing**, and **K8GB**.

Since this is a bootcamp chapter, we won't get into every topic in detail. However, by the end of this chapter, you should have a solid understanding of the foundational concepts of Kubernetes, which will be crucial for comprehending the remaining chapters. Even if you already possess a strong background in Kubernetes, you may find this chapter valuable as a refresher before we get into more advanced topics. In this chapter, you will learn the following topics:

- An overview of Kubernetes components
- Exploring the control plane
- Understanding the worker node components
- Interacting with the API server
- Introducing Kubernetes resources

By the end of this chapter, you will have a solid understanding of the most commonly used cluster resources. Understanding Kubernetes resources is important for both cluster operators and cluster administrators.

## Technical requirements

This chapter has no technical requirements.

If you want to execute commands while learning about the resources, you can use the Kind cluster that was deployed in the previous chapter.

## An overview of Kubernetes components

Understanding the components of systems in an infrastructure is essential for delivering services effectively. In today's wide landscape of installation options, many Kubernetes users may not have felt the need to fully comprehend the integration of different Kubernetes components.

Just a few years ago, establishing a Kubernetes cluster involved the manual installation and configuration of each component. This process presented a steep learning curve and often resulted in frustration. As a result, many individuals and organizations concluded that "Kubernetes is overly complex." However, the benefit of manual installation was the in-depth understanding it provided regarding the interaction between each component. If any issues arose within the cluster after installation, you would have a clear understanding of where to investigate.

To understand how Kubernetes components work together, you must first understand the different components of a Kubernetes cluster.

The following diagram is from the <http://Kubernetes.io> site and shows a high-level overview of a Kubernetes cluster component:

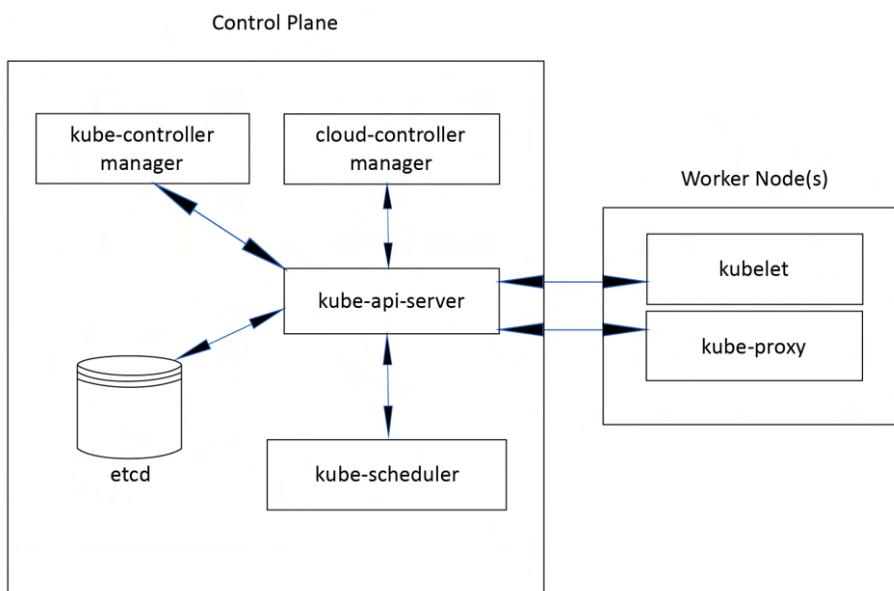


Figure 3.1: Kubernetes cluster components

As you can see, the Kubernetes cluster is made up of several components. As we progress through the chapter, we'll discuss these components and the role they play in a Kubernetes cluster.

## Exploring the control plane

The control plane, as its name suggests, has authority over every aspect of a cluster. In the absence of a functioning control plane, the cluster loses its ability to schedule workloads, create new deployments, and manage Kubernetes objects. Recognizing the criticality of the control plane, it is highly advisable to deploy with **high availability (HA)** support, deploying a minimum of three control plane nodes. Many production environments even utilize more than three control plane nodes, but the key principle is to have an odd number of nodes, which is required so we can maintain a highly available control plane if we lose a single etcd node.

Now, let's delve into the importance of the control plane and its components, providing a comprehensive understanding of their pivotal role in a functioning cluster.

## The Kubernetes API server

The first component to understand in a cluster is the kube-apiserver component. Since Kubernetes is **application programming interface (API)-driven**, every request that comes into a cluster goes through the API server. Let's look at a simple get nodes request using an API endpoint, using the IP address for the control plane, which, in an enterprise, is usually fronted by a load balancer. In our example, our load balancer has an entry for the three control plane nodes on **10.240.100.100**:

```
https://10.240.100.100:6443/api/v1/nodes?limit=500
```

If you attempt to make an API call without any credentials, you will receive a permission denied request. Using a pure API request directly is something that is very common when creating a pipeline for application deployment or even a Kubernetes add-on component. However, the most common method for users to interact with Kubernetes is the kubectl utility.

Every command that is issued using kubectl calls an API endpoint behind the scenes. In the preceding example, if we executed a kubectl get nodes command, an API request would be sent to the kube-apiserver process using the address **10.240.100.100** on port **6443**.

The API call requested the /api/v1/nodes endpoint, which returned a list of the nodes in the cluster:

NAME	STATUS	ROLES	AGE	VERSION
home-k8s-control-plane	Ready	control-plane,master	45d	v1.27.3
home-k8s-control-plane2	Ready	control-plane,master	45d	v1.27.3
home-k8s-control-plane3	Ready	control-plane,master	45d	v1.27.3
home-k8s-worker	Ready	worker	45d	v1.27.3
home-k8s-worker2	Ready	worker	45d	v1.27.3
home-k8s-worker3	Ready	worker	45d	v1.27.3

In the absence of a functioning API server, all requests directed to your cluster will fail. Therefore, it becomes crucial to ensure the continuous operation and health of the kube-apiserver.

By running three or more control plane nodes, we minimize any potential impact that could be caused by the loss of a control plane node.

Remember, from the last chapter, that when running more than one control plane node, you need to have a load balancer in front of the cluster's API server. The Kubernetes API server can be fronted by most standard solutions, including F5, HAProxy, and Seesaw.

## The etcd database

Describing etcd as the foundation of your Kubernetes cluster would not be an overstatement. etcd functions as a robust and highly efficient distributed key-value database that Kubernetes relies on to store all cluster data. Every resource present within the cluster is associated with a specific key in the etcd database. If you can access the node or pod that hosts etcd, you will be able to use the etcdctl executable to explore all the keys stored within the database. The code snippet provided below offers an example extracted from a cluster based on KinD:

```
etcdctl-3.5.12 --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/
etcd/ca.crt --key=/etc/kubernetes/pki/etcd/server.key --cert=/etc/kubernetes/
pki/etcd/server.crt get / --prefix --keys-only
```

The output from the preceding command contains too much data to list it all in this chapter. A base KinD cluster will return approximately 314 entries.

All keys start with `/registry/<resource>`. For example, one of the keys that was returned is the `ClusterRole` for the `cluster-admin` key, as follows: `/registry/clusterrolebindings/cluster-admin`.

We can use the key name to retrieve the value using the `etcdctl` utility by slightly modifying our previous command, as follows:

```
etcdctl-3.5.12 --endpoints=https://127.0.0.1:2379 --cacert=/etc/kubernetes/pki/
etcd/ca.crt --key=/etc/kubernetes/pki/etcd/server.key --cert=/etc/kubernetes/
pki/etcd/server.crt get /registry/clusterrolebindings/cluster-admin
```

The output will contain characters that cannot be interpreted by your shell, but you will get an idea of the data stored in etcd. For the `cluster-admin` key, the output shows us the following:

```
/registry/clusterrolebindings/cluster-admin
k8s
2
rbac.authorization.k8s.io/v1ClusterRoleBinding
[]
cluster-admin"*$7b235e81-52de-4001-b354-994dad0279ee2[[REDACTED]]Z,
rbac-defaultsbootstrapstrapping
+rbac.authorization.kubernetes.io/autoupdate=true[[REDACTED]]
kube-apiserverUpdaterbac.authorization.k8s.io/v[[REDACTED]]FieldsV1:
[{"f:metadata":{"f:annotations":{".":{}}, "f:rbac.authorization.
kubernetes.io/autoupdate":{}}, "f:labels":{".":{}}, "f:kubernetes.io/
bootstrapping":{}}, {"f:roleRef":{}}, {"f:subjects":{}}]B4
```

```
Grouprbac.authorization.k8s.io/system:masters"7
rbac.authorization.k8s.io
cluster-admin"           ClusterRole
```

We describe the entries in etcd to offer an understanding of how Kubernetes stores and utilizes data to manage cluster objects. While you've already observed the direct database output for the `cluster-admin` key, in typical scenarios, you would utilize the command `kubectl get clusterrolebinding cluster-admin -o yaml` to query the API server for the same data. Using `kubectl`, the command would yield the following information:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: "true"
  creationTimestamp: "2023-06-21T13:21:08Z"
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  name: cluster-admin
  resourceVersion: "171"
  uid: 350a50f0-f945-4f0e-b187-ab3063cafca3
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:masters
```

*Figure 3.2: kubectl ClusterRoleBinding output*

If you look at the output from the `kubectl` command and compare it with the output from the `etcdctl` query, you will see matching information. You will rarely have to interact with etcd directly; instead, you will execute `kubectl` commands, and the request will go to the API server, which then queries the etcd database for the resource's information.

It's worth noting that while etcd is by far the most used backend database for Kubernetes, it isn't the only one. The k3s project, which was originally built to strip down Kubernetes for edge use cases, replaced etcd with relational databases. When we dive into vclusters, which use k3s, we'll see that it uses SQLite instead of etcd.

## kube-scheduler

As its name suggests, the `kube-scheduler` component oversees the allocation of pods to nodes. Its primary task is to consistently monitor pods that have yet to be assigned to any specific node. The scheduler then assesses the resource requirements of each pod to determine the most suitable placement. This assessment takes multiple factors into account, including the availability of node resources, constraints, selectors, and affinity/anti-affinity rules. Nodes that satisfy these requirements are considered feasible nodes. Finally, from the resulting list of compatible nodes, the scheduler chooses the most appropriate one for scheduling the pod.

## kube-controller-manager

The `Kubernetes Controller Manager` is a central control system in the Kubernetes control plane; it's responsible for managing and coordinating other controllers that handle specific tasks for the cluster.

The Controller Manager contains multiple controllers, each dedicated to a specific function within the cluster. These controllers continuously monitor the cluster's current state and adapt dynamically to maintain the desired configuration.

All of the controllers are contained in a single executable, reducing complexity and management. Some of the controllers included are shown in *Table 3.1*.

Each controller provides a unique function to a cluster, and each controller and its function is listed here:

Controller	Responsibilities
Endpoints	Monitors new services and creates endpoints to the pods with matching labels
Namespace	Monitors actions for namespaces
Node	Monitors the status of nodes in the cluster, detecting node failures or additions, and taking appropriate actions to maintain the desired number of nodes
Replication	Monitors the replicas for pods, taking action to either remove a pod or add a pod to get to the desired state
Service Accounts	Monitors Service accounts

*Table 3.1: Controllers and their functions*

Each controller runs a non-terminating (never-ending) control loop. These control loops monitor the state of each resource, making any changes required to normalize the state of the resource. For example, if you needed to scale a deployment from one to three nodes, the replication controller would notice that the current state has one pod running, and the desired state is to have three pods running. To move the current state to the desired state, two additional pods would be requested by the replication controller.

## **cloud-controller-manager**

This is one component that you may not have run into, depending on how your clusters are configured. Similar to the `kube-controller-manager` component, this controller contains four controllers in a single binary.

The cloud controller provides integrations specific to a particular cloud provider's Kubernetes service, enabling the utilization of cloud-specific functionalities such as load balancers, persistent storage, auto-scaling groups, and other features.

## **Understanding the worker node components**

Worker nodes, as implied by their name, have the duty of carrying out tasks within a Kubernetes cluster. In our previous conversation about the `kube-scheduler` element in the control plane, we emphasized that when a new pod requires scheduling, the `kube-scheduler` selects the suitable node for its execution. The `kube-scheduler` relies on data provided by the worker nodes to make this determination. This data is regularly updated to ensure a distribution of pods throughout the cluster, making the most of the cluster resources.

Each worker node has two main components, `kubelet` and `kube-proxy`.

### **kubelet**

You may hear a worker node referred to as a `kubelet`. The `kubelet` is an agent that runs on all worker nodes, and it is responsible for ensuring that containers are running and healthy on the node.

### **kube-proxy**

Contrary to the name, `kube-proxy` is not a proxy server at all (though it was in the original version of Kubernetes).

Depending on the CNI deployed in your cluster, you may or may not have a `kube-proxy` component on your nodes. CNIs like **Cilium** can be run with `kube-proxy` or in a `kube-proxyless` mode. In our KinD clusters, we have deployed Calico, which relies on the presence of `kube-proxy`.

When `kube-proxy` is deployed, its main purpose is to oversee network connectivity for pods and services in the cluster, providing network traffic routing to the destination pod(s).

## **Container runtime**

Each node also needs a container runtime. A container runtime is responsible for running the containers. The first thing you might think of is Docker, and while Docker is a container runtime, it is not the only runtime option available. Over the last several years, other options have replaced Docker as the preferred container runtime for clusters.

The two most prominent Docker replacements are **CRI-O** and **containerd**. At the time of writing this chapter, KinD only offers official support for Docker and Red Hat's **Podman**.

## Interacting with the API server

As we mentioned earlier, you interact with the API server using either direct API requests or the `kubectl` utility. We will focus on using `kubectl` for the majority of our interaction in this book, but we will call out using direct API calls wherever applicable.

## Using the Kubernetes `kubectl` utility

`kubectl` is a single executable file that allows you to interact with the Kubernetes API using a **command-line interface (CLI)**. It is available for most major operating systems and architectures, including Linux, Windows, and macOS.

Note: We have already installed `kubectl` using the KinD script that created our cluster in *Chapter 2*. Installation instructions for most operating systems are located on the Kubernetes site at <https://kubernetes.io/docs/tasks/tools/install-kubectl/>. Since we are using Linux as our operating system for the exercises in the book, we will cover installing `kubectl` on a Linux machine. Follow these steps:

1. To download the latest version of `kubectl`, you can run a `curl` command that will download it, as follows:

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

2. After downloading, you need to make the file executable by running the following command:

```
chmod +x ./kubectl
```

3. Finally, we will move the executable to our path, as follows:

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

You now have the latest `kubectl` utility on your system and can execute `kubectl` commands from any working directory.

Kubernetes is updated about every 4 months. This includes upgrades to the base Kubernetes cluster components and the `kubectl` utility. You may run into a version mismatch between a cluster and your `kubectl` command, requiring you to either upgrade or download your `kubectl` executable. You can always check the version of both by running a `kubectl version` command, which will output the version of both the API server and the `kubectl` client. The output from a version check is shown in the following code snippet – please note, your output may differ from our example output:

```
Client Version: v1.30.0-6+43a0480e94cee1  
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3  
Server Version: v1.30.0-6+43a0480e94cee1
```

As you can see from the output, the `kubectl` client is running version 1.30.0 and the cluster is running 1.30.0. A minor version difference in the two will not cause any issues. In fact, the official supported version difference is within one major version release. So, if your client is running version 1.29 and the cluster is running 1.30.0, you would be within the supported version difference. While this may be supported, it doesn't mean that you won't run into issues if you are trying to use any new commands or resources included in the higher version. In general, you should try to keep your cluster and client version in sync to avoid any issues.

Through the remainder of this chapter, we will discuss Kubernetes resources and how you interact with the API server to manage each one. But before diving into the different resources, we wanted to mention one commonly overlooked option of the `kubectl` utility: the `verbose` option.

## Understanding the verbose option

When you execute a `kubectl` command, the only outputs you will see by default are any direct responses to your command. If you were to look at all pods in the `kube-system` namespace, you would receive a list of all pods. In most cases, this is the desired output, but what if you issued a `get Pods` request and received an error from the API server? How could you get more information about what might be causing the error?

By adding the `verbose` option to your `kubectl` command, you can get additional details about the API call itself and any replies from the API server. Often, the replies from the API server will contain additional information that may be useful to find the root cause of the issue.

The `verbose` option has multiple levels ranging from 0 to 9; the higher the number, the more output you will receive.

The following screenshot has been taken from the Kubernetes site, detailing each level and what the output will include:

Verbosity	Description
--v=0	Generally useful for this to always be visible to a cluster operator.
--v=1	A reasonable default log level if you don't want verbosity.
--v=2	Useful steady state information about the service and important log messages that may correlate to significant changes in the system. This is the recommended default log level for most systems.
--v=3	Extended information about changes.
--v=4	Debug level verbosity.
--v=6	Display requested resources.
--v=7	Display HTTP request headers.
--v=8	Display HTTP request contents.
--v=9	Display HTTP request contents without truncation of contents.

Figure 3.3: Verbosity description

You can experiment with the levels by adding the `-v` or `--v` option to any `kubectl` command.

## General `kubectl` commands

The CLI allows you to interact with Kubernetes in an imperative and declarative manner. Using an imperative command involves you telling Kubernetes what to do—for example, `kubectl run nginx --image nginx`. This tells the API server to create a new pod called `nginx` that runs an image called `nginx`. While imperative commands are useful for development and quick fixes or testing, you will use declarative commands more often in a production environment. In a declarative command, you tell Kubernetes what you want. To use declarative commands, you send a manifest to the API server, written in either **JavaScript Object Notation (JSON)** or **YAML Ain't Markup Language (YAML)**, which declares what you want Kubernetes to create.

`kubectl` includes commands and options that can provide general cluster information or information about a resource. The table below contains a cheat sheet of commands and what they are used for. We will use many of these commands in future chapters, so you will see them in action throughout the book:

Cluster Commands	
<code>api-resources</code>	Lists supported API resources
<code>api-versions</code>	Lists supported API versions
<code>cluster-info</code>	Lists cluster information, including the API server and other cluster endpoints
Object Commands	
<code>get &lt;object&gt;</code>	Retrieves a list of all objects (i.e., pods, ingress, etc.)
<code>describe &lt;object&gt;</code>	Provides details for the object
<code>logs &lt;pod name&gt;</code>	Retrieve the logs for a pod
<code>edit &lt;object&gt;</code>	Edits an object interactively
<code>delete &lt;object&gt;</code>	Deletes an object
<code>label &lt;object&gt;</code>	Labels an object
<code>annotate &lt;object&gt;</code>	Annotates an object
<code>run</code>	Creates a pod

*Table 3.2: Cluster and object commands*

With an understanding of each Kubernetes component and how to interact with the API server using imperative commands, we can now move on to Kubernetes resources and how we use `kubectl` to manage them.

## Introducing Kubernetes resources

In this section, we will provide a substantial amount of information. However, as this is a bootcamp, we won't go into exhaustive details about each resource. It's worth noting that each resource could easily warrant its own dedicated chapter or even multiple chapters in a book. Since numerous books on basic Kubernetes already cover these resources extensively, we will focus on the essential aspects necessary for a basic understanding of each resource. As we progress through the subsequent chapters, we will supplement additional details about the resources as we expand our cluster using the exercises provided in the book.

Prior to delving into a comprehensive understanding of Kubernetes resources, let's begin by introducing the concept of Kubernetes manifests.

### Kubernetes manifests

The files that we will use to create Kubernetes resources are referred to as manifests. A manifest can be created using YAML or JSON—most manifests use YAML, and that is the format we will use throughout the book.



It's important to note that while we are working with YAML files, `kubectl` will convert all YAML into JSON when interacting with your API server. All API calls are made with JSON, even if the manifests are written in YAML.

The content of a manifest will vary depending on the resource, or resources, that will be created. At a minimum, all manifests require a base configuration that includes `apiVersion`, the kind of resource, and `metadata` fields, as can be seen here:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: grafana
  name: grafana
  namespace: monitoring
```

The preceding manifest alone is not complete; we are only showing the beginning of a full `Deployment` manifest. As you can see in the file, we start with the three required fields that all manifests are required to have: the `apiVersion`, `kind`, and `metadata` fields.

You may also notice that there is a format for fields in the file. YAML is very format-specific, and if the format of any line is off by even a single space, you will receive an error when you try to deploy the manifest. This takes time to get used to, and even after creating manifests for a long time, formatting issues will still pop up from time to time.

## What are Kubernetes resources?

When you want to add or delete something from a cluster, you are interacting with Kubernetes resources. This interaction is how you declare your desired state for the resource, which may be to create, delete, or scale a resource. Based on the desired state, the API server will make sure that the current state equals the desired state. For example, if you have a deployment that starts with a single replica, you can change the deployment resource from 1 to 3 replicas. When the API server sees that the current state is 1, it will scale the deployment out to 3 replicas by creating the additional 2 pods.

To retrieve a list of resources a cluster supports, you can use the `kubectl api-resources` command. The API server will reply with a list of all resources, including any valid short name, namespace support, and supported API group.

There are approximately 58 base resources included with a Kubernetes cluster, but it's very common to have many more than 58 in a production cluster. Many add-on components, like Calico, will extend the Kubernetes API with new objects. As a cluster has different add-ons deployed in the cluster, don't be surprised at 100+ resources in a list.

An abbreviated list of the most common resources is shown below:

NAME	SHORT NAMES	API VERSION	NAME-SPACED
apiservices	apiregistration.k8s.io/v1	FALSE	
certificatesigningrequests	Csr	certificates.k8s.io/v1	FALSE
clusterrolebindings	rbac.authorization.k8s.io/v1	FALSE	
clusterroles	rbac.authorization.k8s.io/v1	FALSE	
componentstatuses	Cs	v1	FALSE
configmaps	Cm	v1	TRUE
controllerrevisions	apps/v1	TRUE	
cronjobs	Cj	batch/v1	TRUE
csidrivers	storage.k8s.io/v1	FALSE	
csinodes	storage.k8s.io/v1	FALSE	
csistoragecapacities	storage.k8s.io/v1	TRUE	
customresourcedefinitions	crd,crds	apiextensions.k8s.io/v1	FALSE
daemonsets	Ds	apps/v1	TRUE
deployments	Deploy	apps/v1	TRUE
endpoints	Ep	v1	TRUE
endpointslices	discovery.k8s.io/v1	TRUE	

events	Ev	v1	TRUE
events	Ev	events.k8s.io/v1	TRUE
flowschemas	flowcontrol.apiserver.k8s.io/v1beta3	FALSE	
horizontalpodautoscalers	Hpa	autoscaling/v2	TRUE
ingressclasses	networking.k8s.io/v1	FALSE	
ingresses	Ing	networking.k8s.io/v1	TRUE
jobs	batch/v1	TRUE	
limitranges	Limits	v1	TRUE
localsubjectaccessreviews	authorization.k8s.io/v1	TRUE	
mutatingwebhookconfigurations	admissionregistration.k8s.io/v1	FALSE	
namespaces	Ns	v1	FALSE
networkpolicies	Netpol	networking.k8s.io/v1	TRUE
nodes	No	v1	FALSE
persistentvolumeclaims	Pvc	v1	TRUE
persistentvolumes	pv	v1	FALSE
poddisruptionbudgets	pdb	policy/v1	TRUE
pods	po	v1	TRUE
podtemplates	v1	TRUE	
priorityclasses	pc	scheduling.k8s.io/v1	FALSE
prioritylevelconfigurations	flowcontrol.apiserver.k8s.io/v1beta3	FALSE	
profiles	projectcalico.org/v3	FALSE	
replicasetss	rs	apps/v1	TRUE
replicationcontrollers	rc	v1	TRUE
resourcequotas	quota	v1	TRUE
rolebindings	rbac.authorization.k8s.io/v1	TRUE	
roles	rbac.authorization.k8s.io/v1	TRUE	
runtimeclasses	node.k8s.io/v1	FALSE	
secrets	v1	TRUE	

selfsubjectaccessreviews	authorization.k8s.io/v1	FALSE	
selfsubjectrulesreviews	authorization.k8s.io/v1	FALSE	
serviceaccounts	sa	v1	TRUE
services	svc	v1	TRUE
statefulsets	sts	apps/v1	TRUE
storageclasses	sc	storage.k8s.io/v1	FALSE
subjectaccessreviews	authorization.k8s.io/v1	FALSE	
tokenreviews	authentication.k8s.io/v1	FALSE	
validatingwebhookconfigurations	admissionregistration.k8s.io/v1	FALSE	
volumeattachments	storage.k8s.io/v1	FALSE	

Table 3.3: Kubernetes API resources

As this chapter functions as a bootcamp, we will provide a short overview of the resources found in *Table 3.3*. To effectively comprehend the following chapters, it is important for you to possess a strong understanding of the objects and their respective functions.

Some resources will also be explained in greater detail in future chapters, including `Ingress`, `RoleBindings`, `ClusterRoles`, `StorageClasses`, and more.

## Reviewing Kubernetes resources

Most resources in a cluster are run in a namespace, and to create/edit/read them, you should supply the `-n <namespace>` option to any `kubectl` command. To find a list of resources that accept a namespace option, you can reference the output from *Table 3.3*. If a resource can be referenced by a namespace, the `NAMESPACED` column will show `TRUE`. If the resource is only referenced by the cluster level, the `NAMESPACED` column will show `FALSE`.

## Apiservices

Apiservices provide the primary entry point for communication and interaction between Kubernetes components and all external resources, such as users, applications, and other services. They provide a set of endpoints that allow users and applications to perform various operations, such as creating, updating, and deleting Kubernetes resources (i.e., pods, deployments, services, and namespaces).

Apiservices handle the authentication, authorization, and validation of requests, allowing only authorized users and applications to access or modify resources. They also handle resource versioning and other critical aspects of a Kubernetes cluster.

They can also extend Kubernetes functionality by developing custom controllers, operators, or other components that interact with the API Services to manage and automate various aspects of the cluster's behavior. One example of this is our CNI, Calico, which adds 31 extra api-resources to a cluster.

## CertificateSigningRequests

A CertificateSigningRequest (CSR) allows you to request a certificate from a certificate authority. These are typically used to obtain trusted certificates for securing communication within a Kubernetes cluster.

## ClusterRoles

A ClusterRole is a collection of permissions that enable interaction with the API of the cluster. It pairs an action, or verb, with an API group to define a specific permission. For example, if you intended to restrict a continuous integration/continuous delivery (CI/CD) pipeline's ability to only patch Deployments for updating image tags, you could utilize a ClusterRole similar to the following:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: patch-deployment
  rules:
    - apiGroups: ["apps/v1"]
      resources: ["deployments"]
      verbs: ["get", "list", "patch"]
```

A ClusterRole can apply to APIs at both the cluster and namespace levels.

## ClusterRoleBindings

Once you have specified a ClusterRole, the next step is to create an association between the ClusterRole and a subject using a ClusterRoleBinding. This binding links the ClusterRole to a user, group, or service account, granting them the permissions defined within the ClusterRole.

We'll explore ClusterRoleBinding in more detail in *Chapter 7, RBAC Policies and Auditing*.

## ComponentStatus

The Kubernetes control plane is a crucial component for a cluster; it is essential for the operation of the cluster. ComponentStatus is an object that shows the health and status of different Kubernetes control plane components. It provides an indicator of the overall health of a component, providing information on whether it is operating correctly or has errors.

## ConfigMaps

A ConfigMap is a resource that stores data in key-value pairs, enabling the separation of configuration from your application. ConfigMaps can hold various types of data, including literal values, files, or directories, allowing flexibility in managing your application's configuration.

Here is an imperative example:

```
kubectl create configmap <name> <data>
```

The `<data>` option will vary based on the source of the ConfigMap. To use a file or a directory, you supply the `--from-file` option and either the path to a file or an entire directory, as shown here:

```
kubectl create configmap config-test --from-file=/apps/nginx-config/nginx.conf
```

This would create a new ConfigMap named `config-test`, with the `nginx.conf` key containing the content of the `nginx.conf` file as the value.

If you need to have more than one key added in a single ConfigMap, you put each file into a directory and create the ConfigMap using all of the files in the directory. For example, you have three files in a directory located at `~/config/myapp`. The files each contain data and are called `config1`, `config2`, and `config3`. To create a ConfigMap that would add each file into a key, you need to supply the `--from-file` option and point to the directory, as follows:

```
kubectl create configmap config-test --from-file=/apps~/config/myapp
```

This would create a new ConfigMap with three key values called `config1`, `config2`, and `config3`. Each key will contain a value equal to the content of each file in the directory.

To quickly show a ConfigMap, using the example mentioned above, we can retrieve it using the `get` command, `kubectl get configmaps config-test`, resulting in the following output:

NAME	DATA	AGE
config-test	3	7s

The ConfigMap is comprised of three keys, indicated by the presence of the number 3 under the DATA column. For a more detailed examination, we can utilize the `kubectl get` command with the additional `-o yaml` option appended to the `kubectl get configmaps config-test` command. This will show the output of each key's value represented in YAML format, as demonstrated below:

```
apiVersion: v1
data:
  config1: |
    First Configmap Value
  config2: |
    Yet Another Value from File2
  config3: |
    The last file - Config 3
kind: ConfigMap
metadata:
  creationTimestamp: "2023-06-10T01:38:51Z"
  name: config-test
  namespace: default
  resourceVersion: "6712"
  uid: a744d772-3845-4631-930c-e5661d476717
```

By examining the output, it shows that each key within the ConfigMap corresponds to the filenames found in the directory—config1, config2, and config3. Each key retains the value obtained from the data within its respective file.

One limitation of ConfigMaps that you should keep in mind is that the data is easily accessible to anyone with permission to the resource. As you can see from the preceding output, a simple get command shows the data in cleartext.

Due to this design, you should never store sensitive information such as a password in a ConfigMap. Later in this section, we will cover a resource that was designed to store secret data information, called a Secret.

## ControllerRevisions

A ControllerRevision is like a snapshot of a particular version or update to a controller's settings. It's mainly used by specific controllers, such as the StatefulSet controller, to keep track of and manage changes made to their configurations as time goes on.

Whenever there are modifications or updates to the configuration of a resource managed by a controller, a new ControllerRevision is created. Each revision includes the desired setup of the controller and a revision number. These revisions are stored in the Kubernetes API server, allowing you to refer to or revert to them whenever necessary.

## CronJobs

If you have used Linux cronjobs in the past, then you already know what a Kubernetes CronJob resource is. If you don't have a Linux background, a cronjob is used to create a scheduled task. As another example, if you are a Windows person, it's similar to Windows scheduled tasks.

An example manifest that creates a CronJob is shown in the following code snippet:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello-world
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello-world
              image: busybox
              args:
                - /bin/sh
```

```
- -c  
- date; echo Hello World!  
restartPolicy: OnFailure
```

The schedule format follows the standard cron format. From left to right, each \* represents the following:

- Minute (0–59)
- Hour (0–23)
- Day (1–31)
- Month (1–12)
- Day of the week (0–6) (Sunday = 0, Saturday = 6)

CronJob accept step values, which allow you to create a schedule that can execute every minute, every 2 minutes, or every hour.

Our example manifest will create a CronJob that runs an image called hello-world every minute and outputs Hello World! in the Pod log.

## CSI drivers

Kubernetes uses the CsiDriver resource to connect nodes to a storage system.

You can list all CSI drivers that are available on a cluster by executing the `kubectl get csidriver` command. In one of our lab clusters, we are using NetApp's SolidFire for storage, so our cluster has the Trident CSI driver installed, as can be seen here:

```
NAME CREATED AT  
csi.trident.netapp.io 2019-09-04T19:10:47Z
```

## CSI nodes

To avoid storing storage information in the node's API resource, the CSINode resource was added to the API server to store information generated by the CSI drivers. The information that is stored includes mapping Kubernetes node names to CSI node names, CSI driver availability, and the volume topology.

## CSIStorageCapacities

CSIStorageCapacity is a component that stores information about the storage capacity for a given CSI, representing the available storage capacity for a given StorageClass. This information is used when K8s decides where to create new PersistentVolumes.

## CustomResourceDefinitions

A CustomResourceDefinition (CRD) is a way for users to make their own custom resources in a Kubernetes cluster.

It outlines the structure, format, and behavior of the custom resource, including its API endpoints and supported operations. Once a CRD is made and added to the cluster, it becomes a built-in resource type that can be managed using regular Kubernetes tools and APIs.

## DaemonSets

A DaemonSet enables the deployment of a pod on each node in a cluster or on a specific set of nodes. It is commonly utilized to deploy essential components like logging, which are required on every node in the cluster. Once a DaemonSet is set up, it automatically creates a pod on each existing node.

Moreover, as new nodes are added to the cluster, the DaemonSet ensures that a pod is deployed on the newly joined nodes.

## Deployments

We mentioned earlier that you should never deploy a pod directly. One reason for this is that you cannot scale a pod or perform a rolling upgrade when a pod is created in this way. Deployments offer you many advantages, including a way to manage your upgrades declaratively and the ability to roll back to previous revisions. Creating a Deployment is actually a three-step process that is executed by the API server: a Deployment is created, which creates a ReplicaSet, which then creates the pod(s) for the application.

Even if you don't plan to scale or perform rolling upgrades to the application, you should still use Deployments by default so that you can leverage the features at a future date.

## Endpoints

An Endpoint maps a service to a pod or pods. This will make more sense when we explain the Service resource. For now, you only need to know that you can use the CLI to retrieve endpoints by using the `kubectl get endpoints` command. In a new KinD cluster, you will see a value for the Kubernetes API server in the default namespace, as illustrated in the following code snippet:

NAMESPACE	NAME	ENDPOINTS	AGE
default	Kubernetes	172.17.0.2:6443	22h

The output shows that the cluster has a service called `kubernetes` that has an endpoint at the **Internet Protocol (IP)** address `172.17.0.2` on port `6443`. The IP that is returned in our example is the address to which our Docker control plane container has been assigned.

Later, you will see how looking at endpoints can be used to troubleshoot service and ingress issues.

## EndPointSlices

Endpoints do not scale well—they store all endpoints in a single resource. When dealing with a smaller deployment that may have a handful of pods, this isn't an issue. As clusters grow and applications scale, endpoint sizes also grow, and this will impact the performance of your control plane and cause additional network traffic as endpoints change.

EndPointSlices are designed to take on more significant and larger scenarios that require scalability and precise control over network endpoints. By default, each EndPointSlice can hold up to 100 endpoints, which can be increased by adding the `--max-endpoints-per-slice` option to the `kube-controller-manager`.

Imagine that you have a deployment of a Service in Kubernetes with a large number of pods. If one of those pods is deleted, Kubernetes will only update the specific slice that contains the information about that pod. When the updated slice is distributed across the cluster, it will only include details for a smaller subset of pods. By doing so, the cluster network remains efficient and avoids becoming overwhelmed with excessive data.

## Events

The Events resource will display any events for a namespace. To get a list of events for the `kube-system` namespace, you would use the `kubectl get events -n kube-system` command.

## FlowSchemas

Kubernetes clusters have predefined settings that manage the handling of concurrent requests to the API server, ensuring that the traffic does not overload the server. However, you have the flexibility to customize and configure your own flow schema and priority levels for requests directed at the API server in your clusters. This allows you to define specific rules and preferences for how requests are handled and prioritized, tailoring the behavior of the API server to suit your specific requirements and workload.

For example, you have a namespace that has an important application deployed. You could create a FlowSchema with a high priority so the API server would handle requests for the namespace before other requests.

## HorizontalPodAutoscalers

One of the biggest advantages of running a workload on a Kubernetes cluster is the ability to automatically scale your pods. While you can scale using the `kubectl` command or by editing a manifest's replica count, these are not automated and require manual intervention.

**Horizontal Pod Autoscalers (HPAs)** provide the ability to scale an application based on a set of criteria. Using metrics such as CPU and memory usage, or your own custom metrics, you can set a rule to scale your pods out when you need more pods to maintain your service level.

After a cooldown period, Kubernetes will scale the application back to the minimum number of pods defined in the policy.

To quickly create an HPA for an NGINX Deployment, we can execute a `kubectl` command using the `autoscale` option, as follows:

```
kubectl autoscale deployment nginx --cpu-percent=50 --min=1 --max=5
```

You can also create a Kubernetes manifest to create your HPAs. Using the same options as those we did in the CLI, our manifest would look like this:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: nginx-deployment
spec:
```

```
maxReplicas: 5
minReplicas: 1
scaleTargetRef:
  apiVersion: apps/v1
  kind: Deployment
  name: nginx-deployment
targetCPUUtilizationPercentage: 50
```

Both options will create an HPA that will scale `nginx-deployment` up to 5 replicas when the Deployment hits a CPU utilization of 50%. Once the Deployment usage falls below 50% and the cooldown period is reached (by default, 5 minutes), the replica count will be reduced to 1.

## IngressClasses

`IngressClasses` allow you to define and oversee various types of Ingress controllers. They offer the ability to personalize and adjust the behavior of these controllers according to specific needs, providing customizable routing of incoming traffic to services. `IngressClasses` allow you to manage and fine-tune Ingress controllers, ensuring that traffic is handled in a manner that aligns with your requirements.

The most important role an `IngressClass` has is to let you define multiple Ingress controllers in a single cluster. For instance, the Kubernetes Dashboard version 3 uses a specific `IngressClass` to make sure its `Ingress` objects are bound to an NGINX instance that doesn't have a `LoadBalancer`, so it can't be accessed from outside the cluster. You can also use this feature to connect Ingress controllers to different networks.

## Ingress

An `Ingress` resource is a tool that lets you create rules for incoming HTTP and HTTPS traffic to services using options like hostnames, paths, or request headers. It acts as a middleman between the external traffic and the services running in the cluster. By using `Ingress`, you can define how different types of traffic should be routed to specific services, giving you granular control over the flow of incoming requests.

We will discuss `Ingress` in depth in the next chapter, but a quick description of what `Ingress` provides is that it allows you to expose your application to the outside world using an assigned URL.

## Jobs

`Jobs` allow you to execute a specific number of executions of a pod or pods. Unlike a `CronJob` resource, these pods are not run on a set schedule, but rather they will execute once when they are created.

## LimitRanges

We will discuss the `Quota` resource later in this chapter, but a `LimitRange` is a configuration that allows you to establish and enforce specific boundaries and restrictions on resource allocations for pods and containers within a given namespace. By utilizing `LimitRanges`, you can define limits on resources, such as CPU, memory, and storage, ensuring that pods and containers operate efficiently and prevent any negative impact on the overall cluster environment.

## LocalSubjectAccessReview

LocalSubjectAccessReview is a feature that helps you check if a user or group in the cluster has the required permissions to perform a specific action on a local resource. It enables you to review access permissions directly within the cluster without relying on external API requests.

Using LocalSubjectAccessReview, you can specify the user or group identity along with the action and resource you want to evaluate. The Kubernetes API server will then verify the permissions against the local access control policies. It will respond with whether the requested action is allowed or denied.

## MutatingWebhookConfiguration

MutatingWebhookConfiguration is used to create webhooks that can intercept and modify requests sent to the API server, providing a way to automatically modify the requested resources.

A MutatingWebhookConfiguration contains a set of rules that determine which requests should be intercepted and processed by a webhook. When a request matches the defined rules, the MutatingWebhookConfiguration triggers the corresponding webhooks, which can then modify the payload. Modifications can include adding, removing, or modifying fields and annotations in the resource being created or updated.

## Namespaces

A Namespace is a resource to divide a cluster into logical units. Each Namespace allows granular management of resources, including permissions, quotas, and reporting.

The Namespace resource is used for namespace tasks, which are cluster-level operations. Using the namespace resource, you can execute commands including `create`, `delete`, `edit`, and `get`.

The syntax for the command is `kubectl <verb> ns <namespace name>`.

For example, to describe the `kube-system` namespace, we would execute a `kubectl describe namespaces kube-system` command.

This will return information for the namespace, including any labels, annotations, and assigned quotas, as illustrated in the following code snippet:

```
Name: kube-system
Labels: <none>
Annotations: <none>
Status: Active
No resource quota.
No LimitRange
resource.
```

In the preceding output, you can see that this namespace does not have any labels, annotations, or resource quotas assigned.

This section is only meant to introduce the concept of namespaces as a management unit in multi-tenant clusters. If you plan to run clusters with multiple tenants, you need to understand how namespaces can be used to secure a cluster.

## NetworkPolicies

NetworkPolicy resources let you define how network traffic, both ingress (incoming) and egress (outgoing), can flow through your cluster. They allow you to use Kubernetes native constructs to define which pods can talk to other Pods. If you've ever used security groups in **Amazon Web Services (AWS)** to lock down access between two groups of systems, it's a similar concept. As an example, the following policy will allow traffic on port 443 to pods in the `myns` namespace from any namespace with the `app.kubernetes.io/name: ingress-nginx` label on it (which is the default label for the `nginx-ingress` namespace):

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-ingress
  namespace: myns
spec:
  PodSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          app.kubernetes.io/name: ingress-nginx
  ports:
  - protocol: TCP
    port: 443
```

A NetworkPolicy is another resource that you can use to secure a cluster. They should be used in all production clusters, but in a multi-tenant cluster, they should be considered a **must-have** to secure each namespace in the cluster.

## Nodes

The nodes resource is a cluster-level resource that is used to interact with the cluster's nodes. This resource can be used with various actions including `get`, `describe`, `label`, and `annotate`.

To retrieve a list of all of the nodes in a cluster using `kubectl`, you need to execute a `kubectl get nodes` command. On a new KinD cluster running a simple one-node cluster, this would display as follows:

NAME	STATUS	ROLES	AGE	VERSION
kind-control-plane	Ready	master	22h	v1.30.0

You can also use the nodes resource to get details of a single node using the `describe` command. To get a description of the KinD node listed previously, we can execute `kubectl describe node kind-control-plane`, which would return details on the node, including consumed resources, running pods, IP classless inter-domain routing (CIDR) ranges, and more.

## PersistentVolumeClaims

A PVC is a namespaced resource that is used by a pod to consume persistent storage. A PVC uses a **persistent volume (PV)** to map the actual storage resource, which can be on any support storage system, including NFS and iSCSI.

As with most resources we have discussed, you can issue `get`, `describe`, and `delete` commands on a PVC resource. Since these are used by pods in the namespace, PVCs must be created in the same namespace as the pod(s) that will use the PVC.

## PersistentVolumes

PVs are used by PVCs to create a link between the PVC and the underlying storage system. Manually maintaining PVs is a messy, manual task, and it should be avoided. Instead, Kubernetes includes the ability to manage most common storage systems using the **Container Storage Interface (CSI)**.

Most CSI solutions that are used in an Enterprise cluster provide auto-provisioning support, as we discussed in *Chapter 2* when we introduced Rancher's local provisioner. Solutions that support auto-provisioning remove the administrative overhead that is required to create PVs manually, taking care of the creation and mapping of the PVs to PVCs automatically.

## PodDisruptionBudgets

A PodDisruptionBudget (PDB) is a resource that creates boundaries on the maximum number of unavailable pods at any given time. Its purpose is to prevent situations where multiple pods are terminated simultaneously, which could result in service disruptions or failures. By defining the minimum number of available pods, referred to as the "`minAvailable`" parameter, you can guarantee that a specific quantity of pods remains functional during maintenance or other disruptive occurrences.

The `kube-scheduler` in a cloud will use this information to figure out how to replace nodes during an upgrade. You need to be careful when using a PodDisruptionBudget because you could find a situation where an upgrade is halted.

## Pods

The pod resource is used to interact with the pods that are running your container(s). Using the `kubectl` utility, you can use commands such as `get`, `delete`, and `describe`. For example, if you wanted to get a list of all pods in the `kube-system` namespace, you would execute a `kubectl get Pods -n kube-system` command that would return all pods in the namespace, as follows:

NAME	READY	STATUS	RESTARTS	AGE
calico-kube-controllers-c6c8dc655-vnrt7	1/1	Running	0	15m
calico-node-4d9px	1/1	Running	0	15m
calico-node-r4zsj	1/1	Running	0	15m

coredns-558bd4d5db-8mxzp	1/1	Running	0	15m
coredns-558bd4d5db-fxnkt	1/1	Running	0	15m
etcd-cluster01-control-plane	1/1	Running	0	15m
kube-apiserver-cluster01-control-plane	1/1	Running	0	15m
kube-controller-manager-cluster01-control-plane	1/1	Running	0	15m
kube-proxy-npxqd	1/1	Running	0	15m
kube-proxy-twn7s	1/1	Running	0	15m
kube-scheduler-cluster01-control-plane	1/1	Running	0	15m

While you can create a pod directly, you should avoid doing so unless you are using a pod for quick troubleshooting. pods that are created directly cannot use many of the features provided by Kubernetes, including scaling, automatic restarts, or rolling upgrades.

Instead of creating a pod directly, you should use a Deployment, StatefulSet, or, in some rare cases, ReplicaSet resource or replication controller.

## PodTemplates

PodTemplates provide a way to create templates or blueprints for creating pods. They function as reusable configurations that include the desired specifications and settings for pods. They include the metadata and specifications of a pod, including the name, labels, containers, volumes, and other attributes.

PodTemplates are commonly used in other Kubernetes objects such as ReplicaSets, Deployments, and StatefulSets. These resources rely on a PodTemplate to generate and manage a collection of pods with consistent configurations and behavior.

## PriorityClasses

PriorityClasses provide a way to prioritize pods based on their importance. This allows the Kubernetes scheduler to make better decisions regarding resource allocation and pod scheduling in a cluster.

To define PriorityClasses, you create a new PriorityClass resource associated with numeric values that indicate the priority level. pods with higher priority values are given priority over lower values when it comes to resource allocation and scheduling.

Using PriorityClasses, you can guarantee that crucial workloads are given higher priority in terms of resource allocation and scheduling, providing the necessary resources to run smoothly.

## PriorityLevelConfigurations

PriorityLevelConfigurations are objects that help define priority levels for requests sent to the API server. They provide control over how API requests are processed and prioritized within a cluster. Using PriorityLevelConfigurations, you can establish multiple priority levels, assigned to specific attributes. These attributes include setting limits on the maximum number of queries per second (QPS) and concurrent requests for a particular priority level. This allows for more efficient resource management and allocation based on the importance of different API requests.

PriorityLevelConfigurations allow you to enforce policies that ensure critical requests always receive enough resources, providing flexibility in managing the processing and allocation of resources for API requests.

## ReplicaSets

ReplicaSets can be used to create a pod or a set of pods (replicas). Similar to the ReplicationController resource, a ReplicaSet will maintain the set number of pods defined in the replica count. If there are too few pods, Kubernetes will make up the difference and create the missing pods. If there are too many pods for a ReplicaSet, Kubernetes will delete pods until the number is equal to the replica count set.

In general, you should avoid creating ReplicaSets directly. Instead, you should create a Deployment, which will create and manage a ReplicaSet.

## Replication controllers

Replication controllers will manage the number of running pods, keeping the desired replicas specified running at all times. If you create a replication controller and set the replica count to 5, the controller will always keep five pods of the application running.

Replication controllers have been replaced by the ReplicaSet resource, which we just discussed in its own section. While you can still use replication controllers, you should consider using a Deployment or a ReplicaSet.

## ResourceQuotas

It is becoming very common to share a Kubernetes cluster between multiple teams, referred to as a **multi-tenant cluster**. Since you will have multiple teams working in a single cluster, you should create quotas to limit the potential of a single tenant consuming all the resources in a cluster or on a node.

Limits can be set on most cluster resources, including the following:

- Central processing unit (CPU)
- Memory
- PVCs
- ConfigMaps
- Deployments
- Pods, and more

Setting a limit will stop any additional resources from being created once the limit is hit. If you set a limit of 10 pods for a namespace and a user creates a new Deployment that attempts to start 11 Pods, the eleventh pod will fail to start up and the user will receive an error.

A basic manifest file to create a quota for memory and CPU would look like this:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: base-memory-cpu
```

```
spec:  
  hard:  
    requests.cpu: "2"  
    requests.memory: 8Gi  
    limits.cpu: "4"  
    limits.memory: 16Gi
```

This will set a limit on the total amount of resources the namespace can use for CPU and memory requests and limits.

Many of the options you can set in a quota are self-explanatory, like pods, PVCs, services, etc. When you set a limit, it means that the set limit is the maximum allowed for that resource in the namespace. For example, if you set a limit on a pod to 5, when an attempt is made to create a sixth pod in that namespace, it will be denied.

Some quotas have more than one option that can be set: specifically, CPU and memory. In our example, both resources have set a request and a limit. Both values are very important to understand to ensure efficient use of your resources and to limit the potential availability of the application.

A request is essentially a reservation of that specific resource. When a pod is deployed, you should always set a request on your CPU and memory, and the value should be the minimum required to start your application. This value will be used by the scheduler to find a node that meets the request that has been set. If there are no nodes with the requested resource available, the pod will fail to be scheduled.

Now, since a request will reserve the resource, that means once all nodes in the cluster have 100% of requests assigned, any additional pod creations will be denied since the requests are at 100%. Even if your actual cluster CPU or memory utilization is at 10%, pods will fail to be scheduled since the request, or **reservation**, is at 100%. If requests are not carefully thought out, it will lead to wasted resources, and that will lead to an increased cost to run the platform.

Limits on CPU and memory set the maximum value that the pod will be able to utilize. This is different from a request since limits are not a reservation of the resource. However, limits still need to be carefully planned out from an application side. If you set the CPU limit too low, the application may experience performance issues, and if you set the memory limit too low, the pod will be terminated, impacting availability while it is restarted.

Once a quota has been created, you can view the usage using the `kubectl describe` command. In our example, we named the ResourceQuota as `base-memory-cpu`.

To view the usage, we will execute the `kubectl get resourcequotas base-memory-cpu` command, resulting in the following output:

```
Name: base-memory-cpu  
Namespace: default  
Resource Used Hard  
-----  
limits.cpu 0 4
```

```
limits.memory 0 16Gi
requests.cpu 0 2
requests.memory 0 8Gi
```

ResourceQuotas serve as a means to manage and control the allocation of resources within a cluster. They allow you to assign specific CPU and memory resources to individual namespaces, ensuring that each tenant has sufficient resources to run their applications effectively. Additionally, ResourceQuotas act as a safeguard, preventing a poorly optimized or resource-intensive application from adversely affecting the performance of other applications in the cluster.

## RoleBindings

The RoleBinding resource is how you associate a Role or ClusterRole with a subject and namespace. For instance, the following RoleBinding will allow the `aws-codebuild` user to apply the `patch-openunison` ClusterRole to the `openunison` namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: patch-openunison
  namespace: openunison
subjects:
- kind: User
  name: aws-codebuild
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: patch-deployment
  apiGroup: rbac.authorization.k8s.io
```

Even though this references a ClusterRole, it will only apply to the `openunison` namespace. If the `aws-codebuild` user tries to patch a Deployment in another namespace, the API server will stop it.

## Roles

As with a ClusterRole, Roles combine API groups and actions to define a set of permissions that can be assigned to a subject. The difference between a ClusterRole and a Role is that a Role can only have resources defined at the namespace level and they apply only within a specific namespace.

## RuntimeClasses

RuntimeClasses are used to set up and customize different runtime environments for running containers. They provide the flexibility to choose and configure the container runtime that best suits your workloads. By using RuntimeClasses, you can fine-tune the container runtime according to your specific requirements.

Each `RuntimeClass` is linked to a specific container runtime, like Docker or Containerd. They include configurable parameters that define how the chosen container runtime behaves. These parameters include resource limits, security configurations, and environment variables.

## Secrets

Earlier, we described how to use a `ConfigMap` resource to store configuration information. We mentioned that `ConfigMap` should never be used to store any type of sensitive data. This is the job of a `Secret`.

`Secrets` are stored as Base64-encoded strings, which aren't a form of encryption. So, why separate `Secrets` from `ConfigMap`? Providing a separate resource type offers an easier way to maintain access controls and the ability to inject sensitive information using an external secret management system.

`Secrets` can be created using a file, directory, or from a literal string. As an example, we have a MySQL image we want to execute, and we would like to pass the password to the pod using a `Secret`. On our workstation, we have a file called `dbpwd` in our current working directory that has our password in it. Using the `kubectl` command, we can create a `Secret` by executing `kubectl create secret generic mysql-admin --from-file=../dbpwd`.

This would create a new `Secret` called `mysql-admin` in the current namespace, with the content of the `dbpwd` file. Using `kubectl`, we can get the output of the `Secret` by running the `kubectl get secret mysql-admin -o yaml` command, which would output the following:

```
apiVersion: v1
data:
  dbpwd: c3VwZXJzZWNyZXQtGFzc3dvcmQK
kind: Secret
metadata:
  creationTimestamp: "2020-03-24T18:39:31Z"
  name: mysql-admin
  namespace: default
  resourceVersion: "464059"
  uid: 69220ebd-c9fe-4688-829b-242fffc9e94fc
type: Opaque
```

Looking at the preceding output, you can see that the `data` section contains the name of our file and then a Base64-encoded value, which was created from the content of the file.

If we copy the Base64 value from the `Secret` and pipe it out to the `base64` utility, we can easily decode the password, as follows:

```
echo c3VwZXJzZWNyZXQtGFzc3dvcmQK | base64 -d
supersecret-password
```

When using the `echo` command to Base64-encode strings, add the `-n` flag to avoid adding an additional `\n`. Instead of `echo 'test' | base64`, use `echo -n 'test' | base64`.

Everything is stored in etcd but we are concerned that someone may be able to hack into the etcd node and steal a copy of the etcd database. Once someone has a copy of the database, they could easily use the etcdctl utility to look through the content to retrieve all of our Base64-encoded Secrets. Luckily, Kubernetes added a feature to encrypt Secrets when they are written to a database.

Enabling this feature can be fairly complex for many users, and while it sounds like a good idea, it does present some potential issues that you should consider before implementing it. If you would like to read the steps on encrypting your Secrets at rest, you can view these on the Kubernetes site at <https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/>.

Another option to secure Secrets is to use a third-party secrets management tool such as HashiCorp's Vault or CyberArk's Conjur. We'll cover integration with secret management tools in *Chapter 9, Managing Secrets in Kubernetes*.

## **SelfSubjectAccessReviews**

`SelfSubjectAccessReviews` objects enable users or entities to check their own permissions for performing specific actions on resources in their namespace.

To use `SelfSubjectAccessReviews`, users provide their username along with the desired action and resource to check. The cluster evaluates the permissions of the provided user against the access control policies in the namespace, and the API server responds with whether the requested action is allowed or denied.

`SelfSubjectAccessReviews` and the next resource, `SelfSubjectRulesReviews`, may look very similar, but they serve different functions. The main point to keep in mind for `SelfSubjectAccessReviews` is that they assess individual access permissions for specific actions on resources.

## **SelfSubjectRulesReviews**

`SelfSubjectRulesReviews` objects are used to determine the set of rules that a user or entity has permissions for within a namespace, providing the ability to investigate the access control rules for their own actions and resources.

To use a `SelfSubjectRulesReview`, you provide your identity, and the API server assesses the permissions associated with the identity in a namespace.

`SelfSubjectRulesReviews` offer a more comprehensive view over `SelfSubjectAccessReviews`, providing a deeper understanding of the entire set of rules that govern a user's permissions within a namespace.

## **Service accounts**

Kubernetes uses `ServiceAccounts` to enable access controls for workloads. When you create a `Deployment`, you may need to access other services or Kubernetes resources.

Since Kubernetes is a secure system, each resource or service your application tries to access will evaluate role-based access control (RBAC) rules to accept or deny the request.

Creating a service account using a manifest is a straightforward process, requiring only a few lines in the manifest. The following code snippet shows a service account manifest to create a service account for a Grafana Deployment:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: grafana
  namespace: monitoring
```

You combine the service account with role bindings and Roles to allow access to the required services or objects.

We'll cover how to use ServiceAccounts in depth in *Chapter 6, Integrating Enterprise Authentication into Your Cluster*.

## Services

When you create a pod, it will receive an IP address from the CIDR range that was assigned when the cluster was created. In most clusters, the assigned IPs are only addressable within the cluster itself, referred to as “**island mode**.” Since pods are ephemeral, the assigned IP address will likely change during an application’s life cycle, which becomes problematic when any service or application needs to connect to the pod. To address this, we can create a Kubernetes service, which will also receive an IP address, but since services aren’t deleted during an application’s life cycle, the address will remain the same.

A service will dynamically maintain a list of pods to target based on labels that match the service selector, creating a list of endpoints for the service.

A service stores information about how to expose the application, including which pods are running the application and the network ports to reach them.

Each service has a network type that is assigned when they are created, and they include the following:

- **ClusterIP:** A network type that is only accessible inside the cluster itself. This type can still be used for external requests using an Ingress controller, which will be discussed in a later chapter. The ClusterIP type is the default type that will be used if no type is specified when you create a service.
- **NodePort:** A network type that exposes the service to a random port between ports 30000 and 32767. This port becomes accessible by targeting any worker node in a cluster on the assigned NodePort. Once created, each node in the cluster will receive the port information, and incoming requests will be routed via kube-proxy.
- **LoadBalancer:** This type requires an add-on to use inside a cluster. If you are running Kubernetes on a public cloud provider, this type will create an external load balancer that will assign an IP address to your service. Most on-premises Kubernetes installations do not include support for the LoadBalancer type, but some offerings such as Google’s Anthos do offer support for it. In a later chapter, we will explain how to add an open-source project called MetallB to a Kubernetes cluster to provide support for the LoadBalancer type.

- **ExternalName:** This type is different from the other three. Unlike the other three options, this type will not assign an IP address to the service. Instead, this is used to map the internal Kubernetes **Domain Name System (DNS)** name to an external service.

As an example, we have deployed a pod running Nginx on port 80. We want to create a service that will allow this pod to receive incoming requests on port 80 from within the cluster. The code for this can be seen in the following snippet:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: nginx-web-frontend
    name: nginx-web
spec:
  ports:
    - name: http
      port: 80
      targetPort: 80
  selector:
    app: nginx-web
```

In our manifest, we create a label with a value of `app` and assign a value of `nginx-web-frontend`. We have called the service itself `nginx-web` and we exposed the service on port 80, targeting the pod port of 80. The last two lines of the manifest are used to assign the pods that the service will forward to, also known as Endpoints. In this manifest, any pod that has the label of `app` with a value of `nginx-web` in the namespace will be added as an endpoint to the service. Finally, you may have noticed that we didn't specify a service type in our manifest. Since we didn't specify the type, it will be created as the default service type of `ClusterIP`.

## StatefulSets

`StatefulSets` offer some unique features when creating pods. They provide features that none of the other pod creation methods offer, including the following:

- Known pod names
- Ordered Deployment and scaling
- Ordered updates
- Persistent storage creation

The best way to understand the advantages of a `StatefulSet` is to review an example manifest from the Kubernetes site, shown in the following screenshot:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3 Create Three Pods
  template:
    metadata:
      labels:
        app: nginx
    spec:
      terminationGracePeriodSeconds: 10
      containers:
        - name: nginx Name that will be used for the pods
          image: k8s.gcr.io/nginx-slim:0.8
          ports:
            - containerPort: 80
              name: web
          volumeMounts:
            - name: www Mount PVC at /usr/share/nginx/html
              mountPath: /usr/share/nginx/html
      volumeClaimTemplates:
        - metadata:
            name: www
          spec:
            accessModes: [ "ReadWriteOnce" ]
            storageClassName: nfs PVC Creation - Using the storage class named nfs
            resources:
              requests:
                storage: 1Gi
```

Figure 3.4: StatefulSet manifest example

Now, we can look at the resources that the StatefulSet created.

The manifest specifies that there should be three replicas of a pod named nginx. When we get a list of pods, you will see that three pods were created using the nginx name, with an additional dash and an incrementing number. This is what we meant in the overview when we mentioned that Pods will be created with known names, as illustrated in the following code snippet:

NAME	READY	STATUS	RESTARTS	AGE
web-0	1/1	Running	0	4m6s
web-1	1/1	Running	0	4m2s
web-2	1/1	Running	0	3m52s

The pods are also created in order—`web-0` must be fully deployed before `web-1` is created, and then, finally, `web-2`.

Finally, for this example, we also added a PVC to each pod using the `VolumeClaimTemplate` in the manifest. If you look at the output of the `kubectl get pvc` command, you will see that three PVCs were created with the names we expected (note that we removed the `VOLUME` column due to space), as illustrated in the following code snippet:

NAME	STATUS	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
www-web-0	Bound	1Gi	RWO	nfs	13m
www-web-1	Bound	1Gi	RWO	nfs	13m
www-web-2	Bound	1Gi	RWO	nfs	12m

In the `VolumeClaimTemplate` section of the manifest, you will see that we assigned the name `www` to the PVC claim. When you assign a volume in a `StatefulSet`, the PVC name will combine the name used in the claim template, combined with the name of the pod. Using this naming, you can see why Kubernetes assigned the PVC names `www-web-0`, `www-web-1`, and `www-web-2`.

## Storage classes

Storage classes are used to define a storage endpoint. Each storage class can be assigned labels and policies, allowing a developer to select the best storage location for their persistent data. You may create a storage class for a backend system that has all **Non-Volatile Memory Express (NVMe)** drives, assigning it the name `fast`, while assigning a different class to a NetApp **Network File System (NFS)** volume running standard drives, using the name `standard`.

When a PVC is requested, the user can assign a `StorageClass` that they wish to use. When the API server receives the request, it finds the matching name and uses the `StorageClass` configuration to create the volume on the storage system using a provisioner.

At a very high level, a `StorageClass` manifest does not require a lot of information. Here is an example of a storage class using a provisioner from the Kubernetes incubator project to provide NFS auto-provisioned volumes, named `nfs`:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: nfs
provisioner: nfs
```

Storage classes allow you to offer multiple storage solutions to your users. You may create a class for cheaper, slower storage while offering a second class that supports high throughput for high data requirements. By providing a different class to each offering, you allow developers to select the best choice for their application.

## SubjectAccessReviews

`SubjectAccessReviews` are used to check if an entity has permission to perform a specific action on a resource. They allow users to request access reviews and get information about their privileges. By providing an identity, desired action, and resource, the API server determines if the action is allowed or denied. This helps users verify their permissions to a resource, which can help to identify access issues to a Kubernetes resource.

For example, Scott wants to verify his ability to create pods in a namespace called `sales`. To do this, Scott creates a `SubjectAccessReview` in the `sales` namespace, including his username, the create action, and the target resource, pods.

The API server verifies whether he has permission to create pods in the `sales` namespace and sends a response back. The response from the API server includes whether the requested action is permitted or denied.

Knowing if an entity has permission to execute an action on a resource helps to minimize frustrations when a deployment fails due to permissions.

## TokenReviews

`TokenReviews` are API objects used to authenticate and verify the legitimacy of an authentication token linked to a user or entity in the cluster. If the token is valid, the API server retrieves the details about the associated user or entity.

When users submit an authentication token to the Kubernetes API server, it validates the token against the internal authentication system. It verifies that the token is legitimate and determines the user or entity associated with it.

The API server provides information about the token's validity and the user or entity, including the username, **user identifier (UID)**, and group membership.

## ValidatingWebhookConfigurations

`ValidatingWebhookConfiguration` is a collection of rules that determine what admission requests are intercepted and handled by a webhook. Each rule contains the specific resources and operations that the webhook should handle.

It provides a way to enforce specific policies or rules by applying validation logic to admission requests. Many add-ons to Kubernetes provide a `ValidatingWebhookConfiguration` – one of the most common is the NGINX ingress controller.

You can view all of the `ValidatingWebhookConfigurations` in your cluster by executing `kubectl get validatingwebhookconfigurations`. For the KinD clusters we have deployed, you will have a single entry for NGINX ingress admissions:

NAME	WEBHOOKS	AGE
ingress-nginx-admission	1	3d10h

## VolumeAttachments

VolumeAttachments create connections between external storage volumes and nodes in a cluster. They control the association of persistent volumes with specific nodes, enabling the nodes to access and utilize the storage resources.

## Summary

In this chapter, you were provided with a fast-paced Kubernetes bootcamp, where you were exposed to a wealth of technical information. Remember that as you get deeper into the world of Kubernetes, everything will become more manageable and easier to grasp. It's important to note that many of the resources discussed in this chapter will be further explored and explained in subsequent chapters, providing you with a deeper understanding.

You gained insights into each Kubernetes component and their interdependencies, which form the cluster. Armed with this knowledge, you now possess the necessary skills to investigate and identify the root causes of errors or issues within a cluster. We explored the control plane, which encompasses api-server, kube-scheduler, etcd, and controller managers. Additionally, you familiarized yourself with Kubernetes nodes that run the kubelet and kube-proxy components, along with a container runtime.

We also delved into the practical use of the kubectl utility, which will be your primary tool for interacting with a cluster. You learned about several essential commands, such as commands for accessing logs and providing descriptive information, which you will utilize on a daily basis.

In the next chapter, we will create a development Kubernetes cluster that we will use as the base cluster for the remaining chapters. Throughout the remainder of the book, we will reference many of the resources that were presented in this chapter, helping to explain them by using them in real-world examples.

## Questions

1. A Kubernetes control plane does not include which of the following components?

- a. api-server
- b. kube-scheduler
- c. etcd
- d. Ingress controller

Answer: d

2. What is the name of the component that keeps all of the cluster information?

- a. api-server
- b. Master controller
- c. kubelet
- d. etcd

Answer: d

3. Which component is responsible for selecting the node that will run a workload?
  - a. kubelet
  - b. api-server
  - c. kube-scheduler
  - d. Pod-scheduler

Answer: c

4. Which option would you add to a `kubectl` command to see additional output from a command?
  - a. Verbose
  - b. -v
  - c. -verbose
  - d. -log

Answer: b

5. Which service type creates a randomly generated port, allowing incoming traffic to any worker node on the assigned port to access the service?
  - a. LoadBalancer
  - b. ClusterIP
  - c. None—it's the default for all services
  - d. NodePort

Answer: d

6. If you need to deploy an application on a Kubernetes cluster that requires known pod names and a controlled startup of each pod, which object would you create?
  - a. StatefulSet
  - b. Deployment
  - c. ReplicaSet
  - d. ReplicationController

Answer: a

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 4

## Services, Load Balancing, and Network Policies

In the previous chapter, we kicked off our Kubernetes Bootcamp to give you a quick but thorough introduction to Kubernetes basics and objects. We started by breaking down the main parts of a Kubernetes cluster, focusing on the control plane and worker nodes. The control plane is the brain of the cluster, managing everything including scheduling tasks, creating deployments, and keeping track of Kubernetes objects. The worker nodes are used to run the applications, including components like the `kubelet` service, keeping the containers healthy, and `kube-proxy` to handle the network connections.

We looked at how you interact with a cluster using the `kubectl` tool, which lets you run commands directly or use YAML or JSON manifests to declare what you want Kubernetes to do. We also explored most Kubernetes resources. Some of the more common resources we discussed included `DaemonSets`, which ensure a pod runs on all or specific nodes, `StatefulSets` to manage stateful applications with stable network identities and persistent storage, and `ReplicaSets` to keep a set number of pod replicas running.

The Bootcamp chapter should have helped to provide a solid understanding of Kubernetes architecture, its key components and resources, and basic resource management. Having this base knowledge sets you up for the more advanced topics in the next chapters.

In this chapter, you'll learn how to manage and route network traffic to your Kubernetes services. We'll begin by explaining the fundamentals of load balancers and how to set them up to handle incoming requests to access your applications. You'll understand the importance of using service objects to ensure reliable connections to your pods, despite their ephemeral IP addresses.

Additionally, we'll cover how to expose your web-based services to external traffic using an Ingress controller, and how to use LoadBalancer services for more complex, non-HTTP/S workloads. You'll get hands-on experience by deploying a web server to see these concepts in action.

Since many readers are unlikely to have a DNS infrastructure to facilitate name resolution, which is required for Ingress to work, we will manage DNS names using a free internet service, nip.io.

Finally, we'll explore how to secure your Kubernetes services using network policies, ensuring both internal and external communications are protected.

The following topics will be covered in this chapter:

- Introduction to load balancers and their role in routing traffic.
- Understanding service objects in Kubernetes and their importance.
- Exposing web-based services using an Ingress controller.
- Using LoadBalancer services for complex workloads.
- Deploying an NGINX Ingress controller and setting up a web server.
- Utilizing the nip.io service for managing DNS names.
- Securing services with network policies to protect communications.

As this chapter ends, you will understand deeply the various methods to expose and secure your workloads in a Kubernetes cluster.

## Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, though 8 GB is suggested.
- Scripts from the `chapter4` folder from the repository, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>.

## Exposing workloads to requests

Through our experience, we've come to realize that there are three concepts in Kubernetes that people may find confusing: **Services**, **Ingress controllers**, and **LoadBalancer Services**. These are important to know in order to make your workloads accessible to the outside world. Understanding how each of these objects function and the various options you have, is crucial. So, let's start our deep dive into each of these topics.

## Understanding how Services work

As we mentioned earlier, when a workload is running in a pod, it gets assigned an IP address. However, there are situations where a pod might restart, and when that happens, it will get a new IP address. So, it's not a good idea to directly target a pod's workload because its IP address can change.

One of the coolest things about Kubernetes is its ability to scale your Deployments. When you scale a Deployment, Kubernetes adds more pods to handle the increased resource requirements. Each of these pods gets its own unique IP address. But here's the thing: most applications are designed to target only one IP address or name.

Imagine if your application went from running just one pod to suddenly running 10 pods due to scaling. How would you make use of these additional pods since you can only target a single IP address? That's what we're going to explore next.

Services in Kubernetes utilize labels to create a connection between the service and the pods handling the workload. When pods start up, they are assigned labels, and all pods with the same label, as defined in the deployment, are grouped together.

Let's take an NGINX web server as an example. In our Deployment, we would create a manifest like this:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    run: nginx-frontend
  name: nginx-frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      run: nginx-frontend
  template:
    metadata:
      labels:
        run: nginx-frontend
    spec:
      containers:
        - image: bitnami/nginx
          name: nginx-frontend
```

This deployment will create three NGINX servers and each pod will be labeled with `run=nginx-frontend`. We can verify whether the pods are labeled correctly by listing the pods using `kubectl`, and adding the `--show-labels` option, `kubectl get pods --show-labels`.

This will list each pod and any associated labels:

```
nginx-frontend-6c4dbf86d4-72cbc      1/1     Running      0
19s    pod-template-hash=6c4dbf86d4,run=nginx-frontend
nginx-frontend-6c4dbf86d4-8zlwc      1/1     Running      0
19s    pod-template-hash=6c4dbf86d4,run=nginx-frontend
nginx-frontend-6c4dbf86d4-xfz6m       1/1     Running      0
19s    pod-template-hash=6c4dbf86d4,run=nginx-frontend
```

In the example, each pod is given a label called `run=nginx-frontend`. This label plays a crucial role when configuring the service for your application. By leveraging this label in the service configuration, the service will automatically generate the required endpoints without manual intervention.

## Creating a Service

In Kubernetes, a Service is a way to make your application accessible to other programs or users. Think of it like a gateway or an entry point to your application.

There are four different types of services in Kubernetes, and each type serves a specific purpose. We will go into the details of each type in this chapter, but for now, let's take a look at them in simple terms:

Service Type	Description
ClusterIP	Creates a service that is accessible from inside of the cluster.
NodePort	Creates a service that is accessible from inside or outside of the cluster using an assigned port.
LoadBalancer	Creates a service that is accessible from inside or outside of the cluster. For external access, an additional component is required to create the load-balanced object.
ExternalName	Creates a service that does not target an endpoint in the cluster. Instead, it is used to provide a service name that targets any external DNS name as an endpoint.

Table 4.1: Kubernetes service types

There is an additional service type that can be created, known as a headless service. A Kubernetes Headless Service is a service type that enables direct communication with individual pods instead of distributing traffic across them like other services. Unlike regular Services that assign a single, fixed IP address to a group of pods, a Headless Service doesn't assign a cluster IP.

A Headless Service is created by specifying none for the clusterIP spec in the Service definition.

To create a service, you need to create a Service object that includes kind, selector, type, and any ports that will be used to connect to the service. For our NGINX Deployment example, we want to expose the Service on ports 80 and 443. We labeled the deployment with run=nginx-frontend, so when we create a manifest, we will use that name as our selector:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx-frontend
  name: nginx-frontend
spec:
  selector:
    run: nginx-frontend
  ports:
  - name: http
    port: 80
```

```
protocol: TCP
targetPort: 80
- name: https
  port: 443
  protocol: TCP
  targetPort: 443
type: ClusterIP
```

If a type is not defined in a service manifest, Kubernetes will assign a default type of `ClusterIP`.

Now that a service has been created, we can verify that it was correctly defined using a few `kubectl` commands. The first check we will perform is to verify that the service object was created. To check our service, we use the `kubectl get services` command:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
nginx-frontend	ClusterIP	10.43.142.96	<none>	80/TCP,443/TCP

After verifying that the service has been created, we can verify that the `Endpoints/Endpointslices` were created. Remember from the Bootcamp chapter that `Endpoints` are any pod that have a matching label that we used in our service. Using `kubectl`, we can verify the `Endpoints` by executing `kubectl get ep <service name>`:

NAME	ENDPOINTS
nginx-frontend	10.42.129.9:80,10.42.170.91:80,10.42.183.124:80 + 3 more...

We can see that the Service shows three `Endpoints`, but it also shows `+3 more` in the endpoint list. Since the output is truncated, the output from a `get` is limited and it cannot show all of the endpoints. Since we cannot see the entire list, we can get a more detailed list if we describe the endpoints. Using `kubectl`, you can execute the `kubectl describe ep <service name>` command:

```
Name: nginx-frontend
Namespace: default
Labels: run=nginx-frontend
Annotations: endpoints.kubernetes.io/last-change-trigger-time:
2020-04-06T14:26:08Z
Subsets:
  Addresses: 10.42.129.9,10.42.170.91,10.42.183.124
  NotReadyAddresses: <none>
  Ports:
    Name Port Protocol
    ---- - - -
    http 80   TCP
    https 443  TCP
Events: <none>
```

If you compare the output from our `get` and `describe` commands, it may appear that there is a mismatch in the `Endpoints`. The `get` command showed a total of six `Endpoints`: it showed three IP `Endpoints` and, because it was truncated, it also listed `+3`, for a total of six `Endpoints`. The output from the `describe` command shows only three IP addresses, and not six. Why do the two outputs appear to show different results?

The `get` command will list each endpoint and port in the list of addresses. Since our service is defined to expose two ports, each address will have two entries, one for each exposed port. The address list will always contain every socket for the service, which may list the endpoint addresses multiple times, once for each socket.

The `describe` command handles the output differently, listing the addresses on one line with all of the ports listed below the addresses. At first glance, it may look like the `describe` command is missing three addresses, but since it breaks the output into multiple sections, it will only list the addresses once. All ports are broken out below the address list; in our example, it shows ports `80` and `443`.

Both commands show similar data, but it is presented in a different format.

Now that the service is exposed to the cluster, you could use the assigned service IP address to connect to the application. While this would work, the address may change if the `Service` object is deleted and recreated. So, rather than targeting an IP address, you should use the DNS that was assigned to the service when it was created.

In the next section, we will explain how to use internal DNS names to resolve services.

## **Using DNS to resolve services**

So far, we have shown you that when you create certain objects in Kubernetes, the object will be assigned an IP address. The problem is that when you delete an object like a pod or service, there is a high likelihood that when you redeploy that object, it will receive a different IP address. Since IPs are transient in Kubernetes, we need a way to address objects with something other than a changing IP address. This is where the built-in DNS service in Kubernetes clusters comes in.

When a service is created, an internal DNS record is automatically generated, allowing other workloads within the cluster to query it by name. If all pods reside in the same namespace, we can conveniently access the services using a simple, short name like `mysql-web`. However, in cases where services are utilized by multiple namespaces, and workloads need to communicate with a service in a different namespace, the service must be targeted using its full name.

The following table provides an example of how a service may be accessed from various namespaces:

Cluster name: <code>cluster.local</code>	
Target Service: <code>mysql-web</code>	
Target Service Namespace: <code>database</code>	
Pod Namespace	Valid Names to Connect to the MySQL Service
<code>database</code>	<code>mysql-web</code>
<code>kube-system</code>	<code>mysql-web.database.svc</code> <code>mysql-web.database.svc.cluster.local</code>
<code>productionweb</code>	<code>mysql-web.database.svc</code> <code>mysql-web.database.svc.cluster.local</code>

*Table 4.2: Internal DNS examples*

As you can see from the preceding table, you can target a service that is in another namespace by using a standard naming convention, `.<namespace>.svc.<cluster name>`. In most cases, when you are accessing a service in a different namespace, you do not need to add the cluster name, since it should be appended automatically.

To expand on the overall concept of services, let's dive into the specifics of each service type and explore how they can be used to access our workloads.

## Understanding different service types

When you create a service, you can specify a service type, but if you do not specify a type, the `ClusterIP` type will be used by default. The service type that is assigned will configure how the service is exposed to either the cluster itself or external traffic.

### The `ClusterIP` service

The most commonly used, and often misunderstood, service type is `ClusterIP`. If you look back at *Table 4.1*, you can see that the description for the `ClusterIP` type states that the service allows connectivity to the service from within the cluster. The `ClusterIP` type does not allow any external communication to the exposed service.

The idea of exposing a service to only internal cluster workloads can be a confusing concept. In the next example, we will describe a use case where exposing a service to just the cluster itself makes sense and also increases security.

For a moment, let's set aside external traffic and focus on our current deployment. Our main goal is to understand how each component works together to form our application. Taking the NGINX example, we will enhance the deployment by adding a backend database that is used by the web server.

So far, this is a simple application: we have our deployments created, a service for the NGINX servers called `web frontend`, and a database service called `mysql-web`. To configure the database connection from the web servers, we have decided to use a `ConfigMap` that will target the database service.

You may be thinking that since we are using a single database server, we could simply use the IP address of the pod. While this would initially work, any restarts to the pod would change the address and the web servers would fail to connect to the database. Since pod IPs are ephemeral, a service should always be used, even if you are only targeting a single pod.

While we may want to expose the web server to external traffic at some point, why would we need to expose the `mysql-web` database service? Since the web server is in the same cluster, and in this case, the same namespace, we only need to use a `ClusterIP` address type so the web server can connect to the database server. Since the database is not accessible from outside of the cluster, it's more secure since it doesn't allow any traffic from outside the cluster.

By using the service name instead of the pod IP address, we will not run into issues when the pod is restarted since the service targets the labels rather than an IP address. Our web servers will simply query the **Kubernetes DNS server** for the `mysql-web` service name, which will contain the endpoints of any pod that matches the `mysql-web` label.

## The NodePort service

A `NodePort` service provides both internal and external access to your service within the cluster. At first, it may seem like the ideal choice for exposing a service since it makes it accessible to everyone. However, it achieves this by assigning a port on the node (which, by default uses a port in the range of 30000-32767). Relying on a `NodePort` can be confusing for users when they need to access a service over the network since they need to remember the specific port that was assigned to the service. You will see how you access a service on a `NodePort` shortly, demonstrating why we do not suggest using it for production workloads.

While in most enterprise environments, you shouldn't use a `NodePort` service for any production workloads, there are some valid reasons to use them, primarily, to troubleshoot accessing a workload. When we receive a call from an application that has an issue, and the Kubernetes platform or Ingress controller is being blamed, we may temporarily change the service from `ClusterIP` to `NodePort` to test connectivity without using an Ingress Controller. By accessing the application using a `NodePort`, we bypass the Ingress controller, taking that component out of the equation as a potential source causing the issue. If we are able to access the workload using the `NodePort` and it works, we know the issue isn't with the application itself and can direct engineering resources to look at the Ingress controller or other potential root causes.

To create a service that uses the `NodePort` type, you just need to set the type to `NodePort` in your manifest. We can use the same manifest that we used earlier to expose an NGINX deployment from the `ClusterIP` example, only changing type to `NodePort`:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: nginx-frontend
  name: nginx-frontend
```

```
spec:  
  selector:  
    run: nginx-frontend  
  ports:  
    - name: http  
      port: 80  
      protocol: TCP  
      targetPort: 80  
    - name: https  
      port: 443  
      protocol: TCP  
      targetPort: 443  
  type: NodePort
```

We can view the endpoints in the same way that we did for a ClusterIP service, using `kubectl`. Running `kubectl get services` will show you the newly created service:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
AGE				
nginx-frontend	NodePort	10.43.164.118	<none>	80:31574/ TCP,443:32432/TCP 4s

The output shows that the type is `NodePort` and that we have exposed the service IP address and the ports. If you look at the ports, you will notice that, unlike a `ClusterIP` service, a `NodePort` service shows two ports rather than one. The first port is the exposed port that the internal cluster services can target, and the second port number is the randomly generated port that is accessible from outside of the cluster.

Since we exposed both ports 80 and 443 for the service, we will have two `NodePorts` assigned. If someone needs to target the service from outside of the cluster, they can target any worker node with the supplied port to access the service.

10.240.100.151:31574

## Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to [nginx.org](http://nginx.org). Commercial support is available at [nginx.org](http://nginx.org).

*Thank you for using nginx*

Figure 4.1: NGINX service using NodePort

Each node maintains a list of the NodePorts and their assigned services. Since the list is shared with all nodes, you can target any functioning node using the port and Kubernetes will route it to a running pod.

To visualize the traffic flow, we have created a graphic showing the web request to our NGINX pod:

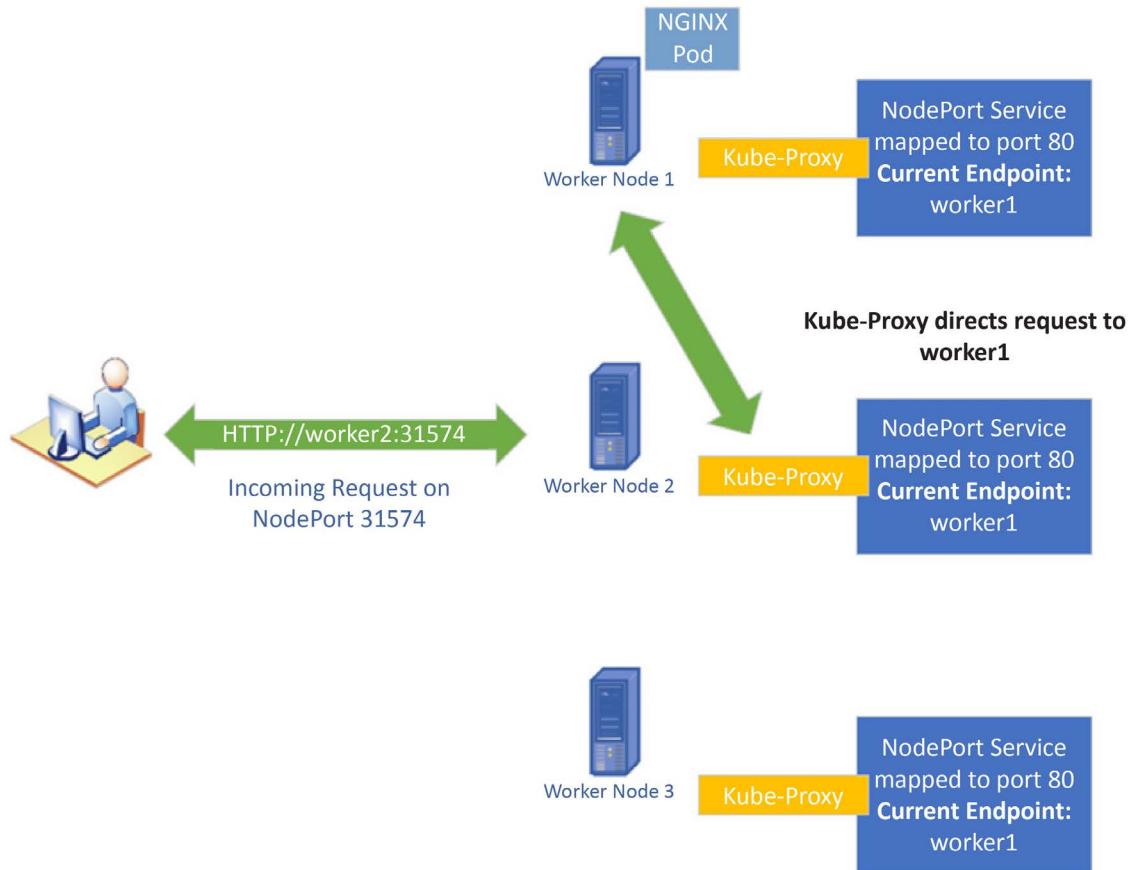


Figure 4.2: NodePort traffic flow overview

There are some issues to consider when using a NodePort to expose a service:

- If you delete and recreate the service, the assigned NodePort will change.
- If you target a node that is offline or having issues, your request will fail.
- Using NodePort for too many services may get confusing. You need to remember the port for each service and remember that there are no *external* names associated with the service. This may get confusing for users who are targeting services in the cluster.

Because of the limitations listed here, you should limit using NodePort services.

## The LoadBalancer service

Many people starting in Kubernetes read about services and discover that the LoadBalancer type will assign an external IP address to a service. Since an external IP address can be addressed directly by any machine on the network, this is an attractive option for a service, which is why many people try to use it first. Unfortunately, since many users may start by using an on-premises Kubernetes cluster, they run into failures trying to create a LoadBalancer service.

The LoadBalancer service relies on an external component that integrates with Kubernetes to create the IP address assigned to the service. Most on-premises Kubernetes installations do not include this type of service. Without the additional components, when you try to use a LoadBalancer service, you will find that your service shows `<pending>` in the EXTERNAL-IP status column.

We will explain the LoadBalancer service and how to implement it later in the chapter.

## The ExternalName service

The ExternalName service is a unique service type with a specific use case. When you query a service that uses an ExternalName type, the final endpoint is not a pod that is running in the cluster, but an external DNS name.

To use an example that you may be familiar with outside of Kubernetes, this is similar to using c-name to alias a host record. When you query a c-name record in DNS, it resolves to a host record rather than an IP address.

Before using this service type, you need to understand the potential issues that it may cause for your application. You may run into issues if the target endpoint is using SSL certificates. Since the hostname you are querying may not be the same as the name on the destination server's certificate, your connection may not succeed because of the name mismatch. If you find yourself in this situation, you may be able to use a certificate that has **subject alternative names (SANs)** added to the certificate. Adding alternative names to a certificate allows you to associate multiple names with a certificate.

To explain why you may want to use an ExternalName service, let's use the following example:

### FooWidgets application requirements

FooWidgets is running an application on their Kubernetes cluster that needs to connect to a database server running on a Windows 2019 server called `sqlserver1.foowidgets.com` (192.168.10.200).

The current application is deployed to a namespace called `finance`.



The SQL server will be migrated to a container in the next quarter.

You have two requirements:

- Configure the application to use the external database server using only the cluster's DNS server.
- FooWidgets cannot make any configuration changes to the applications after the SQL server is migrated.

Based on the requirements, using an `ExternalName` service is the perfect solution. So, how would we accomplish the requirements? (This is a theoretical exercise; you do not need to execute anything on your KinD cluster.) Here are the steps:

1. The first step is to create a manifest that will create the `ExternalName` service for the database server:

```
apiVersion: v1
kind: Service
metadata:
  name: sql-db
  namespace: finance
spec:
  type: ExternalName
  externalName: sqlserver1.foowidgets.com
```

2. With the service created, the next step is to configure the application to use the name of our new service. Since the service and the application are in the same namespace, you can configure the application to target the `sql-db` name.
3. Now, when the application queries for `sql-db`, it will resolve to `sqlserver1.foowidgets.com`, which will forward the DNS request to an external DNS server where the name is resolved to the IP address of `192.168.10.200`.

This accomplishes the initial requirement, connecting the application to the external database server using only the Kubernetes DNS server.

You may be wondering why we didn't simply configure the application to use the database server name directly. The key is the second requirement; limiting any reconfiguration when the SQL server is migrated to a container.

Since we cannot reconfigure the application once the SQL server is migrated to the cluster, we will not be able to change the name of the SQL server in the application settings. If we configured the application to use the original name, `sqlserver1.foowidgets.com`, the application would not work after the migration. By using the `ExternalName` service, we can change the internal DNS service name by replacing the `ExternalHost` service name with a standard Kubernetes service that points to the SQL server.

To accomplish the second goal, go through the following steps:

1. Since we have created a new entry in DNS for the `sql-db` name, we should delete the `ExternalName` service, since it is no longer needed.

2. Create a new service using the name `sql-db` that uses `app=sql-app` as the selector. The manifest would look like the one shown here:

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: sql-db
  name: sql-db
  namespace: finance
spec:
  ports:
    - port: 1433
      protocol: TCP
      targetPort: 1433
  name: sql
  selector:
    app: sql-app
  type: ClusterIP
```

Since we are using the same service name for the new service, no changes need to be made to the application. The app will still target the `sql-db` name, which will now use the SQL server deployed in the cluster.

Now that you know about services, we can move on to load balancers, which will allow you to expose services externally using standard URL names and ports.

## Introduction to load balancers

In this second section, we will discuss the basics between utilizing layer 7 and layer 4 load balancers. To understand the differences between the types of load balancers, it's important to understand the **Open Systems Interconnection (OSI)** model. Understanding the different layers of the OSI model will help you to understand how different solutions handle incoming requests.

## Understanding the OSI model

There are various approaches for exposing an application in Kubernetes, and you'll frequently come across mentions of layer 7 or layer 4 load balancing. These terms indicate the positions they hold in the OSI model, with each layer providing a distinct functionality. Each component that operates in layer 7 will offer different capabilities compared to those at layer 4.

To begin, let's look at a brief overview of the seven layers and a description of each. For this chapter, we are interested in the two highlighted sections, **layer 4** and **layer 7**:

OSI Layer	Name	Description
7	Application	Provides application traffic, including HTTP and HTTPS
6	Presentation	Forms data packets and encryption
5	Session	Controls traffic flow
4	Transport	Communication traffic between devices, including TCP and UDP
3	Network	Routing between devices, including IP
2	Data Link	Performs error checking for physical connection (MAC address)
1	Physical	Physical connection of devices

*Table 4.3: OSI model layers*

You don't need to be an expert in the OSI layers, but you should understand what layer 4 and layer 7 load balancers provide and how each may be used with a cluster.

Let's go deeper into the details of layers 4 and 7:

- **Layer 4:** As the description states in the chart, layer 4 is responsible for the communication of traffic between devices. Devices that run at layer 4 have access to TCP/UDP information. Load balancers that are layer-4-based provide your applications with the ability to service incoming requests for any TCP/UDP port.
- **Layer 7:** Layer 7 is responsible for providing network services to applications. When we say application traffic, we are not referring to applications such as Excel or Word; instead, we are referring to the protocols that support the applications, such as HTTP and HTTPS.

This may be very new for some people and to completely understand each of the layers would require multiple chapters – which is beyond the scope of this book. The main point we want you to take away from this introduction is that applications like databases cannot be exposed externally using a layer 7 load balancer. To expose an application that does not use HTTP/S traffic requires the use of a layer 4 load balancer.

In the next section, we will explain each load balancer type and how to use them in a Kubernetes cluster to expose your services.

## Layer 7 load balancers

Kubernetes offers Ingress controllers as layer 7 load balancers, which provide a means of accessing your applications. Various options are available for enabling Ingress in your Kubernetes clusters, including the following:

- NGINX
- Envoy
- Traefik
- HAProxy

You can think of a layer 7 load balancer as a traffic director for networks. Its role is to distribute incoming requests to multiple servers hosting a website or application.

When you access a website or use an app, your device sends a request to the server asking for the specific web page or data you want. With a layer 7 load balancer, your request doesn't directly reach a single server, instead, it sends the traffic through the load balancer. The layer 7 load balancer examines the content of your request and understands what web page or data is being requested. Using factors like backend server health, current workload, and even your location, the load balancer intelligently selects the best servers to handle your request.

A layer 7 load balancer ensures that all servers are utilized efficiently, and users receive a smooth and responsive experience. Think of this like being at a store that has multiple checkout counters where a store manager guides customers to the least busy checkout, minimizing waiting times and ensuring everyone gets served promptly.

To recap, layer 7 load balancers optimize the overall system performance and reliability.

## Name resolution and layer 7 load balancers

To handle layer 7 traffic in a Kubernetes cluster, you deploy an Ingress controller. Ingress controllers are dependent on incoming names to route traffic to the correct service. This is much easier and faster than in a legacy server deployment model where you would need to create a DNS entry and map it to an IP address before users could access the application externally by name.

Applications that are deployed on a Kubernetes cluster are no different—the users will use an assigned DNS name to access the application. The most common implementation is to create a new wildcard domain that will target the Ingress controller via an external load balancer, such as an F5, HAProxy, or Seesaw. A wildcard domain will direct all traffic for a given domain to the same destination. For example, if your wildcard domain name is `foowidgets.com`, your main entry in the domain would be `*.foowidgets.com`. Any ingress URL name that is assigned using the wildcard domain will have the traffic directed to the external load balancer, where it will be directed to the defined service using your ingress rule URL.

Using the `foowidgets.com` domain as an example, we have three Kubernetes clusters, fronted by an external load balancer with multiple Ingress controller endpoints. Our DNS server would have entries for each cluster, using a wildcard domain that points to the load balancer's virtual IP address:

Domain Name	IP Address	K8s Cluster
<code>*.cluster1.foowidgets.com</code>	192.168.200.100	Production001
<code>*.cluster2.foowidgets.com</code>	192.168.200.101	Production002
<code>*.cluster3.foowidgets.com</code>	192.168.200.102	Development001

Table 4.4: Example of wildcard domain names for Ingress

The following diagram shows the entire flow of the request:

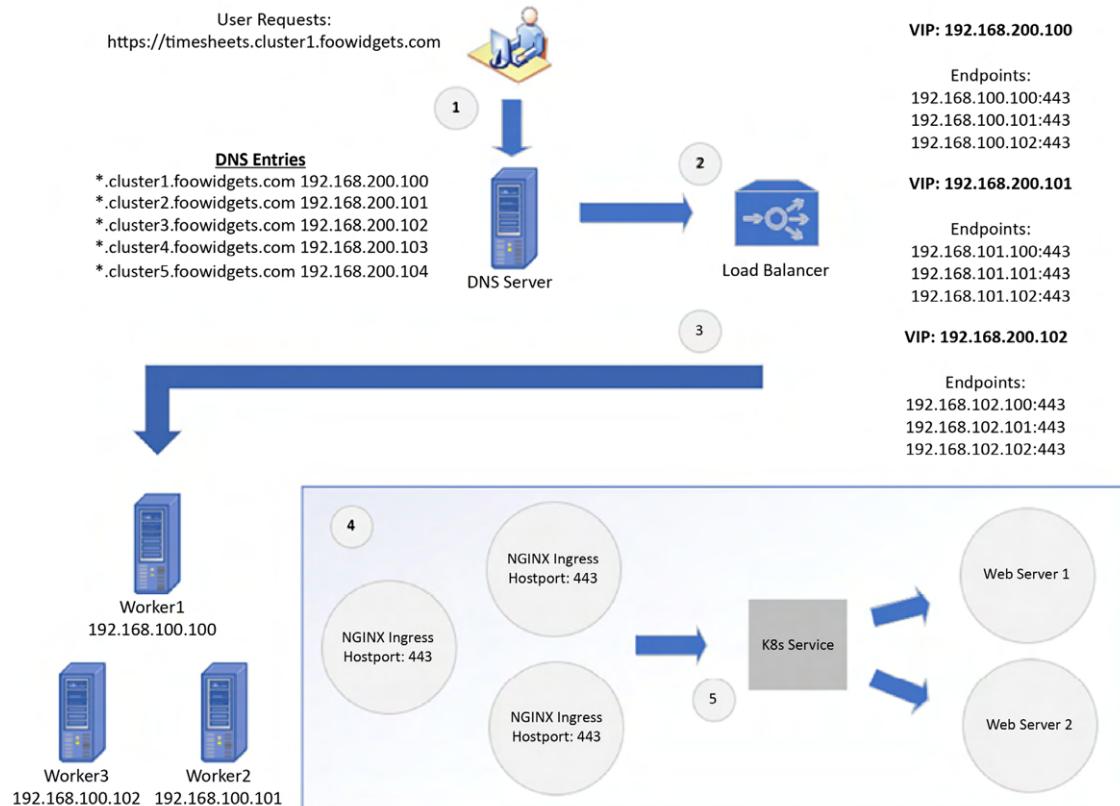


Figure 4.3: Multiple-name Ingress traffic flow

Each of the steps in *Figure 4.3* is detailed here:

1. Using a browser, the user requests this URL: `https://timesheets.cluster1.foowidgets.com`.

2. The DNS query is sent to a DNS server. The DNS server looks up the zone details for `cluster1.foowidgets.com`. There is a single entry in the DNS zone that resolves to the virtual IP (VIP) address, assigned on the load balancer for the domain.
3. The load balancer's VIP for `cluster1.foowidgets.com` has three backend servers assigned, pointing to three worker nodes where we have deployed Ingress controllers.
4. Using one of the endpoints, the request is sent to the Ingress controller.
5. The Ingress controller will compare the requested URL to a list of Ingress rules. When a matching request is found, the Ingress controller will forward the request to the service that was assigned to the Ingress rule.

To help reinforce how Ingress works, it will help to create Ingress rules on a cluster to see them in action. Right now, the key takeaway is that ingress uses the requested URL to direct traffic to the correct Kubernetes services.

## Using nip.io for name resolution

Many personal development clusters, such as our KinD installation, may not have access to a DNS infrastructure or the necessary permissions to add records. To test Ingress rules, we need to target unique hostnames that are mapped to Kubernetes services by the Ingress controller. Without a DNS server, you need to create a localhost file with multiple names pointing to the IP address of the Ingress controller.

For example, if you deployed four web servers, you would need to add all four names to your local hosts. An example of this is shown here:

```
192.168.100.100 webserver1.test.local  
192.168.100.100 webserver2.test.local  
192.168.100.100 webserver3.test.local  
192.168.100.100 webserver4.test.local
```

This can also be represented on a single line rather than multiple lines:

```
192.168.100.100 webserver1.test.local webserver2.test.local webserver3.test.  
local webserver4.test.local
```

If you use multiple machines to test your deployments, you will need to edit the host file on every machine that you plan to use for testing. Maintaining multiple files on multiple machines is an administrative nightmare and will lead to issues that will make testing a challenge.

Luckily, there are free DNS services available that we can use without configuring a complex DNS infrastructure for our KinD cluster.

nip.io is the service that we will use for our KinD cluster name resolution requirements. Using our previous web server example, we will not need to create any DNS records. We still need to send the traffic for the different servers to the NGINX server running on `192.168.100.100` so that Ingress can route the traffic to the appropriate service. nip.io uses a naming format that includes the IP address in the hostname to resolve the name to an IP. For example, say that we have four web servers that we want to test called `webserver1`, `webserver2`, `webserver3`, and `webserver4`, with Ingress rules on an Ingress controller running on `192.168.100.100`.

As we mentioned earlier, we do not need to create any records to accomplish this. Instead, we can use the naming convention to have nip.io resolve the name for us. Each of the web servers would use a name with the following naming standard:

```
<desired name>.<INGRESS IP>.nip.io
```

The names for all four web servers are listed in the following table:

Web Server Name	Nip.io DNS Name
webserver1	webserver1.192.168.100.100.nip.io
webserver2	webserver2.192.168.100.100.nip.io
webserver3	webserver3.192.168.100.100.nip.io
webserver4	webserver4.192.168.100.100.nip.io

Table 4.5: nip.io example domain names

When you use any of the preceding names, nip.io will resolve them to 192.168.100.100. You can see an example ping for each name in the following screenshot:

```
[root@localhost /]# ping webserver1.192.168.100.100.nip.io
PING webserver1.192.168.100.100.nip.io (192.168.100.100) 56(84) bytes of data.
[root@localhost /]# ping webserver2.192.168.100.100.nip.io
PING webserver2.192.168.100.100.nip.io (192.168.100.100) 56(84) bytes of data.
[root@localhost /]# ping webserver3.192.168.100.100.nip.io
PING webserver3.192.168.100.100.nip.io (192.168.100.100) 56(84) bytes of data.
[root@localhost /]# ping webserver4.192.168.100.100.nip.io
PING webserver4.192.168.100.100.nip.io (192.168.100.100) 56(84) bytes of data.
```

Figure 4.4: Example name resolution using nip.io

Keep in mind that Ingress rules require unique names to properly route traffic to the correct service. Although knowing the IP address of the server might not be required in some scenarios, it becomes essential for Ingress rules. Each name should be unique and typically uses the first part of the full name. In our example, the unique names are webserver1, webserver2, webserver3, and webserver4.

By providing this service, nip.io allows you to use any name for Ingress rules without the need to have a DNS server in your development cluster.

Now that you know how to use nip.io to resolve names for your cluster, let's explain how to use a nip.io name in an Ingress rule.

## Creating Ingress rules

Remember, ingress rules use names to route the incoming request to the correct service.

The following is a graphical representation of an incoming request showing how Ingress routes the traffic:

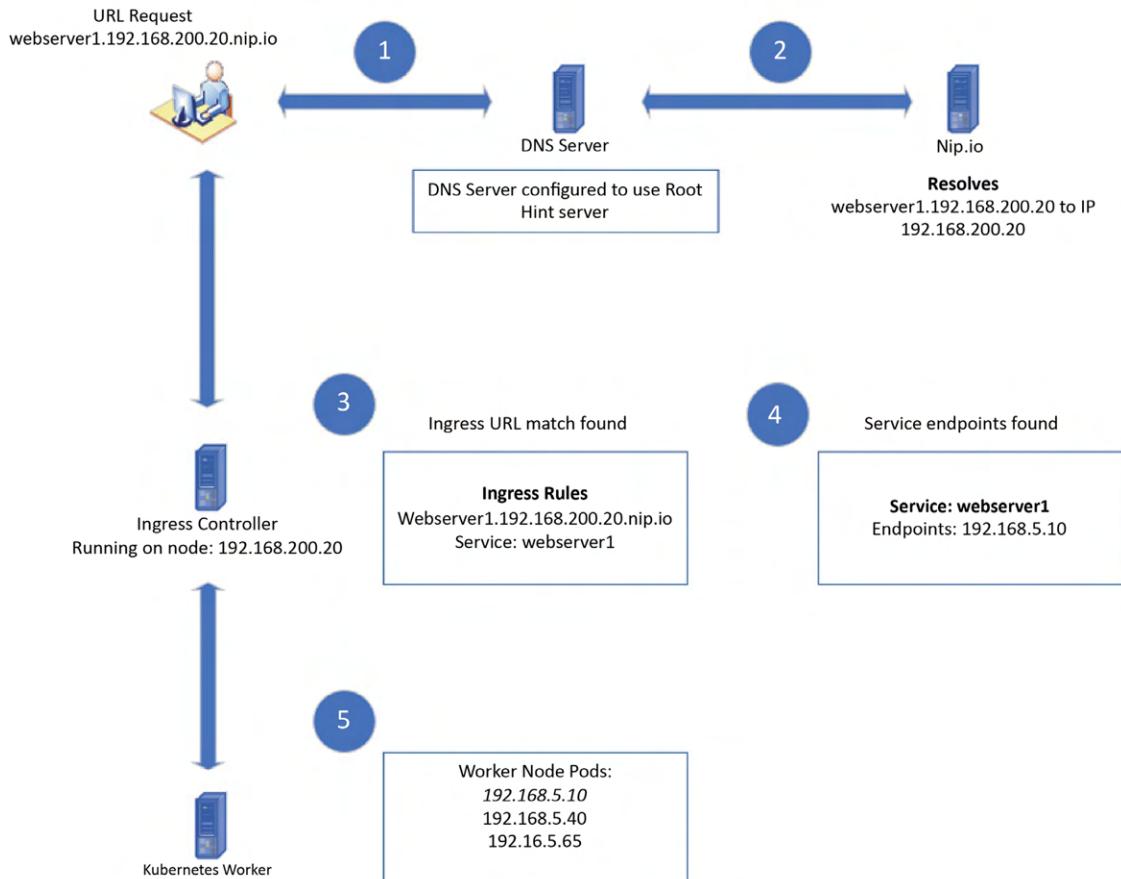


Figure 4.5: Ingress traffic flow

Figure 4.5 shows a high-level overview of how Kubernetes handles incoming Ingress requests. To help explain each step in more depth, let's go over the five steps in greater detail. Using the graphic provided in Figure 4.5, we will explain each numbered step in detail to show how ingress processes the request:

1. The user requests a URL in their browser named `http://webserver1.192.168.200.20.nip.io`. A DNS request is sent to the local DNS server, which is ultimately sent to the `nip.io` DNS server.
2. The `nip.io` server resolves the domain name to the `192.168.200.20` IP address, which is returned to the client.
3. The client sends the request to the Ingress controller, which is running on `192.168.200.20`. The request contains the complete URL name, `webserver1.192.168.200.20.nip.io`.
4. The Ingress controller looks up the requested URL name in the configured rules and matches the URL name to a service.
5. The service endpoints will be used to route traffic to the assigned pods.
6. The request is routed to an endpoint pod running the web server.

Using the preceding example traffic flow, let's create an NGINX pod, service, and Ingress rule to see this in action. In the `chapter4/ingress` directory, we have provided a script called `nginx-ingress.sh`, which will deploy the web server and expose it using an ingress rule of `webserver.w.x.y.nip.io`. When you execute the script, it will output the complete URL you can use to test the ingress rule.

The script will execute the following steps to create our new NGINX deployment and expose it using an ingress rule:

1. A new NGINX deployment called `nginx-web` is deployed, using port `8080` for the web server.
2. We create a service, called `nginx-web`, using a `ClusterIP` service (the default) on port `8080`.
3. The IP address of the host is discovered and used to create a new ingress rule that will use the hostname `webserver.w.x.y.z.nip.io`. The `w.x.y.z` web server will be replaced with the IP address of your host.

Once deployed, you can test the web server by browsing to it from any machine on your local network using the URL that is provided by the script. In our example, the host's IP address is `192.168.200.20`, so our URL will be `webserver.192.168.200.20.nip.io`.



Figure 4.6: NGINX web server using nip.io for Ingress

With the details provided in this section, it is possible to generate ingress rules for multiple containers utilizing unique hostnames. It's important to note that you aren't restricted to using a service like `nip.io` for name resolution; you can employ any name resolution method that is accessible in your environment. In a production cluster, you would typically have an enterprise DNS infrastructure. However, in a lab environment, like our KinD cluster, `nip.io` serves as an excellent tool for testing scenarios that demand accurate naming conventions.

Since we will use `nip.io` naming standards throughout the book, so it's important to understand the naming convention before moving on to the next chapter.

## Resolving Names in Ingress Controllers

As discussed earlier, Ingress controllers are primarily level 7 load balancers and are mostly concerned with HTTP/S. How does an Ingress controller get the name of the host? You might think it's included in the network requests, but it isn't. A DNS name is used by the client, but at the networking layer, there are no names, only IP addresses.

So, how does the `Ingress` controller know what host you want to connect to? It depends on whether you're using HTTP or HTTPS. If you're using HTTP, your `Ingress` controller will get the hostname from the `Host` HTTP header. For instance, here's a simple request from an HTTP client to a cluster:

```
GET / HTTP/1.1
Host: k8sou.apps.192-168-2-14.nip.io
User-Agent: curl/7.88.1
Accept: */*
```

The second line tells the `Ingress` controller which host, and which Service, you want the request to go to. This is trickier with HTTPS because the connection is encrypted and the decryption needs to happen before you can read the `Host` header.

You'll find that when using HTTPS, your `Ingress` controller will serve different certificates based on which Service you want to connect to, also based on hostnames. In order to route without yet having access to the `Host` HTTP header, your `Ingress` controller will use a protocol called **Server Name Indication (SNI)**, which includes the requested hostname as part of the TLS key exchange. Using SNI, your `Ingress` controller is able to determine which `Ingress` configuration object applies to a request before the request is decrypted.

## Using Ingress Controllers for non-HTTP traffic

The use of SNI offers an interesting side effect, which means that `Ingress` controllers can sort of pretend to be level 4 load balancers when using TLS. Most `Ingress` controllers offer a feature called TLS passthrough, where instead of decrypting the traffic, the `Ingress` controller simply routes it to a Service based on the request's SNI. Using our earlier example of a web server's backend database, if you were to configure your `Ingress` object with a TLS passthrough annotation (which is different for each controller) you could then expose your database through your `Ingress`.

Given how easy it is to create `Ingress` objects, you may think this is a security issue. That's why so much of this book is dedicated to security. It's quite easy to misconfigure your environment!

A major disadvantage to using TLS passthrough, outside of potential security issues, is that you lose many of your `Ingress` controller's native routing and control functions. For instance, if you're deploying a web application that maintains its own session state, you generally will configure your `Ingress` object to use sticky sessions so that each user's request goes back to the same container. This is accomplished by embedding cookies into HTTP responses, but if the controller is just passing the traffic through, it can't do that.

Layer 7 load balancers, like NGINX Ingress, are commonly deployed for various workloads, including web servers. However, other deployments might require a more sophisticated load balancer, operating at a lower layer of the OSI model. As we move down the model, we gain access to additional lower-level features that certain workloads require.

Before moving on to layer 4 load balancers, if you deployed the NGINX example on your cluster, you should delete all of the objects before moving on. To easily remove the objects, you can execute the `nginx-ingress-remove.sh` script in the `chapter4/ingress` directory. This script will delete the deployment, service, and ingress rule.

## Layer 4 load balancers

Similar, to layer 7 load balancers, a layer 4 load balancer is also a traffic controller for a network, but with a number of differences compared to a layer 7 load balancer.

The layer 7 load balancer understands the content of incoming requests, making decisions based on specific information like web pages or data being requested. A layer 4 load balancer works at a lower level, looking at the basic information contained in the incoming network traffic, such as IP addresses and ports, without inspecting the actual data.

When you access a website or use an app, your device sends a request to the server with a unique IP address and a specific port number – also called a **socket**. The layer 4 load balancer observes this address and port to efficiently distribute incoming traffic across multiple servers. To help visualize how layer 4 load balancers work, think of it as a traffic cop that efficiently directs incoming cars to different lanes on a highway. The load balancer doesn't know the exact destination or purpose of each car; it just looks at their license plate numbers and directs them to the appropriate lane to ensure smooth traffic flow.

In this way, the layer 4 load balancer ensures that the servers receive a fair share of incoming requests and that the network operates efficiently. It's an essential tool to make sure that websites and applications can handle a large number of users without getting overwhelmed, helping to maintain a stable and reliable network.

There are lower-level networking operations in the process that are beyond the scope of this book. HAProxy has a good summary of the terminology and example configurations on its website at <https://www.haproxy.com/fr/blog/loadbalancing-faq/>.

In summary, a layer 4 load balancer is a network tool that distributes incoming traffic based on IP addresses and port numbers, allowing websites and applications to perform efficiently and deliver a seamless user experience.

## Layer 4 load balancer options

There are multiple options available to you if you want to configure a layer 4 load balancer for a Kubernetes cluster. Some of the options include the following:

- HAProxy
- NGINX Pro
- Seesaw
- F5 Networks
- MetalLB

Each option provides layer 4 load balancing, but for the purpose of this book, we will use **MetalLB**, which has become a popular choice for providing a layer 4 load balancer to a Kubernetes cluster.

## Using MetalLB as a layer 4 load balancer

Remember that in *Chapter 2, Deploying Kubernetes Using KinD*, we had a diagram showing the flow of traffic between a workstation and the KinD nodes. Because KinD was running in a nested Docker container, a layer 4 load balancer would have had certain limitations when it came to networking connectivity. Without additional network configuration on the Docker host, you will not be able to target the services that use the LoadBalancer type outside of the Docker host itself. However, if you deploy MetalLB to a standard Kubernetes cluster running on a host, you will not be limited to accessing services outside of the host itself.

MetalLB is a free, easy-to-configure layer 4 load balancer. It includes powerful configuration options that give it the ability to run in a development lab or an enterprise cluster. Since it is so versatile, it has become a very popular choice for clusters requiring layer 4 load balancing.

We will focus on installing MetalLB in layer 2 mode. This is an easy installation and works for development or small Kubernetes clusters. MetalLB also offers the option to deploy using BGP mode, which allows you to establish peering partners to exchange networking routes. If you would like to read about MetalLB's BGP mode, you can read about it on MetalLB's site at <https://metallb.universe.tf/concepts/bgp/>.

### Installing MetalLB

Before we deploy MetalLB to see it in action, we should start with a new cluster. While this isn't required, it will limit any issues from any resources you may have been testing from previous chapter. To delete the cluster and redeploy a fresh cluster, follow the steps below:

1. Delete the cluster using the `kind delete` command.

```
kind delete cluster --name cluster01
```

2. To redeploy a new cluster, change your directory to the `chapter2` directory where you cloned the repo
3. Create a new cluster using the `create-cluster.sh` in the root of the `chapter2` directory
4. Once deployed, change your directory to the `chapter4/metallb` directory

We have included a script called `install-metallb.sh` in the `chapter4/metallb` directory. The script will deploy MetalLB v0.13.10 using a pre-built configuration file called `metallb-config.yaml`. Once completed, the cluster will have the MetalLB components deployed, including the controller and the speakers.

The script, which you can look at, to understand what each step does by looking at the comments, execute the following steps to deploy MetalLB in your cluster:

1. MetalLB is deployed into the cluster. The script will wait until the MetalLB controller is fully deployed.
2. The script will find the IP range used on the Docker network. These will be used to create two different pools to use for LoadBalancer services.

3. Using the values for the address pools, the script will inject the IP ranges into two resources - `metallb-pool.yaml` and `metallb-pool-2.yaml`.
4. The first pool is deployed using `kubectl apply` and it also deploys the `L2Advertisement` resource.
5. The script will show the pods from the MetalLB namespace to confirm they have been deployed.
6. Finally, a NGINX web server pod will be deployed called `nginx-1b` and a LoadBalancer service to provide access to the deployment using a MetalLB IP address.

MetalLB resources like address pools and the `L2Advertisement` resource will be explained in the upcoming sections.

If you want to read about the available options when you deploy MetalLB, you can visit the installation page on the MetalLB site: <https://metallb.universe.tf/installation/>.

Now that MetalLB has been deployed to the cluster, let's explain the MetalLB configuration file that configures how MetalLB will handle requests.

## Understanding MetalLB's custom resources

MetalLB is configured using two custom resources that contain MetalLB's configuration. We will be using MetalLB in layer 2 mode, and we will create two custom resources: the first is for the IP address range called `IPAddressPool` and the second configures what pools are advertised, known as an `L2Advertisement` resource.

The OSI model and the layers may be new to many readers – layer 2 refers to the layer of the OSI model; it plays a crucial role in enabling communication within a local network. It's the layer where devices determine how to utilize the network infrastructure, like ethernet cables, and establish how to identify other devices. Layer 2 only deals with the local network segment; it doesn't handle the task of directing traffic between different networks. That is the responsibility of layer 3 (the network layer) in the OSI model.

To put it simply, you can view layer 2 as the facilitator for devices within the same network to communicate. It achieves this by assigning MAC addresses (unique addresses) to devices and providing a method for sending and receiving data, which are organized into network packets. We have provided pre-configured resources in the `chapter4/metallb` directory called `metallb-pool.yaml` and `L2Advertisement.yaml`. These files will configure MetalLB in layer 2 mode with an IP address range that is part of the Docker network, which will be advertised through the `L2Advertisement` resource.

To keep the configuration simple, we will use a small range from the Docker subnet in which KinD is running. If you were running MetalLB on a standard Kubernetes cluster, you could assign any range that is routable in your network, but we are limited in how KinD clusters deal with network traffic.

Let's get into the details of how we created the custom resources. To begin, we need the IP range we want to advertise, and for our KinD cluster, that means we need to know what network range Docker is using. We can get the subnet by inspecting the KinD bridge network that KinD uses, using the `docker network inspect` command:

```
docker network inspect kind | grep -i subnet
```

In the output, you will see the assigned subnet, similar to the following:

```
"Subnet": "172.18.0.0/16"
```

This is an entire **Class B** address range. We know that we will not use all of the IP addresses for running containers, so we will use a small range from the subnet in our MetalLB configuration.

 Note: The term **Class B** is a reference to how IP addresses are divided into classes to define the range and structure of addresses for different network sizes. The primary classes are **Class A**, **Class B**, and **Class C**. Each class has a specific range of addresses and is used for different purposes.

These classes help organize and allocate IP addresses efficiently, ensuring that networks of different sizes have appropriate address spaces. For private networks, which are networks not directly connected to the internet, each class has a specific IP range reserved for this internal use:

- Class A Private Range: 10.0.0.0 to 10.255.255.255
- Class B Private Range: 172.16.0.0 to 172.31.255.255
- Class C Private Range: 192.168.0.0 to 192.168.255.255

Understanding subnets and class ranges is very important but it is beyond the scope of this book. If you are new to TCP/IP, you should consider reading about subnetting and class ranges.

If we look at our `metallb-pool.yaml` configuration file, we will see the configuration for `IPAddressPool`:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: pool-01
  namespace: metallb-system
spec:
  addresses:
    - 172.18.200.100-172.18.200.125
```

This manifest defines a new `IPAddressPool` called `pool-01` in the `metallb-system` namespace, with an IP range set to `172.18.200.100 – 172.18.200.125`.

`IPAddressPool` only defines the IP addresses that will be assigned to `LoadBalancer` services. To advertise the addresses, you need to associate the pools with an `L2Advertisement` resource. In the `chapter4/metallb` directory, we have a pre-defined `L2Advertisement` called `l2advertisement.yaml`, which is linked to the address pool we created, as shown here:

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2-all-pools
  namespace: metallb-system
```

When examining the preceding manifest, you might notice that there is minimal configuration involved. As we mentioned earlier, `IPAddressPool` needs to be associated with `L2Advertisement`, but in our current configuration, we haven't specified any linking to the address pool we created. So, the question now is, how will our `L2Advertisement` announce or make use of the `IPAddressPool` we've created?

If you do not specify any pools in an `L2Advertisement` resource, each `IPAddressPool` that is created will be exposed. However, if you had a scenario where you only needed to advertise a few address pools, you could add the pool names to the `L2Advertisement` resource so that only the assigned pools would be advertised. For example, if we had three pools named `pool1`, `pool2`, and `pool3` in a cluster, and we only wanted to advertise `pool1` and `pool3`, we would create an `L2Advertisement` resource like the following example:

```
apiVersion: metallb.io/v1beta1
kind: L2Advertisement
metadata:
  name: l2-all-pools
  namespace: metallb-system
spec:
  ipAddressPools:
    - pool1
    - pool3
```

With configuration out of the way, we will move on to explain how MetalLB's components interact to assign IP addresses to services.

## MetalLB components

Our deployment, which uses the standard manifest provided by the MetalLB project, will create a `Deployment` that will install the MetalLB controller and a `DaemonSet` that will deploy the second component to all nodes, called the speaker.

### The Controller

The controller will receive announcements from the speaker on each worker node. These announcements show each service that has requested a `LoadBalancer` service, showing the assigned IP address that the controller assigned to the service:

```
{"caller":"main.go:49","event":"startUpdate","msg":"start of service update","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.437701161Z"}  
{"caller":"service.go:98","event":"ipAllocated","ip":"10.2.1.72","msg":"IP address assigned by controller","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.438079774Z"}  
{"caller":"main.go:96","event":"serviceUpdated","msg":"updated service object","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.467998702Z"}
```

In the preceding example output, a Service called `my-grafana-operator/grafana-operator-metrics` has been deployed and MetalLB has assigned the IP address `10.2.1.72`.

## The Speaker

The speaker component is what MetalLB uses to announce the `LoadBalancer` service's IPs to the local network. This component runs on each node and ensures that the network configuration and the routers in your network are aware of the IP addresses assigned to the `LoadBalancer` services. This allows the `LoadBalancer` to receive traffic on its assigned IP address without needing additional network interface configurations on each node.

The speaker component in MetalLB is responsible for telling the local network how to access the services you've set up within your Kubernetes cluster. Think of it as the messenger that tells other devices on the network about the route they should take to send data meant for your applications.

It is primarily responsible for four tasks:

- **Service detection:** When a service is created in Kubernetes, the speaker component is always watching for `LoadBalancer` services.
- **IP address management:** The speaker is in charge of managing IP addresses. It decides which IP addresses should be assigned to make the services accessible to external communication.
- **Route announcements:** After MetalLB's speaker identifies the services that require external access and assigns the IP addresses, it communicates the route throughout your local network. It provides instructions to the network on how to connect to the services using the designated IP addresses.
- **Load balancing:** MetalLB performs network load balancing. If you have multiple pods, which all applications should, the speaker will distribute incoming network traffic among the pods, ensuring that the load is balanced for performance and reliability.

By default, it is deployed as a `DaemonSet` for redundancy – regardless of how many speakers are deployed, only one is active at any given time. The main speaker will announce all `LoadBalancer` service requests to the controller and if that speaker pod experiences a failure, another speaker instance will take over the announcements.

If we look at the speaker log from a node, we can see announcements, similar to the following example:

```
{"caller":"main.go:176","event":"startUpdate","msg":"start of service update","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.437231123Z"} {"caller":"main.go:189","event":"endUpdate","msg":"end of service update","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.437516541Z"} {"caller":"main.go:176","event":"startUpdate","msg":"start of service update","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.464140524Z"} {"caller":"main.go:246","event":"serviceAnnounced","ip":"10.2.1.72","msg":"service has IP, announcing","pool":"default","protocol":"layer2","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.464311087Z"} {"caller":"main.go:249","event":"endUpdate","msg":"end of service update","service":"my-grafana-operator/grafana-operator-metrics","ts":"2020-04-21T21:10:07.464470317Z"}
```

The preceding announcement is for a Grafana component. In the announcement, you can see that the service has been assigned an IP address of 10.2.1.72 – this announcement will also go to the MetalLB controller, as we showed in the previous section.

Now that you have installed MetalLB and understand how the components create the services, let's create our first LoadBalancer service on our KinD cluster.

## Creating a LoadBalancer service

In the layer 7 load balancer section, we created a deployment running NGINX that we exposed by creating a service and an Ingress rule. At the end of the section, we deleted all of the resources to prepare for this test. If you followed the steps in the Ingress section and have not deleted the service and Ingress rule, please do so before creating the LoadBalancer service.

The MetalLB deployment script included an NGINX server with a LoadBalancer service. It will create an NGINX Deployment with a LoadBalancer service on port 80. The LoadBalancer service will be assigned an IP address from our defined pool, and since it's the first service to use the address pool, it will likely be assigned 172.18.200.100.

You can test the service by using curl on the Docker host. Using the IP address that was assigned to the service, enter the following command:

```
curl 172.18.200.100
```

You will receive the following output:

```
surovich@kind3:~$ curl 172.18.200.100
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

Figure 4.7: Curl output to the LoadBalancer service running NGINX

Adding MetalLB to a cluster allows you to expose applications that otherwise could not be exposed using a layer 7 balancer. Adding both layer 7 and layer 4 services to your clusters allows you to expose almost any application type you can think of, including databases.

In the next section, we will explain some of the advanced options that are available to create advanced IPAddressPool configurations.

## Advanced pool configurations

The MetalLB IPAddressPool resource offers a number of advanced options that are useful in different scenarios, including the ability to disable automatic assignments of addresses, use static IP addresses and multiple address pools, scope a pool to a certain namespace or service, and handle buggy networks.

### Disabling automatic address assignments

When a pool is created, it will automatically start to assign addresses to any service that requests a LoadBalancer type. While this is a common implementation, you may have special use cases where a pool should only assign an address if it is explicitly requested.

### Assigning a static IP address to a service

When a service is assigned an IP address from the pool, it will keep the IP until the service is deleted and recreated. Depending on the number of LoadBalancer services being created, it is possible that the same IP address could be assigned when it is re-created, but there is no guarantee and we have to assume that the IP may change.

If we have an add-on like `external-dns`, which will be covered in the next chapter, you may not care that the IP address changes on a service since you would be able to use a name that is registered with the assigned IP address. In some scenarios, you may have little choice in deciding whether you can use the IP or name for a service and may experience issues if the address were to change during a redeployment.

As of the time of this writing, Kubernetes includes the ability to assign an IP address that a service will be assigned by adding `spec.loadBalancerIP` to the service resource, with the desired IP address. By using this option, you can “statically” assign the IP address to your service and if the service is deleted and redeployed, it will stay the same. This becomes useful in multiple scenarios, including the ability to add the known IP to other systems like **Web Application Firewalls (WAFs)** and firewall rules.

Starting in Kubernetes 1.24, the `loadBalancerIP` spec has been deprecated and while it will work in Kubernetes 1.27, the field may be removed in a future K8s release. Since the option will be removed at some point, it is suggested to use a solution that is included in the layer 4 load balancer you have deployed. In the case of MetalLB, they have added an annotation to assign an IP called `metallb.universe.tf/loadBalancerIPs`. Setting this field to the desired IP address will accomplish the same goal of using the deprecated `spec.loadBalancerIP`.

You may be thinking that assigning a static IP may come with some potential risks like conflicting IP assignments, which cause connectivity issues. Luckily, MetalLB has some features to mitigate these potential risks. If MetalLB is not the owner of the requested address or if the address is already being utilized by another service, the IP assignment will fail. If this failure occurs, MetalLB will generate a warning event, which can be viewed by running the `kubectl describe service <service name>` command.

The following manifest shows how to use both the native Kubernetes `loadBalancerIP` and MetalLB’s annotation to assign a static IP address to a service. The first example shows the deprecated `spec.loadBalancerIP`, assigning an IP address of `172.18.200.210` to the service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-web
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: nginx-web
  type: LoadBalancer
  loadBalancerIP: 172.18.200.210
```

The following example shows how to set MetalLB’s annotation to assign the same IP address:

```
apiVersion: v1
```

```
kind: Service
metadata:
  name: nginx-web
  annotations:
    metallb.universe.tf/loadBalancerIPs: 172.18.200.210
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: nginx-web
  type: LoadBalancer
```

The next section will discuss how to add additional address pools to your MetalLB configuration and how to use the new pools to assign an IP address to a service.

## Using multiple address pools

In our original example, we created a single node pool for our cluster. It's not uncommon to have a single address pool for a cluster, but in a more complex environment, you may need to add additional pools to direct traffic to a certain network, or you may need to simply add an additional pool due to simply running out of address in your original pool.

You can create as many address pools as you require in a cluster. We assigned a handful of addresses in our first pool, and now we need to add an additional pool to handle the number of workloads on the cluster. To create a new pool, we simply need to deploy a new `IPAddressPool`, as shown in the following:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: pool-02
  namespace: metallb-system
spec:
  addresses:
  - 172.18.201.200-172.18.201.225
```



The current release of MetalLB will require a restart of the MetalLB controller for the new address pool to be available.

Notice the name of this pool is `pool-01`, with a range of `172.18.201.200 – 172.18.201.225`, whereas our original pool was `pool-01` with a range of `172.18.200.200 – 172.18.200.225`. Since we have deployed an `L2Advertisement` resource that exposes `IPAddressPools`, we do not need to create anything for the new pool to be announced.

Now that we have two active pools in our cluster, we can use a MetalLB annotation called `metallb.universe.tf/address-pool` in a service to assign the pool we want to pull an IP address from, as shown in the following example:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-web
  annotations:
    metallb.universe.tf/address-pool: pool-02
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: nginx-web
  type: LoadBalancer
```

If we deploy this service manifest and then look at the services in the namespace, we will see that it has been assigned an IP address from the new pool, `pool-02`:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
nginx-lb-pool02	LoadBalancer	10.96.52.153	172.18.201.200	80:30661/TCP
3m8s				

Our cluster now offers `LoadBalancer` services the option of using either `pool-01` or `pool-02`, based on the workload requirements.

You may be wondering how multiple address pools work if a service request does not explicitly define which pool to use. This is a great question, and we can control that by setting a value, known as a priority, to an address pool when created, defining the order of the pool that will assign the IP address.

Pools are a powerful feature, offering a highly configurable and flexible solution to provide the appropriate IP address pools to specific services.

MetalLB's flexibility doesn't stop with address pools. You may find that you have a requirement to create a pool that only a certain namespace or namespaces are allowed to use. This is called **IP pool scoping** and in the next section, we will discuss how to configure a scope to limit a pool's usage based on a namespace.

When multiple `IPAddressPools` are available, MetalLB determines the availability of IPs by sorting the matching pools based on their priorities. The sorting starts with the highest priority (lowest priority number) and then proceeds to lower priority pools. If multiple `IPAddressPools` have the same priority, MetalLB selects one of them randomly. If a pool lacks a specific priority or is set to 0, it is considered the lowest priority and is used for assignment only when pools with defined priorities cannot be utilized.

In the following example, we have created a new pool called pool-03 and set a priority of 50 and another pool called pool-04 with a priority of 70:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: pool-03
  namespace: metallb-system
spec:
  addresses:
    - 172.168.210.0/24
  serviceAllocation:
    priority: 50
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: pool-04
  namespace: metallb-system
spec:
  addresses:
    - 172.168.211.0/24
  serviceAllocation:
    priority: 70
```

If you create a service without selecting a pool, the request will match both of the pools shown previously. Since pool-03 has a lower priority number, it has a higher priority and will be used before pool-04 unless the pool is out of address, which will cause the request to use an IP from the pool-04 address pool.

As you can see, pools are powerful and flexible, providing a number of options to address different workload requirements. We have discussed how to select the pool using annotation and how different pools with priorities work. In the next section, we will discuss how we can link a pool to certain namespaces, limiting the workloads that can request an IP address from certain address pools.

## IP pool scoping

Multitenant clusters are common in enterprise environments and, by default, a MetalLB address pool is available to any deployed LoadBalancer service. While this may not be an issue for many organizations, you may need to limit a pool, or pools, to only certain namespaces to limit what workloads can use certain address pools.

To scope an address pool, we need to add some fields to our IPAddressPool resource. For our example, we want to deploy an address pool that has the entire Class C range available to only two namespaces, web and sales:

```
apiVersion: metallb.io/v1beta1
```

```
kind: IPAddressPool
metadata:
  name: ns-scoped-pool
  namespace: metallb-system
spec:
  addresses:
    - 172.168.205.0/24
  serviceAllocation:
    priority: 50
  namespaces:
    - web
    - sales
```

When we deploy this resource, the only services that can request an address from the pool must exist in either the `web` or `sales` namespaces. If a request is made from any other namespace for `ns-scoped-pool`, it will be denied and an IP address in the `172.168.205.0` range will not be assigned to the service.

The last option we will discuss in the next section is known as handling buggy networks.

## Handling buggy networks

MetalLB has a field that some networks may require to handle IP blocks ending in either `.0` or `.255`. Older networking devices may flag the traffic as a possible **Smurf** attack, blocking the traffic. If you happen to run into this scenario, you will need to set the `AvoidBuggyIPs` field in the `IPAddressPool` resource to `true`.



At a high level, a **Smurf** attack sends a large number of network messages to special addresses that will reach all computers on the network. The traffic makes all computers think that the traffic is coming from a specific address, causing all of the computers to send a response to that specific machine. This traffic results in a **denial-of-service** attack, causing the machine to go offline and disrupting any services that were running.

To avoid this issue, setting the `AvoidBuggyIPs` field will prevent the `.0` and `.255` addresses from being used. An example manifest is shown here:

```
apiVersion: metallb.io/v1beta1
kind: IPAddressPool
metadata:
  name: buggy-example-pool
  namespace: metallb-system
spec:
  addresses:
    - 172.168.205.0/24
  avoidBuggyIPs: true
```

Adding MetalLB as a layer 4 load balancer to your cluster allows you to migrate applications that may not work with simple layer 7 traffic.

As more applications are migrated or refactored for containers, you will run into many applications that require multiple protocols for a single service. In the next section, we will explain some scenarios where having multiple protocols for a single service is required.

## Using multiple protocols

Earlier versions of Kubernetes did not allow services to assign multiple protocols to a LoadBalancer service. If you attempted to assign both TCP and UDP to a single service, you would receive an error that multiple protocols were not supported and the resource would fail to deploy.

Although MetalLB still provides support for this, there's little incentive to utilize those annotations since newer versions of Kubernetes introduced an alpha feature gate called `MixedProtocolLBService` in version 1.20. It has since graduated to general availability starting in Kubernetes version 1.26, making it a base feature that enables the use of different protocols for LoadBalancer-type services when multiple ports are defined.

Using a `CoreDNS` example, we need to expose our `CoreDNS` to the outside world. We will explain a use case in the next chapter where we need to expose a `CoreDNS` instance to the outside world using both TCP and UDP.

Since DNS servers use both TCP and UDP port 53 for certain operations, we need to create a service that will expose our service as a `LoadBalancer` type, listening to both TCP and UDP port 53. Using the following example, we create a new service that has both TCP and UDP defined:

```
apiVersion: v1
kind: Service
metadata:
  name: coredns-ext
  namespace: kube-system
spec:
  selector:
    k8s-app: kube-dns
  ports:
    - name: dns-tcp
      port: 53
      protocol: TCP
      targetPort: 53
    - name: dns-udp
      port: 53
      protocol: UDP
      targetPort: 53
  type: LoadBalancer
```

If we deployed the manifest and then looked at the services in the `kube-system` namespace, we would see that the service was created successfully and that both port 53 on TCP and UDP have been exposed:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
coredns-ext	LoadBalancer	10.96.192.4	172.18.200.101	53:31889/
TCP, 53:31889/UDP	6s			

You will see that the new service was created, `coredns-ext`, assigned the IP address of `172.18.200.101`, and exposed on TCP and UDP port 53. This will now allow the service to accept connections on both protocols using port 53.

One issue that many load balancers have is that they do not provide name resolution for the service IPs. Users prefer to target an easy-to-remember name rather than random IP addresses when they want to access a service. Kubernetes does not provide the ability to create externally accessible names for services, but there is an incubator project to enable this feature. In *Chapter 5*, we will explain how we can provide external name resolution for Kubernetes services.

In the final section of the chapter, we will discuss how to secure our workloads using network policies.

## Introducing Network Policies

Security is something that all Kubernetes users should think about from day 1. By default, every pod in a cluster can communicate with any other pod in the cluster, even other namespaces that you may not own. While this is a basic Kubernetes concept, it's not ideal for most enterprises, and when using multi-tenant clusters, it becomes a big security concern. We need to increase the security and isolation of workloads, which can be a very complex task, and this is where network policies come in.

NetworkPolicies provide users the ability to control their network traffic for both egress and ingress using a defined set of rules between pods, namespaces, and external endpoints. Think of a network policy as a firewall for your clusters, providing fine-grained access controls based on various parameters. Using network policies, you can control which pods are allowed to communicate with other pods, restrict traffic to specific protocols or ports, and enforce encryption and authentication requirements.

Like most Kubernetes objects that we have discussed, network policies allow control based on labels and selectors. By matching the labels specified in a network policy, Kubernetes can determine which pods and namespaces should be allowed or denied network access.

Network policies are an optional feature in Kubernetes, and the CNI being used in the cluster must support them to be used. On the KinD cluster we created, we deployed **Calico**, which does support network policies, however, not all network plugins support network policies out of the box, so it's important to plan out your requirements before deploying a cluster.

In this section, we will explain the options provided by network policies to enhance the overall security of your applications and cluster.

## Network policy object overview

Network policies provide a number of options to control both ingress and egress traffic. They can be granular to only allow certain pods, namespaces, or even IP addresses to control the network traffic.

There are four parts to a network policy. Each part is described in the following table.

Spec	Description
podSelector	This limits the scope of workloads that a policy is applied to, using a label selector. If no selector is provided, the policy will affect every pod in the namespace.
policyTypes	This defines the policy rules. The valid types are ingress and egress.
ingress	(optional) This defines the rules to follow for ingress traffic. If there are no rules defined, it will match all incoming traffic.
egress	(optional) This defines the rules to follow for egress traffic. If there are no rules defined, it will match all outgoing traffic.

Table 4.6: Parts of a network policy

The ingress and egress portions of the policy are optional. If you do not want to block any egress traffic, simply omit the egress spec. If a spec is not defined, all traffic will be allowed.

### The podSelector

The podSelector field is used to tell you what workloads a network policy will affect. If you wanted the policy to only affect a certain deployment, you would define a label that would match a label in the deployment. The label selectors are not limited to a single entry; you can add multiple label selectors to a network policy, but all selectors must match for the policy to be applied to the pod. If you want the policy to be applied to all pods, leave the podSelector blank; it will apply the policy to every pod in the namespace.

In the following example, we have defined that the policy will only be applied to pods that match the label app=frontend:

```
spec:
  podSelector:
    matchLabels:
      app: frontend
```

The next field is the type of policy, which is where you define a policy for ingress and egress.

### The policyTypes

The policyType field specifies the type of policy being defined, determining the scope and behavior of the NetworkPolicy. There are two available options for policyType:

policyType	Description
ingress	ingress controls incoming network traffic to pods. It defines the rules that control the sources that are allowed to access the pods matching the podSelector specified in the NetworkPolicy. Traffic can be allowed from specific IP CIDR ranges, namespaces, or from other pods within the cluster.
egress	egress controls outgoing network traffic from pods. It defines rules that control the destinations that pods are allowed to communicate with. Egress traffic can be restricted by specific IP CIDR ranges, namespaces, or other pods within the cluster.

Table 4.7: The policy types

Policies can contain ingress, egress, or both options. If one policy type is not included, it will not affect that traffic type. For example, if you only include an ingress policyType, all egress traffic will be allowed at any location on the network.

As we mentioned, when you create a rule for either ingress or egress traffic, you can provide no label, a single label, or multiple labels that must match for the policy to take effect. The following example shows an ingress block that has three labels; in order for the policy to affect a workload, all three declared fields must match:

```
ingress:
  - from:
    - ipBlock:
        cidr: 192.168.0.0/16
    - namespaceSelector:
        matchLabels:
          app: backend
    - podSelector:
        matchLabels:
          app: database
```

In the preceding example, any incoming traffic needs to be coming from a workload that has an IP address in the 192.168.0.0 subnet, in the namespace that has a label of app=backend, and finally, the requesting pod must have a label of app=database.

While the example shows options for ingress traffic, the same options are available for egress traffic.

Now that we have covered the options that are available in a network policy, let's move on to creating a full policy using a real-world example.

## Creating a Network Policy

We have included a network policy script in the chapter4/netpol directory in the book's GitHub repository called netpol.sh. When you execute the script, it will create a namespace called sales, with a few pods with labels and a network policy. The policy that is created will be the basis for the policy we will go over in this section.

When you create a network policy, you need to have an understanding of the desired network restrictions. The person who is most aware of the application traffic flow is best suited to help create a working policy. You need to consider the pods or namespaces that should be able to communicate, which protocols and ports should be allowed, and whether you need any additional security like encryption or authentication.

Like other Kubernetes objects, you need to create a manifest using the `NetworkPolicy` object and provide metadata like the name of the policy and the namespace that it will be deployed in.

Let's use an example where you have a backend pod running PostgreSQL in the same namespace as a web server. We know that the only pod that needs to talk to the database server is the web server itself and no other communication should be allowed to the database.

To begin, we need to create our manifest, and it will start out by declaring the API, kind, policy name, and namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-netpol
  namespace: sales
```

The next step is to add the pod selector to specify the pods that will be affected by the policy. This is done by creating a `podSelector` section where you define selectors based on any pods with matching labels. For our example, we want our policy to apply to pods that are part of the backend database application. The pods for the application have all been labeled with `app=backend-db`:

```
spec:
  podSelector:
    matchLabels:
      app.kubernetes.io/name: postgresql
```

Now that we have declared what pods to match, we need to define the `ingress` and `egress` rules, which are defined with the `spec.ingress` or `spec.egress` section of the policy. For each rule type, you can set the allowed protocols and ports for the application, and control from where an external request is allowed to access the port. To build on our example, we want to add an `ingress` rule that will allow port 5432 from pods with a label of `app=backend`:

```
spec:
  podSelector:
    matchLabels:
      app: frontend
  ingress:
  - from:
    - podSelector:
```

```

matchLabels:
  app: frontend
ports:
- protocol: TCP
  port: 5432

```

As the last step, we will define our policy type. Since we are only concerned with incoming traffic to the PostgreSQL pod, we only need to declare one type, `ingress`:

```

policyTypes:
- Ingress

```

Once this policy is deployed, the pods running in the `sales` namespace that have an `app=backend-db` label will only receive traffic from pods that have a label of `app=frontend` on TCP port 5432. Any request other than port 5432 from a frontend pod will be denied. This policy makes our PostgreSQL deployment very secure since any incoming traffic is tightly locked down to a specific workload and TCP port.

When we execute the script from the repository, it will deploy PostgreSQL to the cluster and add a label to the deployment. We are going to use the label to tie the network policy to the PostgreSQL pod. To test the connectivity, and the network policy, we will run a `netshoot` pod and use `telnet` to test connecting to the pod on port 5432.

We need to know the IP address to test the network connection. To get the IP for the database server, we just need to list the pods in the namespace using the `-o wide` option, to list the IP of the pods. Now that PostgreSQL is running, we will simulate a connection by running a `netshoot` pod with a label that doesn't match `app: frontend`, which will result in a failed connection. See the following:

```

kubectl get pods -n sales -o wide
NAME        READY   STATUS    RESTARTS   AGE     IP
db-postgresql-0   0/1     Running   0          45s   10.240.189.141

kubectl run tmp-shell --rm -i --tty --image nicolaka/netshoot --labels
app=wronglabel -n sales
tmp-shell ~ telnet 10.240.189.141:5432

```

The connection will eventually time out since the incoming request has a pod labeled `app=wronglabel`. The policy will look at the labels from the incoming request and if none of them match, it will deny the connection.

Finally, let's see whether we created our policy correctly. We will run `netshoot` again, but this time with the correct label, we will see that the connection succeeds:

```

kubectl run tmp-shell --rm -i --tty --image nicolaka/netshoot --labels
app=frontend -n sales
tmp-shell ~ telnet 10.240.189.141:5432
Connected to 10.240.189.141:5432

```

Notice the line, which says `Connected to 10.240.189.141:5432`. This shows that the PostgreSQL pod accepted the incoming request from the netshoot pod since the label for the pod matches the network policy, which is looking for a label of `app=frontend`.

So, why does the network policy allow only port 5432? We didn't set any options to deny traffic; we defined only the allowed traffic. Network policies follow a default deny-all for any policy that isn't defined. In our example, we only defined port 5432, so any request that is not on port 5432 will be denied. Having a deny-all for any undefined communication helps to secure your workload by avoiding any unintended access.

If you wanted to create a deny-all network policy, you would just need to create a new policy that has `ingress` and `egress` added, with no other values. An example is shown as follows:

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

In this example, we have set `podSelector` to `{}`, which means the policy will apply to all pods in the namespace. In the `spec.ingress` and `spec.egress` options, we haven't set any values, and since the default behavior is to deny any communication that doesn't have a rule, this rule will deny all ingress and egress traffic.

## Tools to create network policies

Network policies can be difficult to create manually. It can be challenging to know what ports you need to open, especially if you aren't the application owner.

In *Chapter 13, KubeArmor Securing Your Runtime*, we will discuss a tool called **KubeArmor**, which is a CNCF project that was donated by a company called **AccuKnox**. KubeArmor was primarily a tool to secure container runtimes, but recently they added the ability to watch the network traffic flow between pods. By watching the traffic, they know the “normal behavior” of the network connections for the pod and it creates a `ConfigMap` for each observed network policy in the namespace.

We will go into details in *Chapter 13*; for now, we just wanted to let you know that there are tools to help you create network policies.

In the next chapter, we'll learn how to use **CoreDNS** to create service name entries in DNS using an incubator project called `external-dns`. We will also introduce an exciting CNCF sandbox project called **K8GB** that provides a cluster with Kubernetes' native global load-balancing features.

## Summary

In this chapter, you learned how to expose your workloads in Kubernetes to other cluster resources and external traffic.

The first part of the chapter went over services and the multiple types that can be assigned. The three major service types are `ClusterIP`, `NodePort`, and `LoadBalancer`. Remember that the selection of the type of service will configure how your application is exposed.

In the second part, we introduced two load balancer types, layer 4 and layer 7, each having a unique functionality for exposing workloads. You will often use a `ClusterIP` service along with an ingress controller to provide access to services that use layer 7. Some applications may require additional communication, not provided by a layer 7 load balancer. These applications may require a layer 4 load balancer to expose their services externally. In the load balancing section, we demonstrated the installation and use of `MetallLB`, a popular, open-source, layer 4 load balancer.

We closed out the chapter by discussing how to secure both ingress and egress network traffic. Since Kubernetes, by default, allows communication between all pods in a cluster, most enterprise environments need a way to secure the communication between workloads to only the required traffic for the application. Network policies are a powerful tool to secure a cluster and limit the traffic flow for both incoming and outgoing traffic.

You may still have some questions about exposing workloads, such as the following: how can we handle DNS entries for services that use a `LoadBalancer` type? Or, maybe, how do we make a deployment highly available between two clusters?

In the next chapter, we will expand on using the tools that are useful for exposing your workloads, like name resolution and global load balancing.

## Questions

1. How does a service know what pods should be used as endpoints for the service?

- a. By the service port
- b. By the namespace
- c. By the author
- d. By the selector label

Answer: d

2. What `kubectl` command helps you troubleshoot services that may not be working properly?

- a. `kubectl get services <service name>`
- b. `kubectl get ep <service name>`
- c. `kubectl get pods <service name>`
- d. `kubectl get servers <service name>`

Answer: b

3. All Kubernetes distributions include support for services that use the LoadBalancer type.
  - a. True
  - b. False

Answer: b

4. Which load balancer type supports all TCP/UDP ports and accepts traffic regardless of the packet's contents?
  - a. Layer 7
  - b. Cisco layer
  - c. Layer 2
  - d. Layer 4

Answer: d



# 5

## External DNS and Global Load Balancing

In this chapter, we will build on what you learned in *Chapter 4*. We will discuss some of the limitations of certain load balancer features and how we can configure a cluster to resolve those limitations.

We know that Kubernetes has a built-in DNS server that dynamically allocates names to resources. These are used for applications to communicate intra-cluster, or within the cluster. While this feature is beneficial for internal cluster communication, it doesn't provide DNS resolution for external workloads. Since it does provide DNS resolution, why do we say it has limitations?

In the previous chapter, we used a dynamically assigned IP address to test our `LoadBalancer` service workloads. While our examples have been good for learning, in an enterprise, nobody wants to access a workload running on a cluster using an IP address. To address this limitation, the Kubernetes SIG has developed a project called `ExternalDNS`, which provides the ability to dynamically create DNS entries for our `LoadBalancer` services.

Also, in an enterprise, you will commonly run services that run on multiple clusters to provide failover for your applications. So far, the options we have discussed can't address failover scenarios. In this chapter, we will explain how to implement a solution to provide an automated failover for workloads, making them highly available across multiple clusters.

In this chapter, you will learn the following:

- An introduction to external DNS resolution and global load balancing
- Configuring and deploying ExternalDNS in a Kubernetes cluster
- Automating DNS name registration
- Integrating ExternalDNS with an enterprise DNS server
- Using GSLB to offer global load balancing across multiple clusters

Now, let's jump into the chapter!

## Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, though 8 GB is recommended
- A KinD cluster running **MetallLB** – if you completed *Chapter 4*, you should already have a cluster running MetalLB
- The scripts from the `chapter5` folder from the repository, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Making service names available externally

As we mentioned in the introduction, you may remember that we used IP addresses to test the `LoadBalancer` services we created, whereas for our `Ingress` examples, we used domain names. Why did we have to use IP addresses instead of a hostname for our `LoadBalancer` services?

Although a Kubernetes load balancer assigns a standard IP address to a service, it doesn't automatically generate a DNS name for workloads to access the service. Instead, you must rely on IP addresses to connect to applications within the cluster, which becomes confusing and inefficient. Furthermore, manually registering DNS names for each IP assigned by **MetallLB** presents a maintenance challenge. To deliver a more cloud-like experience and streamline name resolution for `LoadBalancer` services, we need an add-on that can address these limitations.

Similar to the team that maintains KinD, there is a Kubernetes SIG that is working on this feature for Kubernetes called **ExternalDNS**. The main project page can be found on the SIG's GitHub at <https://github.com/kubernetes-sigs/external-dns>.

At the time of writing, the `ExternalDNS` project supports 34 compatible DNS services, including the following:

- AWS Cloud Map
- Amazon's Route 53
- Azure DNS
- Cloudflare
- CoreDNS
- Google Cloud DNS
- Pi-hole
- RFC2136

There are multiple options on how to extend CoreDNS to resolve external names, depending on what main DNS server you may be running. Many of the supported DNS servers will simply register any services dynamically. `ExternalDNS` will see the created resource and use native calls to register services automatically, like **Amazon's Route 53**. Not all DNS servers natively allow for this type of dynamic registration by default.

In these instances, you need to manually configure your main DNS server to forward the desired domain requests to a CoreDNS instance running in your cluster. This is what we will use for the examples in this chapter.

Our Kubernetes cluster currently utilizes CoreDNS to handle cluster DNS name resolution. However, what might be lesser known is that CoreDNS offers more than just internal cluster DNS resolution. It can extend its capabilities to perform external name resolution, effectively resolving names for any DNS zone managed by a CoreDNS deployment.

Now, let's move on to how ExternalDNS installs.

## Setting up ExternalDNS

Right now, our CoreDNS is only resolving names for internal cluster names, so we need to set up a zone for our new LoadBalancer DNS entries.

For our example, a company, **FooWidgets**, wants all Kubernetes services to go into the `foowidgets.k8s` domain, so we will use that as our new zone.

## Integrating ExternalDNS and CoreDNS

ExternalDNS is not an actual DNS server; instead, it is a controller that will watch for objects that request a new DNS entry. Once a request is seen by the controller, it will send the information to an actual DNS server, like CoreDNS, for registration.

The process of how a service is registered is shown in the diagram below.

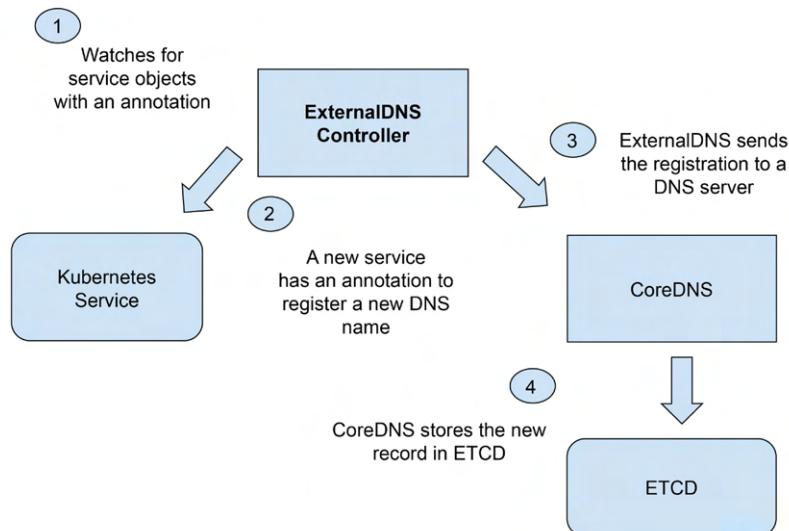


Figure 5.1: ExternalDNS registration flow

In our example, we are using CoreDNS as our DNS server; however, as we mentioned previously, ExternalDNS has support for 34 different DNS services and the list of supported services is constantly growing. Since we will be using CoreDNS as our DNS server, we will need to add a component that will store the DNS records. To accomplish this, we need to deploy an ETCD server in the cluster.

For our example deployment, we will use the ETCD Helm chart.

Helm is a tool for Kubernetes that makes it easier to deploy and manage applications. It uses Helm charts, which are templates that contain the necessary configuration and resource values for the application. It automates the setup of complex applications, ensuring they are consistently and reliably configured. It's a powerful tool, and you will find that many projects and vendors offer their applications, by default, using Helm charts. You can read more about Helm on their main home page at <https://v3.helm.sh/>.

One reason Helm is such a powerful tool is its ability to use custom options that can be declared when you run the `helm install` command. The same options can also be declared in a file that is passed to the installation using the `-f` option. These options make deploying complex systems easier and reproducible since the same values file can be used on any deployment.

For our deployment example, we have included a `values.yaml` file, located in the `chapter5/etc` directory, that we will use to configure our ETCD deployment.

Now, finally, let's deploy ETCD! We have included a script called `deploy-etcd.sh` in the `chapter5/etc` directory that will deploy ETCD with a single replica in a new namespace called `etcd-dns`. Execute the script while in the `chapter5/etc` directory.

Thanks to Helm, the script only has two commands— it will create a new namespace, and then execute a `helm install` command to deploy our ETCD instance. In the real world, you would want to change the replica count to at least 3 to have a highly available ETCD deployment, but we wanted to limit resource requirements for our KinD server.

Now that we have our ETCD for DNS, we can move on to integrating our CoreDNS service with our new ETCD deployment.

## **Adding an ETCD zone to CoreDNS**

As we showed in the diagram in the last section, CoreDNS will store the DNS records in an ETCD instance. This requires us to configure a CoreDNS server with the DNS zone(s) we want to register names in and the ETCD server that will store the records.

To keep resource requirements lower, we will use the included CoreDNS server that most Kubernetes installations include as part of their base cluster creation for our new domain. In the real world, you should deploy a dedicated CoreDNS server to handle just ExternalDNS registrations.

At the end of this section, you will execute a script to deploy a fully configured ExternalDNS service that has all of the options and configurations discussed in this section. The commands used in this section are only for reference; you do not need to execute them on your cluster, since the script will do that for you.

Before we can integrate CoreDNS, we need to know the IP address of our new ETCD service. You can retrieve the address by listing the services in the etcd-dns namespace using kubectl:

```
kubectl get svc etcd-dns -n etcd-dns
```

This will show our ETCD service, along with the IP address of the service:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
etcd-dns	ClusterIP	10.96.149.223	<none>	2379/TCP, 2380/TCP	4m

The Cluster-IP you see in the service list will be used to configure the new DNS zone as a location to store the DNS records.

When you deploy ExternalDNS, you can configure CoreDNS in one of two ways:

- Add a zone to the Kubernetes-integrated CoreDNS service
- Deploy a new CoreDNS service that will be used for ExternalDNS registrations

For ease of testing, we will add a zone to the Kubernetes CoreDNS service. This requires us to edit the CoreDNS ConfigMap found in the kube-system namespace. When you execute the script at the end of this section, the modification will be done for you. It will add the section shown below in **bold** to the ConfigMap.

```
apiVersion: v1
data:  Corefile: |      .:53 {
    errors          health {
        lameduck 5s      }
    ready           kubernetes cluster.local in-addr.arpa ip6.arpa {
        pods insecure          fallthrough in-addr.arpa ip6.arpa
        ttl 30          }
        prometheus :9153      forward . /etc/resolv.conf      etcd
foowidgets.k8s {
    stubzones          path /skydns          endpoint http://10.96.149.
223:2379          }
    cache 30          loop          reload          loadbalance      }

```

The added lines configure a zone called foowidgets.k8s that is ETCD integrated. The first line that we added tells CoreDNS that the zone name, foowidgets.com, is integrated with an ETCD service.

The next line, stubzone, tells CoreDNS to allow you to set up a DNS server as a “stub resolver” for a particular zone. As a stub resolver, this DNS server directly queries specific name servers for a zone’s information without the need for recursive resolution throughout the entire DNS hierarchy.

The third addition is the path /skydns option, which may look confusing since it doesn’t mention CoreDNS. Even though the value is skydns, it is also the default path for CoreDNS integration as well.

Finally, the last line tells CoreDNS where to store the records. In our example, we have an ETCD service running, using IP address 10.96.149.223 on the default ETCD port of 2379.



You could use the Service's host name instead of the IP here. We used the IP to show the relationships between a pod and a Service, but the name `etcd-dns.etcd-dns.svc` would work as well. Which method you choose will depend on your situation. In our KinD cluster, we don't really need to worry about losing the IP because the cluster is disposable. In the real world, you would want to use the host name to protect against the IP address changing.

Now that you understand how to add an ETCD integrated zone in CoreDNS, the next step is to update the deployment options that ExternalDNS requires to integrate with CoreDNS.

## ExternalDNS configuration options

ExternalDNS can be configured to register ingress or service objects. This is configured in the deployment file of ExternalDNS using the source field. The example below shows the options portion of the deployment that we will use in this chapter.

We also need to configure the provider that ExternalDNS will use, and since we are using CoreDNS, we set the provider to `coredns`.

The last option is the log level we want to set, which we set to `info` to keep our log files smaller and easier to read. The arguments that we will use are shown below:

```
spec:  
  serviceAccountName: external-dns  
  containers:  
    - name: external-dns  
      image: registry.k8s.io/external-dns/external-dns:v0.13.5  
      args:  
        - --source=service  
        - --provider=coredns  
        - --log-level=info
```

Now that we have gone over the ETCD options and deployment, how to configure a new zone to use ETCD, and the options to configure ExternalDNS to use CoreDNS as a provider, we can deploy ExternalDNS in our cluster.

We have included a script called `deploy-externaldns.sh` in the `chapter5/externaldns` folder. Execute the script in the directory to deploy ExternalDNS into your KinD cluster. When you execute the script, it will fully configure and deploy an integrated ExternalDNS with ETCD.



### NOTE

If you see a warning when the script updates the `ConfigMap`, you can safely ignore it. Since our `kubectl` command is using `apply` to update the object, Kubernetes will look for a `last-applied-configuration` annotation, if there is one set. Since you likely do not have that in the existing object, you will see the warning that it's missing. This is just a warning and will not stop the `ConfigMap` update, as you can confirm by looking at the last line of the `kubectl update` command, where it shows the `ConfigMap` was updated: `configmap/coredns configured`

Now, we have added the ability for developers to create dynamically registered DNS names for their services. Next, let's see it in action by creating a new service that will register itself in our CoreDNS server.

## Creating a LoadBalancer service with ExternalDNS integration

Our ExternalDNS will watch all services for an annotation that contains the desired DNS name. This is just a single annotation using the format `annotation external-dns.alpha.kubernetes.io/hostname` with the value of the DNS name you want to register. For our example, we want to register a name of `nginx.foowidgets.k8s`, so we would add an annotation to our NGINX service: `external-dns.alpha.kubernetes.io/hostname: nginx.foowidgets.k8s`.

In the `chapter5/externaldns` directory, we have included a manifest that will deploy an NGINX web server using a LoadBalancer service that contains the annotation to register the DNS name.

Deploy the manifest using `kubectl create -f nginx-lb.yaml`, which will deploy the resources in the default namespace. The deployment is a standard NGINX deployment, but the service has the required annotation to tell the ExternalDNS service that you want to register a new DNS name. The manifest for the service is shown below, with the annotation in bold:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    external-dns.alpha.kubernetes.io/hostname: nginx.foowidgets.k8s
  name: nginx-ext-dns
  namespace: default
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
  selector:
    app: nginx
  type: LoadBalancer
```

When ExternalDNS sees the annotation, it will register the requested name in the zone. The hostname from the annotation will log an entry in the ExternalDNS pod – the registration for our new entry, `nginx.foowidgets.k8s`, is shown below:

```
time="2023-08-04T19:16:00Z" level=debug msg="Getting service (&{ 0 10
0 \"heritage=external-dns,external-dns/owner=default,external-dns/
resource=service/default/nginx-ext-dns\" false 0 1  /skydns/k8s/foowidgets/a-
nginx/39ca730e}) with service host ()"
```

```
time="2023-08-04T19:16:00Z" level=debug msg="Getting service (&{172.18.200.101
0 10 0 \"heritage=external-dns,external-dns/owner=default,external-dns/
resource=service/default/nginx-ext-dns\" false 0 1 /skydns/k8s/foowidgets/
nginx/02b5d1d5}) with service host (172.18.200.101)"
time="2023-08-04T19:16:00Z" level=debug msg="Creating new ep (nginx.foowidgets.
k8s 0 IN A 172.18.200.101 []) with new service host (172.18.200.101)"
```

As you can see in the last line of the log, the record was added as an A-record in the DNS server, pointing to the IP address 172.18.200.101.

The last step for confirming that ExternalDNS is fully working is to test a connection to the application. Since we are using a KinD cluster, we must test this from a pod in the cluster. To provide the new names to external resources, we would need to configure our main DNS server(s) to forward requests for the foowidgets.k8s domain to our CoreDNS server. At the end of this section, we will show the steps to integrate a Windows DNS server, which could be any main DNS server on your network, with our Kubernetes CoreDNS server.

Now we can test the NGINX deployment using the DNS name from our annotation. Since you aren't using the CoreDNS server as your main DNS provider, we need to use a container in the cluster to test name resolution. There is a great utility called **Netshoot** that contains a number of useful troubleshooting tools; it's a great tool to have in your toolbox to test and troubleshoot clusters and pods.

To run a Netshoot container, we can use the `kubectl run` command. We only need the pod to run when we are using it to run tests in the cluster, so we will tell the `kubectl run` command to run an interactive shell and to remove the pod after we exit. To run Netshoot, execute:

```
kubectl run tmp-shell --rm -i --tty --image nicolaka/netshoot -- /bin/bash
```

The pod may take a minute or two to become available, but once it starts up, you will see a `tmp-shell` prompt. At this prompt, we can use `nslookup` to verify that the DNS entry was successfully added. If you attempt to look up `nginx.foowidgets.k8s`, you should receive a reply with the IP address of the service.

```
nslookup nginx.foowidgets.k8s
```

Your reply should look similar to the example below:

```
Server:      10.96.0.10
Address:    10.96.0.10#53

Name:      nginx.foowidgets.k8s
Address:  172.18.200.101
```

This confirms that our annotation was successful and ExternalDNS registered our hostname in the CoreDNS server.

The `nslookup` only proves that there is an entry for `nginx.foowidgets.k8s`; it doesn't test the application. To prove that we have a successful deployment that will work when someone enters the name in a browser, we can use the `curl` utility that is included with Netshoot.

```
tmp-shell:~# curl nginx.foowidgets.k8s
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

Figure 5.2: curl test using the ExternalDNS name

The curl output confirms that we can use the dynamically created service name to access the NGINX web server.

We realize that some of these tests aren't very exciting since you can't test them using a standard browser. To allow CoreDNS to be used outside of the cluster, we need to integrate CoreDNS with your main DNS server, which needs to delegate ownership for the zone(s) that are in CoreDNS. When you delegate a zone, any request to your main DNS server for a host in the delegated zone will forward the request to the DNS server that contains the requested zone.

In the next section, we will integrate the CoreDNS running in our cluster with a Windows DNS server. While we are using Windows as our DNS server, the concepts for delegating a zone are similar between operating systems and DNS servers.

## Integrating CoreDNS with an enterprise DNS server

This section will show you how to use a main DNS server to forward the name resolution of the foowidgets.k8s zone to a CoreDNS server running on a Kubernetes cluster.



The steps provided here are an example of integrating an enterprise DNS server with a Kubernetes DNS service. Because of the external DNS requirement and additional setup, the steps are for reference and **should not be executed** on your KinD cluster.

To find a record in a delegated zone, the main DNS server uses a process known as a recursive query. A recursive query refers to a DNS inquiry initiated by a DNS resolver, acting on behalf of a user. Through the recursive query process, the DNS resolver assumes the task of reaching out to multiple DNS servers in a hierarchical pattern. Its objective is to find the authoritative DNS server for the requested domain and initiate the retrieval of the requested DNS record.

The diagram below illustrates the flow of how DNS resolution is provided by delegating a zone to a CoreDNS server in an enterprise environment.

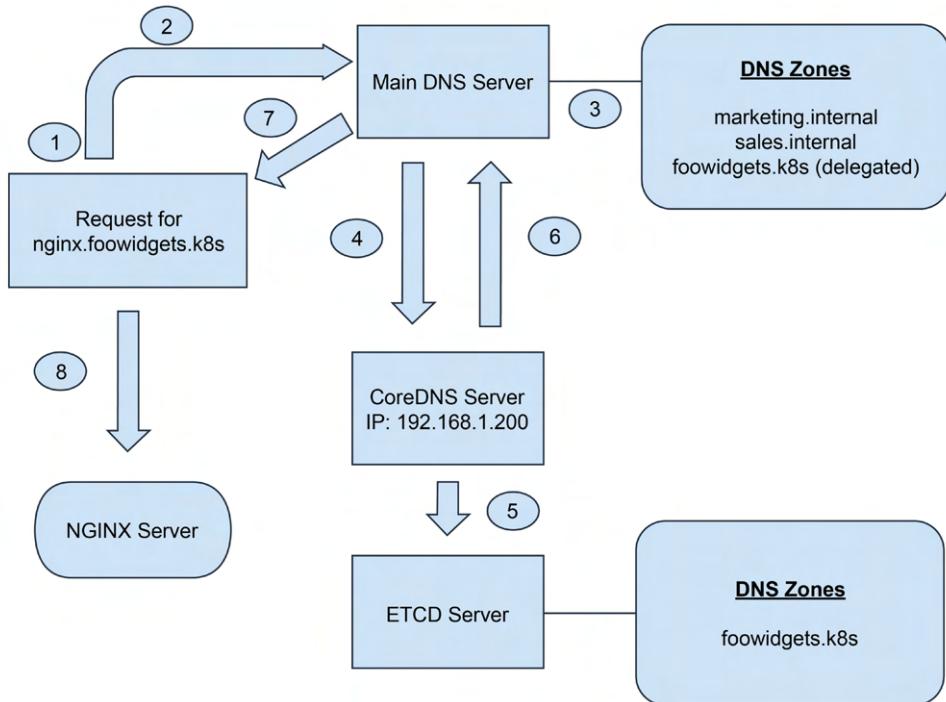


Figure 5.3: DNS delegation flow

1. The local client will look at its DNS cache for the name being requested.
2. If the name is not in the local cache, a request is made to the main DNS server for nginx.foowidgets.k8s.
3. The DNS server receives the query and looks at the zones it knows about. It finds the foowidgets.k8s zone and sees that the zone has been delegated to the CoreDNS running on 192.168.1.200.
4. The main DNS server sends the query to the delegated CoreDNS server.
5. CoreDNS looks for the name, nginx, in the foowidgets.k8s zone.
6. CoreDNS sends the IP address for foowidgets.k8s back to the main DNS server.
7. The main DNS server sends the reply containing the address for nginx.foowidgets.k8s to the client.
8. The client connects to the NGINX server using the IP address returned from CoreDNS.

Let's move on to a real-world example. For our scenario, the main DNS server is running on a Windows 2019 server and we will delegate a zone to a CoreDNS server.

The components deployed are as follows:

- Our network subnet is 10.2.1.0/24
- Windows 2019 or higher server running DNS
- A Kubernetes cluster
- A MetalLB address pool with a range of 10.2.1.70-10.2.1.75
- A CoreDNS instance deployed in a cluster using a LoadBalancer service assigned the IP address 10.2.1.74 from our IP pool
- Deployed add-ons, using the configuration from this chapter including ExternalDNS, an ETCD deployment for CoreDNS, and a new CoreDNS ETCD integrated zone
- Bitnami NGINX deployment to test the delegation

Now, let's go through the configuration steps to integrate our DNS servers.

## Exposing CoreDNS to external requests

We have already covered how to deploy most of the resources that you need to integrate – ETCD, ExternalDNS, and configuring CoreDNS with a new zone that is ETCD-integrated. To provide external access to CoreDNS, we need to create a new service that exposes CoreDNS on TCP and UDP port 53. A complete service manifest is shown below.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    k8s-app: kube-dns
    kubernetes.io/cluster-service: "true"
    kubernetes.io/name: CoreDNS
  name: kube-dns-ext
  namespace: kube-system
spec:
  ports:
  - name: dns
    port: 53
    protocol: UDP
    targetPort: 53
  selector:
    k8s-app: kube-dns
  type: LoadBalancer
  loadBalancerIP: 10.2.1.74
```

There is one new option in the service that we haven't discussed yet – we have added the `spec.loadBalancerIP` to our deployment. This option allows you to assign an IP address to the service so it will have a stable IP address, even if the service is recreated. We need a static IP since we need to enable forwarding from our main DNS server to the CoreDNS server in the Kubernetes cluster.

Once CoreDNS is exposed using a LoadBalancer on port 53, we can configure the main DNS server to forward requests for hosts in the `foowidgets.k8s` domain to our CoreDNS server.

## Configuring the primary DNS server

The first thing to do on our main DNS server is to create a conditional forwarder to the node running the CoreDNS pod.

On the Windows DNS host, we need to create a new conditional forwarder for `foowidgets.k8s` pointing to the IP address that we assigned to the new CoreDNS service. In our example, the CoreDNS service has been assigned to the host `10.2.1.74`:

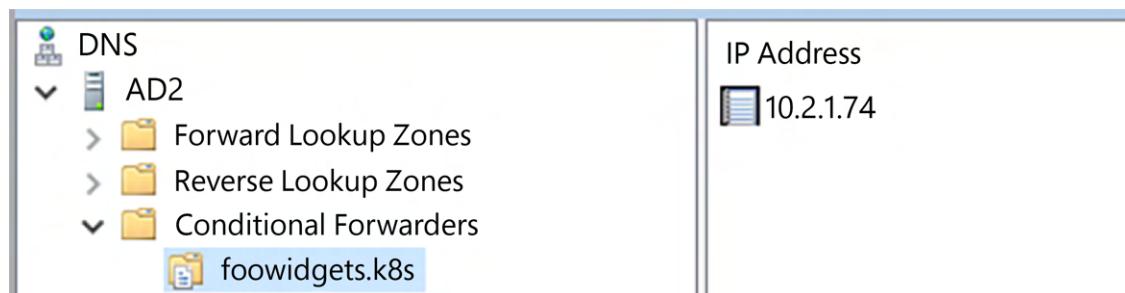


Figure 5.4: Windows conditional forwarder setup

This configures the Windows DNS server to forward any request for a host in the `foowidgets.k8s` domain to the CoreDNS service running on IP address `10.2.1.74`.

## Testing DNS forwarding to CoreDNS

To test the configuration, we will use a workstation on the main network that has been configured to use the Windows DNS server.

The first test we will run is an `nslookup` of the NGINX record that was created by the MetallLB annotation:

From Command Prompt, we execute an `nslookup nginx.foowidgets.k8s` command:

```
Server: AD2.hyper-vplanet.com
Address: 10.2.1.14
Name: nginx.foowidgets.k8s
Address: 10.2.1.75
```

Since the query returned the IP address we expected for the record, we can confirm that the Windows DNS server is forwarding requests to CoreDNS correctly.

We can do one more additional NGINX test from the laptop's browser. In Chrome, we can use the URL registered in CoreDNS, `nginx.foowidgets.k8s`.



Figure 5.5: Success browsing from an external workstation using CoreDNS

One test confirms that the forwarding works, but we want to create an additional deployment to verify the system is fully working.

To test a new service, we deploy a different NGINX server called microbot, with a service that has an annotation assigning the name `microbot.foowidgets.k8s`. MetalLB has assigned the service the IP address of `10.2.1.65`.

Like our previous test, we test the name resolution using `nslookup`:

```
Name: AD2.hyper-vplanet.com  
Address: 10.2.1.65
```

To confirm that the web server is running correctly, we browse to the URL from a workstation:

ⓘ Not secure | `microbot.foowidgets.k8s/`



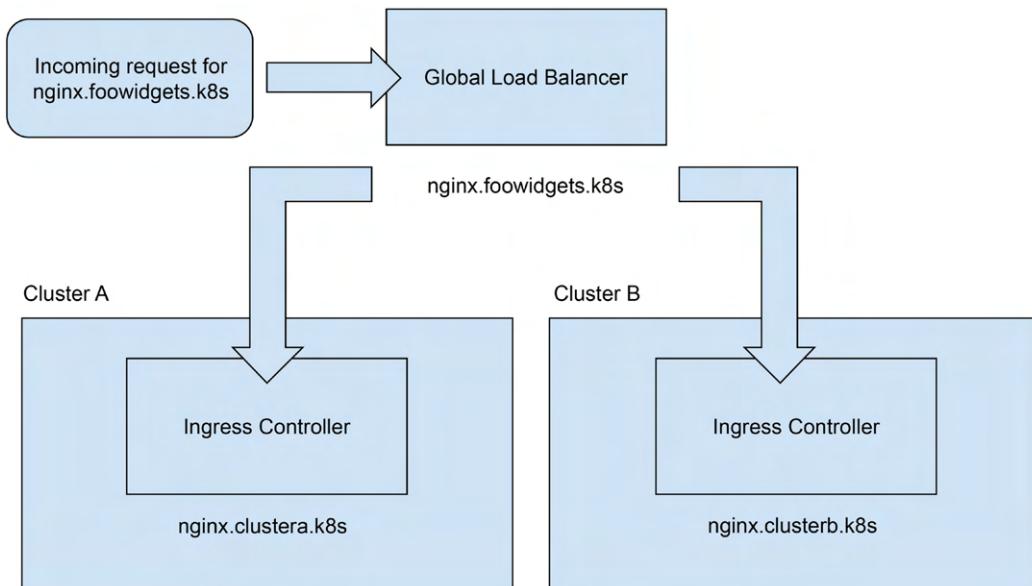
Container hostname: `microbot-5b8559b777-h6tpp`

Figure 5.6: Successful browsing from an external workstation using CoreDNS

Success! We have now integrated an enterprise DNS server with a CoreDNS server running on a Kubernetes cluster. This integration provides users with the ability to register service names dynamically by simply adding an annotation to the service.

## Load balancing between multiple clusters

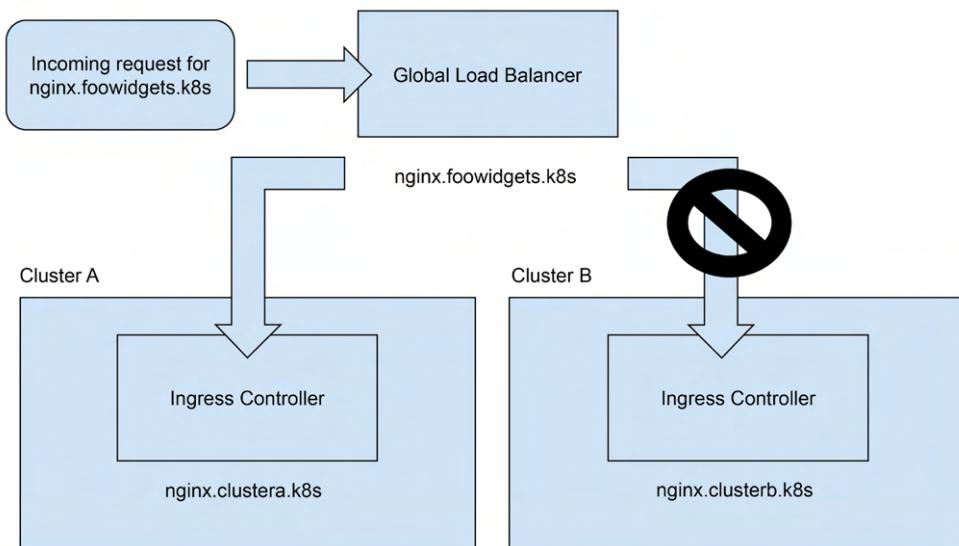
There are various ways to configure running services in multiple clusters, often involving complex and costly add-ons like global load balancers. A global load balancer can be thought of as a traffic cop – it knows how to direct incoming traffic between multiple endpoints. At a high level, you can create a new DNS entry that the global load balancer will control. This new entry will have backend systems added to the endpoint list and based on factors like health, connections, or bandwidth, it will direct the traffic to the endpoint nodes. If an endpoint is unavailable for any reason, the load balancer will remove it from the endpoint list. By removing it from the list, traffic will only be sent to healthy nodes, providing a smooth end user experience. There's nothing worse for a customer than getting a website not found error when they attempt to access a site.



*Figure 5.7: Global load balancing traffic flow*

The figure above shows a healthy workflow where both clusters are running an application that we are load balancing between the clusters. When the request hits the load balancer, it will send the traffic in a round-robin fashion between the two clusters. The `nginx.foowidgets.k8s` request will ultimately send the traffic to either `nginx.cluster.a.k8s` or `nginx.cluster.b.k8s`.

In *Figure 5.8*, we have a failure of our NGINX workload in cluster B. Since the global load balancer has a health check on the running workloads, it will remove the endpoint in cluster B from the `nginx.foowidgets.k8s` entry.



*Figure 5.8: Global load balancing traffic flow with a site failure*

Now, for any traffic that comes into the load balancer requesting `nginx.foowidgets.k8s`, the only endpoint that will be used for traffic is running on cluster A. Once the issue has been resolved on cluster B, the load balancer will automatically add the cluster B endpoint back into the `nginx.foowidgets.k8s` record.

Such solutions are widespread in enterprises, with many organizations utilizing products from companies like **F5**, **Citrix**, **Kemp**, and **A10**, as well as CSP-native solutions like **Route 53** and **Traffic Director**, to manage workloads across multiple clusters. However, there are projects with similar functionality that integrate with Kubernetes, and they come at little to no cost. While these projects may not offer all the features of some vendor solutions, they often meet the needs of most use cases without requiring the full spectrum of expensive features.

One such project is **K8GB**, an innovative open-source project that brings **Global Server Load Balancing (GSLB)** to Kubernetes. With K8GB, organizations can easily distribute incoming network traffic across multiple Kubernetes clusters located in different geographical locations. By intelligently routing requests, K8GB guarantees low latency, optimal response times, and redundancy, providing an exceptional solution to any enterprise.

This section will introduce you to K8GB, but if you want to learn more about the project, browse to the project's main page at <https://www.k8gb.io>.

Since we are using KinD and a single host for our cluster, this section of the book is meant to introduce you to the project and the benefits it provides. This section is for reference only, since it is a complex topic that requires multiple components, some of which are outside of Kubernetes. If you decide you want to implement a solution for yourself, we have included example documentation and scripts in the book's repo under the `chapter5/k8gs-example` directory.

K8GB is a CNCF sandbox project, which means it is in its early stages and any newer version after the writing of this chapter may have changes to objects and configurations.

## Introducing the Kubernetes Global Balancer

Why should you care about a project like K8GB?

Let's consider an internal enterprise cloud as an example, operating a Kubernetes cluster at a production site alongside another cluster at a disaster recovery site. To ensure a smooth user experience, it is important to enable applications to transition seamlessly between these data centers, without requiring any manual intervention during disaster recovery events. The challenge lies in fulfilling the enterprise's demand for high availability of microservices when multiple clusters are simultaneously serving these applications. We need to effectively address the need for continuous and uninterrupted service across geographically dispersed Kubernetes clusters.

This is where **K8GB** comes in.

What makes K8GB an ideal solution for addressing our high availability requirements? As documented on their site, the key features include the following:

- Load balancing is provided using a timeproof DNS protocol that is extremely reliable and works well for global deployments
- There is no requirement for a management cluster
- There is no single point of failure
- It uses native Kubernetes health checks for load balancing decisions
- Configuring is as simple as a single Kubernetes CRD
- It works with any Kubernetes cluster – on-prem or off-prem
- It's free!

As you will see in this section, K8GB provides an easy and intuitive configuration that makes providing global load balancing to your organization easy. This may make K8GB look like it doesn't do very much, but behind the scenes, it provides a number of advanced features, including:

- **Global Load Balancing:** Facilitates the distribution of incoming network traffic among multiple Kubernetes clusters located in different geographic regions. As a result, it enables optimized application delivery, ensuring reduced latency and improved user experience for users.
- **Intelligent Traffic Routing:** Utilizing sophisticated routing algorithms to intelligently steer client requests towards the closest or most appropriate Kubernetes cluster, considering factors like proximity, server health, and application-specific rules. This approach ensures efficient and highly responsive traffic management for optimal application performance.
- **High Availability and Redundancy:** Guarantees high availability and fault tolerance for applications by automatically redirecting traffic in the event of cluster, application, or data center failure. This failover mechanism minimizes downtime during disaster recovery scenarios, ensuring uninterrupted service delivery to users.

- **Automated Failover:** Simplifies operations by enabling automatic failover between data centers without the need for manual intervention. This eliminates the requirement for human-triggered Disaster Recovery (DR) events or tasks, ensuring quick and uninterrupted service delivery and streamlined operations.
- **Integration with Kubernetes:** Offers seamless integration with Kubernetes, simplifying the setup and configuration of GSLB for applications deployed on clusters. Leveraging Kubernetes' native capabilities, K8GB delivers a scalable solution, enhancing the overall management and efficiency of global load balancing operations.
- **On-Prem and Cloud Provider Support:** Provides enterprises a way to efficiently manage GSLB for multiple Kubernetes clusters, enabling seamless handling of complex multi-region deployments and hybrid cloud scenarios. This ensures optimized application delivery across different environments, enhancing the overall performance and resilience of the infrastructure.
- **Customization and Flexibility:** Provides users the freedom to define personalized rules and policies for traffic routing, providing organizations with the flexibility to customize GSLB configurations to meet their unique requirements precisely. This empowers enterprises to optimize traffic management based on their specific needs and ensures seamless adaptation to ever-changing application demands.
- **Monitoring, Metrics, and Tracing:** Includes monitoring, metrics, and tracing capabilities, enabling administrators to access insights into traffic patterns, health, and performance metrics spanning across multiple clusters. This provides enhanced visibility, empowering administrators to make informed decisions and optimize the overall performance and reliability of the GSLB setup.

Now that we have discussed the key features of K8GB, let's get into the details.

## Requirements for K8GB

For a product that provides a complex function like global load balancing, K8GB doesn't require a lot of infrastructure or resources to provide load balancing to your clusters. The latest release, which, as of this chapter's creation, is 0.12.2 – has only a handful of requirements:

- The CoreDNS servers Load Balancer IP address in the main DNS zone using the naming standard `gslb-ns-<k8gb-name>-gb.foowidgets.k8s` – for example, `gslb-ns-us-nyc-gb.foowidgets.k8s` and `gslb-ns-us-buf-gb.foowidgets.k8s`

If you are using K8GB with a service like Route 53, Infoblox, or NS1, the CoreDNS servers will be added to the domain automatically. Since our example is using an on-premises DNS server running on a Windows 2019 server, we need to create the records manually.

- An Ingress controller
- A K8GB controller deployed in the cluster, which will deploy:
  - The K8GB controller
  - A CoreDNS server with the CoreDNS CRD plugin configured – this is included in the deployment on K8GB

Since we have already explored NGINX ingress controllers in previous chapters, we now turn our attention to the additional requirements: deploying and configuring the K8GB controller within a cluster.

In the next section, we will discuss the steps to implement K8GB.

## Deploying K8GB to a cluster

We have included example files in the GitHub repo under the `chapter5/k8gb-example` directory. The scripts are based on the example we will use for the remainder of the chapter. If you decide to use the files in a development cluster, you will need to meet the requirements below:

- Two Kubernetes clusters (a single-node `kubeadm` cluster for each cluster will work)
- CoreDNS deployed in each cluster
- K8GB deployed in each cluster
- An edge DNS server that you can use to delegate the domain for K8GB

Installing K8GB has been made very simple – you only need to deploy a single Helm chart using a `values.yaml` file that has been configured for your infrastructure.

To install K8GB, you will need to add the K8GB repository to your Helm repo list and then update the charts:

```
helm repo add k8gb https://www.k8gb.io  
helm repo update
```

Before we execute a `helm install` command, we need to customize the Helm `values.yaml` file for each cluster deployment. We have included a values file for both of the clusters used in our example, `k8gb-buff-values.yaml` and `k8gb-nyc-values.yaml`, located in the `chapter5/k8gb-example` directory. The options in the values file will be discussed in the *Customizing the Helm chart values* section.

## Understanding K8GB load balancing options

For our example, we will configure K8GB as a failover load balancer between two on-premises clusters; however, K8GB is not limited to just failover. Like most load balancers, K8GB offers a variety of solutions that can be configured differently for each load-balanced URL. It offers the most commonly required strategies, including round robin, weighted round robin, failover, and GeoIP.

Each of the strategies is described below:

- **Round Robin:** If you do not specify a strategy, it will default to a simple round-robin load balancing configuration. Using round robin means that requests will be split between the configured clusters – request 1 will go to cluster 1, request 2 will go to cluster 2, request 3 will go to cluster 1, request 4 will go to cluster 2, and so on.
- **Weighted Round Robin:** Similar to round robin, this strategy provides the ability to specify the percentage of traffic to send to a cluster; for example, 75% of traffic will go to cluster 1 and 15% will go to cluster 2.

- **Failover:** All traffic will go to the primary cluster unless all pods for a deployment become unavailable. If all pods are down in cluster 1, cluster 2 will take over the workload until the pods in cluster 1 become available again, which will then become the primary cluster again.
- **GeoIP:** Directs requests to the closest cluster to the client connection. If the closest host is down, it will use a different cluster similar to how the failover strategy works. To use this strategy, you will need to create a GeoIP database (an example can be found here: <https://github.com/k8gb-io/coredns-crd-plugin/tree/main/terratest/geogen>), and your DNS server needs to support the EDNS0 extension.



**EDNS0** is based on RFC 2671, which outlines how EDNS0 works and its various components, including the format of EDNS0-enabled DNS messages, the structure of EDNS0 options, and guidelines for its implementation. The goal of RFC 2671 is to provide a standardized approach for extending the capabilities of the DNS protocol beyond its original limitations, allowing for the incorporation of new features, options, and enhancements

Now that you know the available strategies, let's go over our example infrastructure for our clusters:

Cluster/Server Details	Details
Corporate DNS Server – New York City IP: 10.2.1.14	Main corporate zone <code>foowidgets.k8s</code> Host records for the CoreDNS servers <code>gslb-ns-us-nyc-gb.foowidgets.k8s</code> <code>gslb-ns-us-buf-gb.foowidgets.k8s</code> Global domain configured delegating to the CoreDNS servers in the clusters <code>gb.foowidgets.k8s</code>
New York City, New York – Cluster 1 Primary Site CoreDNS LB IP: 10.2.1.221 Ingress IP: 10.2.1.98	NGINX Ingress Controller exposed using HostPort CoreDNS deployment exposed using MetalLB
Buffalo, New York – Cluster 2 Secondary Site CoreDNS LB IP: 10.2.1.224 Ingress IP: 10.2.1.167	NGINX Ingress Controller exposed using HostPort CoreDNS deployment exposed using MetalLB

Table 5.1: Cluster details

We will use the details from the above table to explain how we would deploy K8GB in our example infrastructure.

With the details of the infrastructure, we can now create our Helm `values.yaml` files for each deployment. In the next section, we will show the values we need to configure using the example infrastructure, explaining each value.

## Customizing the Helm chart values

Each cluster will have a similar values file; the main changes will be the tag values we use. The values file below is for the New York City cluster:

```
k8gb:  
  dnsZone: "gb.foowidgets.k8s"  
  edgeDNSZone: "foowidgets.k8s"  
  edgeDNSServers:  
    - 10.2.1.14  
  clusterGeoTag: "us-buf"  
  extGslbClustersGeoTags: "us-nyc"  
  
coredns:  
  isClusterService: false  
  deployment:  
    skipConfig: true  
  image:  
    repository: absaoss/k8s_crd  
    tag: v0.0.11  
  serviceAccount:  
    create: true  
    name: coredns  
  serviceType: LoadBalancer
```

The same file contents will be used for the NYC cluster, with the exception of the `clusterGeoTag` and `extGslbClustersGeoTags` values, for the NYC cluster they need to be set to:

```
  clusterGeoTag: "us-nyc"  
  extGslbClustersGeoTags: "us-buf"
```

As you can see, the configuration isn't very lengthy, requiring only a handful of options to configure a usually complex global load balancing configuration.

Now, let's go over some of the main details of the values we are using.

The main details that we will explain are the values in the K8GB section, which configures all of the options K8GB will use for load balancing.

Chart Value	Description
<code>dnsZone</code>	This is the DNS zone that you will use for K8GB – basically, this is the zone that will be used for the DNS records that will be used to store our global load balanced DNS records.
<code>edgeDNSZone</code>	The main DNS zone that contains the DNS records for the CoreDNS servers that are used by the previous option ( <code>dnsZone</code> ).
<code>edgeDNSServers</code>	The edge DNS server – usually the main DNS server used for name resolution.
<code>clusterGeoTag</code>	If you have multiple K8GB controllers, this tag is used to specify instances between each other. In our example, we set these to <code>us-buf</code> and <code>us-nyc</code> for our clusters.
<code>extGslbClusterGeoTags</code>	Specifies the other K8GB controllers to pair with. In our example, each cluster adds the other cluster <code>clusterGeoTags</code> – the Buffalo cluster adds the <code>us-nyc</code> tag and the NYC cluster adds the <code>us-buf</code> tag.
<code>isClusterService</code>	Set to <code>true</code> or <code>false</code> . Used for service upgrades; you can read more at <a href="https://www.k8gb.io/docs/service_upgrade.html">https://www.k8gb.io/docs/service_upgrade.html</a> .
<code>exposeCoreDNS</code>	If set to <code>true</code> , a <code>LoadBalancer</code> service will be created, exposing the CoreDNS deployed in the <code>k8gb</code> namespace on port 53/UDP for external access.
<code>deployment.skipConfig</code>	Set to <code>true</code> or <code>false</code> . Setting it to <code>false</code> tells the deployment to use the CoreDNS shipped with K8GB.
<code>image.repository</code>	Configures the repository to use for the CoreDNS image.
<code>image.tag</code>	Configures the tag to use when pulling the image.
<code>serviceAccount.create</code>	Set to <code>true</code> or <code>false</code> . When set to <code>true</code> , a service account will be created.
<code>serviceAccount.name</code>	Sets the name of the service account from the previous option.
<code>serviceType</code>	Configures the service type for CoreDNS.

Table 5.2: K8GB options

## Using Helm to install K8GB

With the overview of K8GB and the Helm values file complete, we can move on to installing K8GB in the clusters. We have included scripts to deploy K8GB to the Buffalo and NYC clusters. In chapter5/k8gb-example/k8gb, you will see two scripts, `deploy-k8gb-buf.sh` and `deploy-k8gb-nyc.sh` – these should be run in their corresponding clusters.

The script will execute the following steps:

1. Add the K8GB Helm repo to the server's repo list
2. Update the repos
3. Deploy K8GB using the appropriate Helm values file
4. Create a **Gslb record** (covered in the next section)
5. Create a deployment to use for testing in a namespace called `demo`

Once deployed, you will see two pods running in the `k8gb` namespace, one for the `k8gb` controller and the other for the CoreDNS server that will be used to resolve load balancing names:

NAME	READY	STATUS	RESTARTS	AGE
<code>k8gb-8d8b4cb7c-mhg1b</code>	1/1	Running	0	7h58m
<code>k8gb-coredns-7995d54db5-ngdb2</code>	1/1	Running	0	7h37m

We can also verify that the services were created to handle the incoming DNS requests. Since we exposed it using a `LoadBalancer` type, we will see the `LoadBalancer` service on port 53 using the UDP protocol:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)
<code>k8gb-coredns</code>	<code>LoadBalancer</code>	<code>10.96.185.56</code>	<code>10.2.1.224</code>	<code>53:32696/UDP</code>

With the deployment of K8GB complete and verified in both clusters, let's move on to the next section where we will explain how to create our edge delegation for our load balanced zone.

## Delegating our load balancing zone

For our example, we are using a Windows server as our edge DNS server, which is where our K8s DNS names will be registered. On the DNS side, we need to add two DNS records for our CoreDNS servers, and then we need to delegate the load balancing zone to these servers.

The two A records need to be in the main edge DNS zone. In our example, that is the foowidgets.k8s zone. In this zone, we need to add two entries for the CoreDNS servers that are exposed using a LoadBalancer service:

> cluster1.com			
& foowidgets.k8s	gslb-ns-us-buf-gb	Host (A)	10.2.1.224
	gslb-ns-us-nyc-gb	Host (A)	10.2.1.221

Figure 5.9: Adding our CoreDNS servers to the edge zone

Next, we need to create a new delegated zone that will be used for our load balanced service names. In Windows, this is done by *right-clicking* the zone and selecting **New Delegation**; in the delegation wizard, you will be asked for the **Delegated domain**. In our example, we are going to delegate the **gb** domain as our domain.

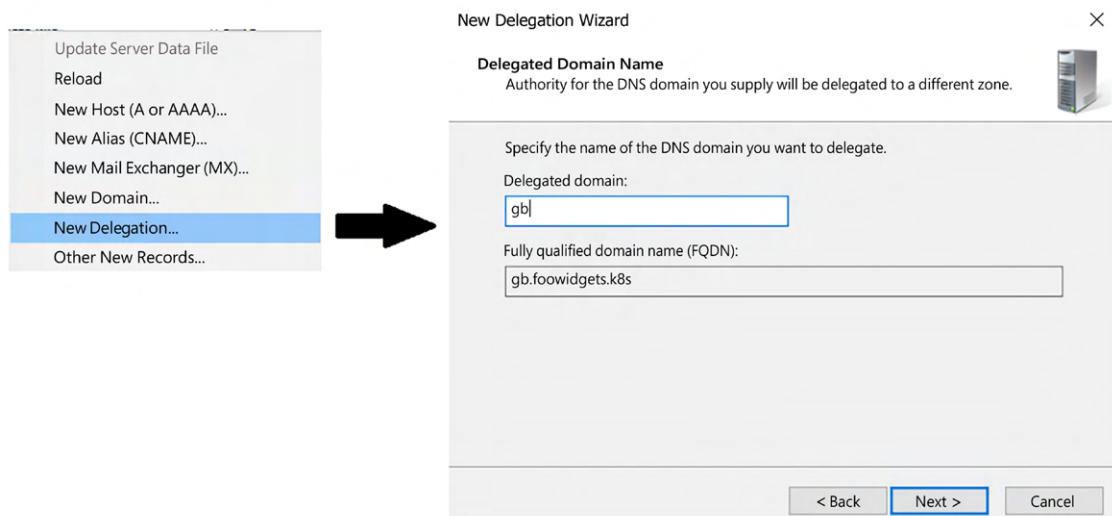


Figure 5.10: Creating a new delegated zone

After you enter the zone name and click **Next**, you will see a new screen to add DNS servers for the delegated domain; when you click **Add**, you will enter the DNS names for your CoreDNS servers. Remember that we created two A records in the main domain, `foowidgets.com`. As you add entries, Windows will verify that the entered name resolves correctly and that DNS queries work. Once you add both CoreDNS servers, the summary screen will show both with their IP addresses.

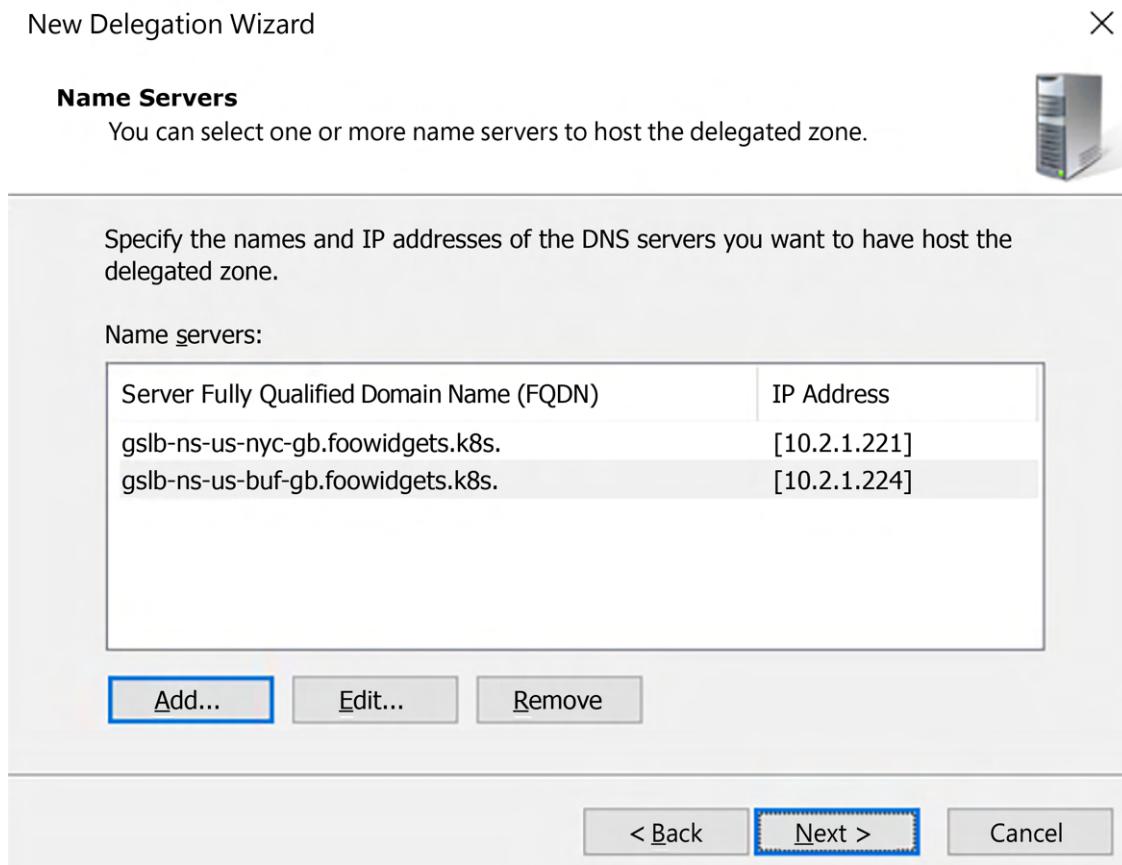


Figure 5.11: Adding the DNS names for the CoreDNS servers

That completes the edge server configuration. For certain edge servers, K8GB will create the delegation records, but there are only a handful of servers that are compatible with that feature. For any edge server that doesn't auto create the delegated servers, you need to create them manually like we did in this section.

Now that we have CoreDNS in our clusters and we have delegated the load balanced zone, we will deploy an application that has global load balancing to test our configuration.

## Deploying a highly available application using K8GB

There are two methods to enable global load balancing for an application. You can create a new record using a custom resource provided by K8GB, or you can annotate an Ingress rule. For our demonstration of K8GB, we will deploy a simple NGINX web server in a cluster and add it to K8GB using the natively supplied custom resource.

## Adding an application to K8GB using custom resources

When we deployed K8GB, a new Custom Resource Definition (CRD) named `Gslb` was added to the cluster. This CRD assumes the role of managing applications marked for global load balancing. Within the `Gslb` object, we define a specification for the Ingress name, mirroring the format of a regular Ingress object. The sole distinction between a standard Ingress and a `Gslb` object lies in the last portion of the manifest, the strategy.

The strategy defines the type of load balancing we want to use, which is failover for our example, and the primary GeoTag to use for the object. In our example, the NYC cluster is our primary cluster, so our `Gslb` object will be set to `us-buf`.

To deploy an application that will leverage load balancing, we need to create the following in both clusters:

1. A standard deployment and service for the application. We will call the deployment `nginx`, using the standard NGINX image.
2. A `Gslb` object in each cluster. For our example, we will use the manifest below, which will declare the Ingress rule and set the strategy to failover using `us-buf` as the primary K8GB. Since the `Gslb` object has the information for the Ingress rule, you do not need to create an Ingress rule; `Gslb` will create the Ingress object for us. Let's look at an example below:

```
apiVersion: k8gb.absa.oss/v1beta1
kind: Gslb
metadata:
  name: gslb-failover-buf
  namespace: demo
spec:
  ingress:
    ingressClassName: nginx
    rules:
      - host: fe.gb.foowidgets.k8s      # Desired GSLB enabled FQDN
        http:
          paths:
            - backend:
                service:
                  name: nginx           # Service name to enable GSLB for
                  port:
                    number: 80
                path: /
                pathType: Prefix
  strategy:
    type: failover                   # Global Load balancing strategy
    primaryGeoTag: us-buf            # Primary cluster geo tag
```

When you deploy the manifest for the `Gslb` object, it will create two Kubernetes objects, the `Gslb` object and an Ingress object.

If we looked at the `demo` namespace for the `Gslb` objects in the `Buffalo` cluster, we would see the following:

NAMESPACE	NAME	STRATEGY	GEOTAG
demo	<code>gslb-failover-buf</code>	failover	<code>us-buf</code>

And if we looked at the Ingress objects in the `NYC` cluster, we would see:

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
<code>gslb-failover-buf</code>	nginx	<code>fe.gb.foowidgets.k8s</code>	10.2.1.167	80	15h

We would also have similar objects in the `NYC` cluster, which we will explain in the *Understanding how K8GB provides global load balancing* section.

## Adding an application to K8GB using Ingress annotations

The second method for adding an application to K8GB is to add two annotations to a standard Ingress rule, which was primarily added to allow developers to add an existing Ingress rule to K8GB.

To add an Ingress object to the global load balancing list, you only need to add two annotations to the Ingress object, `strategy` and `primary-geotag`. An example of the annotations is shown below:

```
k8gb.io/strategy: "failover"
k8gb.io/primary-geotag: "us-buf"
```

This would add the Ingress to K8GB using the failover strategy using the `us-buf` GeoTag as the primary tag.

Now that we have deployed all of the required infrastructure components and all of the required objects to enable global load balancing for an application, let's see it in action.

## Understanding how K8GB provides global load balancing

The design of K8GB is complex, but once you deploy an application and understand how K8GB maintains zone files, it will become easier. This is a fairly complex topic, and it does assume some previous knowledge of how DNS works, but by the end of this section, you should be able to explain how K8GB works.

## Keeping the K8GB CoreDNS servers in sync

The first topic to discuss is how K8GB manages to keep two, or more, zone files in sync to provide seamless failover for our deployments. Seamless failover is a process that ensures an application continues to run smoothly even during system issues or failures. It automatically transitions to the backup system or resource, maintaining an uninterrupted user experience.

As we mentioned earlier, each K8GB CoreDNS server in the clusters must have an entry in the main DNS server.

This is the DNS server and zone that we configured for the edge values in the `values.yaml` file:

```
edgeDNSZone: "foowidgets.k8s"
edgeDNSServer: "10.2.1.14"
```

So, in the edge DNS server (10.2.1.14), we have a host record for each CoreDNS server using the required K8GB naming convention:

<code>gslb-ns-us-nyc-gb.gb.foowidgets.k8s</code>	<code>10.2.1.221</code>	(The NYC CoreDNS load balancer IP)
<code>gslb-ns-us-buf-gb.gb.foowidgets.k8s</code>	<code>10.2.1.224</code>	(The BUF CoreDNS load balancer IP)

K8GB will communicate between all of the CoreDNS servers and update any records that need to be updated due to being added, deleted, or updated.

This becomes a little easier to understand with an example. Using our cluster example, we have deployed an NGINX web server and created all of the required objects in both clusters. After deploying, we would have a `Gslb` and `Ingress` object in each cluster, as shown below:

Cluster: NYC	Cluster: Buffalo (Primary)
Deployment: nginx	Deployment: nginx
<code>Gslb: gslb-failover-nyc</code>	<code>Gslb: gslb-failover-buf</code>
<code>Ingress: fe.gb.foowidgets.k8s</code>	<code>Ingress: fe.gb.foowidgets.k8s</code>
NGINX Ingress IP: <code>10.2.1.98</code>	NGINX Ingress IP: <code>10.2.1.167</code>

Table 5.3: Objects in each cluster

Since the deployment is healthy in both clusters, the CoreDNS servers will have a record for `fe.gb.foowidgets.k8s` with an IP address of `10.2.1.167`, the primary deployment. We can verify this by running a `dig` command on any client machine that uses the edge DNS server (10.2.1.14):

```
; <>> DiG 9.16.23-RH <>> fe.gb.foowidgets.k8s
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6654
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;fe.gb.foowidgets.k8s.           IN      A
;; ANSWER SECTION:
fe.gb.foowidgets.k8s.    30      IN      A      10.2.1.167
;; Query time: 3 msec
;; SERVER: 10.2.1.14#53(10.2.1.14)
```

```
; ; WHEN: Mon Aug 14 08:47:12 EDT 2023
; ; MSG SIZE  rcvd: 65
```

As you can see in the output from `dig`, the host resolved to `10.2.1.167` since the application is healthy in the primary cluster. If we `curl` the DNS name, we will see that the NGINX server in Buffalo replies:

```
# curl fe.gb.foowidgets.k8s
<html>
<h1>Welcome</h1>
<br>
<h1>Hi! This is a webserver in Buffalo for our K8GB example... </h1>
```

We will simulate a failure by scaling the replicas for the deployment in the Buffalo cluster to `0`, which will look like a failed application to K8GB. When the K8GB controller in the NYC cluster sees that the application no longer has any healthy endpoints, it will update the CoreDNS record in all servers with the secondary IP address to fail the service over to the secondary cluster.

Once scaled down, we can use `dig` to verify what host is returned:

```
; <>> DiG 9.16.23-RH <>> fe.gb.foowidgets.k8s
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46104
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;fe.gb.foowidgets.k8s.           IN      A
;; ANSWER SECTION:
fe.gb.foowidgets.k8s.    27      IN      A      10.2.1.98
;; Query time: 1 msec
;; SERVER: 10.2.1.14#53(10.2.1.14)
;; WHEN: Mon Aug 14 08:49:27 EDT 2023
;; MSG SIZE  rcvd: 65
```

We will `curl` again to verify that the workload has been moved to the NYC cluster. When we execute `curl`, we will see that the NGINX server is now located in the NYC cluster:

```
# curl fe.gb.foowidgets.k8s
<html>
<h1>Welcome</h1>
<br>
<h1>Hi! This is a webserver in NYC for our K8GB example... </h1>
</html>
```

Note that the IP address returned is now the IP address for the deployment in the **Buffalo** cluster, the secondary cluster, **10.2.1.98**. This proves that K8GB is working correctly and providing us with a Kubernetes-controlled global load balancer.

Once the application becomes healthy in the primary cluster, K8GB will update CoreDNS and any requests will resolve to the main cluster again. To test this, we scaled the deployment in **Buffalo** back up to **1** and ran another dig test:

```
; <>> DiG 9.16.23-RH <>> fe.gb.foowidgets.k8s
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 6654
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 4000
;; QUESTION SECTION:
;fe.gb.foowidgets.k8s.           IN      A
;; ANSWER SECTION:
fe.gb.foowidgets.k8s.    30      IN      A      10.2.1.167
;; Query time: 3 msec
;; SERVER: 10.2.1.14#53(10.2.1.14)
;; WHEN: Mon Aug 14 08:47:12 EDT 2023
;; MSG SIZE  rcvd: 65
```

We can see that the IP has been updated to reflect the NYC Ingress controller on address **10.2.1.167**, the primary location.

Finally, a last curl to verify that the workload is being serviced out of the **Buffalo** cluster:

```
# curl fe.gb.foowidgets.k8s
<html>
<h1>Welcome</h1>
<br>
<h1>Hi! This is a webserver in Buffalo for our K8GB example... </h1>
</html>
```

K8GB is a unique, and impressive, project from the CNCF that offers global load balancing similar to what other, more expensive, products offer today.

It's a project that we are watching carefully, and if you need to deploy applications across multiple clusters, you should consider looking into the K8GB project as it matures.

## Summary

In this chapter, you learned how to provide automatic DNS registration to any service that uses a LoadBalancer service. You also learned how to deploy a highly available service using the CNCF project, K8GB, which provides global load balancing to a Kubernetes cluster.

These projects have become integral to numerous enterprises, offering users capabilities that previously required the efforts of multiple teams and, often, extensive paperwork, to deliver applications to customers. Now, your teams can swiftly deploy and update applications using standard agile practices, providing your organization with a competitive advantage.

In the next chapter, *Integrating Authentication into Your Cluster*, we will explore the best methods and practices for implementing secure authentication in Kubernetes. You will learn how to integrate enterprise authentication using the OpenID Connect protocol and how to use Kubernetes impersonation. We will also discuss the challenges of managing credentials in a cluster and offer practical solutions for authenticating users and pipelines.

## Questions

1. Kubernetes does not support using both TCP and UDP with services.

- a. True
- b. False

Answer: b

2. ExternalDNS only integrates with CoreDNS.

- a. True
- b. False

Answer: b

3. What do you need to configure on your edge DNS server for K8GB to provide load balancing to a domain?

- a. Nothing, it works without additional configuration
- b. It must point to a cloud-provided DNS server
- c. You must delegate a zone that points to your cluster IP
- d. Create a delegation to your CoreDNS instances

Answer: d

4. What strategy is not supported by K8GB?

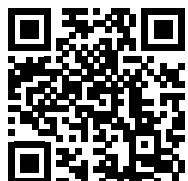
- a. Failover
- b. Round robin
- c. Random distribution
- d. GeoIP

Answer: c

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>





# 6

## Integrating Authentication into Your Cluster

Once a cluster has been built, users will need to interact with it securely. For most enterprises, this means authenticating individual users and pipelines, making sure they can only access what they need in order to do their jobs. This is known as least privileged access. The principle of least privilege is a security practice that centers on providing users, systems, applications, or processes with only the essential access and permissions required to execute their tasks. With Kubernetes, this can be challenging because a cluster is a collection of APIs, not an application with a frontend that can prompt for authentication, nor does it provide a secure way to manage credentials on its own.

Failing to create an authentication strategy can lead to your cluster being taken over. Once a cluster is potentially compromised, it's almost impossible to determine if an attacker has been purged, and you'll need to start over. A breached cluster can also lead to breaches in other systems too, such as a database your applications may be accessing. Authentication is the first step to make sure this doesn't happen.

In this chapter, you'll learn how to integrate enterprise authentication into your cluster using the **OpenID Connect** protocol and Kubernetes impersonation. We'll also cover several anti-patterns and explain why you should avoid using them. To close out the chapter, you'll also learn how to integrate your pipelines into your clusters securely.

In this chapter, we will cover the following topics:

- Understanding how Kubernetes knows who you are
- Understanding OpenID Connect
- Configuring KinD for OpenID Connect
- Introducing impersonation to integrate authentication with cloud-managed clusters
- Configuring your cluster for impersonation
- Configuring impersonation without OpenUnison
- Authenticating from pipelines

Let's get started!

## Technical requirements

To complete the exercises in this chapter, you will require the following:

- An Ubuntu 22.04 server with 8 GB of RAM
- A fresh KinD cluster running with the configuration from *Chapter 2, Deploying Kubernetes Using Kind*

You can access the code for this chapter at the following GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter6>.

## Getting Help

We do our best to test everything, but there are sometimes half a dozen systems or more in our integration labs. Given the fluid nature of technology, sometimes things that work in our environment don't work in yours. Don't worry – we're here to help! Open an issue on our GitHub repo at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/issues>, and we'll be happy to help you out!

## Understanding how Kubernetes knows who you are

In the 1999 sci-fi film *The Matrix*, Neo talks to a child about the Matrix as he waits to see the Oracle. The child explains to him that the trick to manipulating the Matrix is to realize that “*There is no spoon.*”

This is a great way to look at users in Kubernetes because they don't exist. With the exception of service accounts, which we'll talk about later, there are no objects in Kubernetes called “User” or “Group.” Every API interaction must include enough information to tell the API server who the user is and what groups the user is a member of. This assertion can take different forms, depending on how you plan to integrate authentication into your cluster.

In this section, we will get into the details of the different ways Kubernetes can associate a user with a cluster.

## External users

Users who access the Kubernetes API from outside the cluster will usually do so using one of two authentication methods:

- **Certificate:** You can assert who you are by using a client certificate that has information about you, such as your username and groups. The certificate is used as part of the TLS negotiation process.
- **Bearer token:** Embedded in each request, a bearer token can either be a self-contained token that contains all the information needed to verify itself, or a token that can be exchanged by a webhook in the API server for that information.

There is a third method that can be used for authentication, using a **service account**. However, using service accounts to access the API server outside the cluster is strongly discouraged. We'll cover the risks and concerns around using service accounts in the *Other authentication options* section.

## Groups in Kubernetes

Different users can be assigned the same permissions without creating RoleBinding objects for each user individually via groups. Kubernetes includes two types of groups:

- **System assigned:** These groups start with the `system:` prefix and are assigned by the API server. An example group is `system:authenticated`, which is assigned to all authenticated users. Another example of system-assigned groups is the `system:serviceaccounts:namespace` group, where `Namespace` is the name of the namespace that contains all the service accounts for the namespace named in the group.
- **User-asserted groups:** These groups are asserted by the authentication system, either in the token provided to the API server or via the authentication webhook. There are no standards or requirements for how these groups are named. Just as with users, groups don't exist as objects in the API server. Groups are asserted at authentication time by external users and tracked locally for system-generated groups. When asserting a user's groups, the primary difference between a user's unique ID and groups is that the unique ID is expected to be unique, whereas groups are not.

While you may be authorized for access by groups, all access is still tracked and audited based on your user's unique ID.

## Service accounts

Service accounts are objects that exist in the API server to track which pods can access the various APIs. Service account tokens are called **JSON Web Tokens**, or **JWTs**, and depending on how the token was generated, there are two ways to obtain a service account:

- The first is from a secret that can be generated by Kubernetes when a ServiceAccount is created.
- The second is via the `TokenRequest` API, which is used to inject a secret into pods via a mount point or externally from the cluster. All service accounts are used by injecting the token as a header in the request into the API server. The API server recognizes it as a service account and validates it internally.

We will cover how to create these tokens in a specific context later in the chapter.

Unlike users, service accounts **CANNOT** be assigned to arbitrary groups. Service accounts are members of pre-built groups only; you can't create a group of specific service accounts to assign roles.

Now that we have explored the fundamentals of how Kubernetes identifies users, we'll explore how this framework fits into the **OpenID Connect (OIDC)** protocol. OIDC provides the security most enterprises require and is standards-based, but Kubernetes doesn't use it in a way that is typical of many web applications. Understanding these differences and why Kubernetes needs them is an important step in integrating a cluster into an enterprise security environment.

## Understanding OpenID Connect

OpenID Connect is a standard identity federation protocol. It's built on the OAuth2 specification and has some very powerful features that make it the preferred choice to interact with Kubernetes clusters.

The main benefits of OpenID Connect are as follows:

- **Short-lived tokens:** If a token is leaked, such as via a log message or breach, you want the token to expire as quickly as possible. With OIDC, you're able to specify tokens that can live for 1–2 minutes, which means the token will likely have expired by the time an attacker attempts to use it.
- **User and group memberships:** When we start discussing authorization in *Chapter 7, RBAC Policies and Auditing*, we'll see immediately that it's important to manage access by group instead of managing access by referencing users directly. OIDC tokens can embed both the user's identifier and their groups, leading to easier access management.
- **Refresh tokens scoped to timeout policies:** With short-lived tokens, you need to be able to refresh them as needed. The time that a refresh token remains valid can be scoped to your enterprise's web application idle timeout policy, keeping your cluster in compliance with other web-based applications.
- **No plugins required for kubectl:** The kubectl binary supports OpenID Connect natively, so there's no need for any additional plugins. This is especially useful if you need to access your cluster from a jump box or VM because you're unable to install the **Command-Line Interface (CLI)** tools directly onto your workstation. There are convenient plugins though, which we will discuss later in the chapter.
- **More multi-factor authentication options:** Many of the strongest multi-factor authentication options require a web browser. Examples include FIDO and WebAuthn, which use hardware tokens.

OIDC is a peer-reviewed standard that has been in use for several years and is quickly becoming the preferred standard for identity federation. Using an existing standard, over something custom-developed, means that Kubernetes leverages the existing expertise of those peer reviews, instead of creating a new authentication protocol where that experience hasn't been tested.

Identity federation is the term used to describe the assertion of identity data and authentication without sharing a user's confidential secret or password. A classic example of identity federation is logging into your employee website and being able to access your benefits provider without having to log in again. Your employee website doesn't share your password with your benefits provider. Instead, your employee website asserts that you logged in at a certain date and time and provides some information about you. This way, your account is federated across two silos (your employee website and benefits portal), without your benefits portal knowing your employee website password.

As you can see, there are multiple components to OIDC. To fully understand how OIDC works, let's begin by understanding the OpenID Connect protocol.

## The OpenID Connect protocol

The two aspects of the OIDC protocol we will focus on are:

- Using tokens with kubectl and the API server
- Refreshing tokens to keep your tokens up to date

We won't focus too much on obtaining tokens. While the protocol to get a token does follow a standard, the login process does not. How you obtain tokens from an identity provider will vary, and it's based on how you choose to implement your **OIDC Identity Provider (IdP)**.

Three tokens are generated from the OIDC login process:

- **access\_token**: This token is used to make authenticated requests to web services your identity provider may provide, such as obtaining user information. It is NOT used by Kubernetes and can be discarded. This token does not have a standard form. It may be a JWT, or it may not.
- **id\_token**: This token is a JWT that encapsulates your identity, including your unique identifier, groups, and expiration information about you that the API server can use to authorize your access. The JWT is signed by your identity provider's certificate and can be verified by Kubernetes, simply by checking the JWT's signature. This is the token you pass to Kubernetes for each request to authenticate yourself.
- **refresh\_token**: kubectl knows how to refresh your **id\_token** for you automatically once it expires. To do this, it makes a call to your IdP's token endpoint using a **refresh\_token** to obtain a new **id\_token**. A **refresh\_token** should only be used once and is opaque, meaning that you, as the holder of the token, have no visibility into its format, and it really doesn't matter to you. It either works, or it doesn't. The **refresh\_token** never goes to Kubernetes (or any other application). It is only used in communications with the IdP.

The **refresh\_token**'s ability to be used multiple times can be allowed in specific circumstances. There are well-known issues with the Kubernetes Go SDK when multiple processes attempt to refresh a token from the same kubectl configuration file at nearly the same time, causing the user's session to be lost and forcing the user to log in again to obtain a new set of tokens. Many identity providers handle this process differently. Some allow **refresh\_tokens** to be reused for varying amounts of time. When reviewing your choice for an identity provider, it's important to review this part of the functionality because it's often left more "open" by default to give a better user experience. Allowing the long-lived reuse of a **refresh\_token** invalidates much of the security provided by a **refresh\_token** and should be used very carefully.

Once you have your tokens, you can use them to authenticate with the API server. The easiest way to use your tokens is to add them to the kubectl configuration, using command-line parameters:

```
kubectl config set-credentials username --auth-provider=oidc --auth-provider-arg=idp-issuer-url=https://host/uri --auth-provider-arg=client-id=kubernetes --auth-provider-arg=refresh-token=$REFRESH_TOKEN --auth-provider-arg=id-token=$ID_TOKEN
```

`config set-credentials` has a few options that need to be provided. We have already explained `id-token` and `refresh-token`, but there are two additional options:

- `idp-issuer-url`: This is the same URL we will use to configure the API server and points to the base URL used for the IdP's discovery URL.
- `client-id`: This is used by your IdP to identify your configuration. This is unique to a Kubernetes deployment and is not considered secret information.

The OpenID Connect protocol has an optional element, known as a `client_secret`, that is shared between an OIDC client and the IdP. It is used to “authenticate” the client before making any requests, such as refreshing a token. While it's supported by Kubernetes as an option, it's recommended to not use it and instead configure your IdP to use a public endpoint (which doesn't use a secret at all).

The client secret has no practical value, since you'd need to share it with every potential user, and since it's a password, your enterprise's compliance framework will likely require that it is rotated regularly, causing support headaches. Overall, it's just not worth any potential downsides in terms of security.

Instead of using a client secret, you should make sure your endpoint leverages the **Proof Key for Code Exchange (PKCE)** protocol. This protocol was originally created to add a layer of randomness to OIDC requests that don't have client secrets. While this is not something that would be leveraged by the `kubectl` command during the refresh process, you're likely to integrate multiple applications from your cluster into your identity provider (such as dashboards) that may have CLI components and won't be able to use a client secret either. **ArgoCD**, which we will integrate in the last two chapters, is a great example. Its CLI utility works with SSO, but unlike `kubectl`, it will initiate SSO for you. When it does, it includes PKCE because you won't have a `client_secret` on each user's workstation.

Kubernetes requires that your identity provider supports the discovery URL endpoint, which is a URL that provides some JSON to tell you where you can get keys to verify JWTs and the various endpoints available. To access the discovery endpoint, take any issuer URL and add `/well-known/openid-configuration`, which will provide the OIDC endpoint information.

Having worked through how the OpenID Connect protocol and tokens work with Kubernetes, let's next walk through how the various components in Kubernetes and `kubectl` interact with each other.

## Following OIDC and the API's interaction

Once `kubectl` has been configured, all of your API interactions will follow the following sequence:

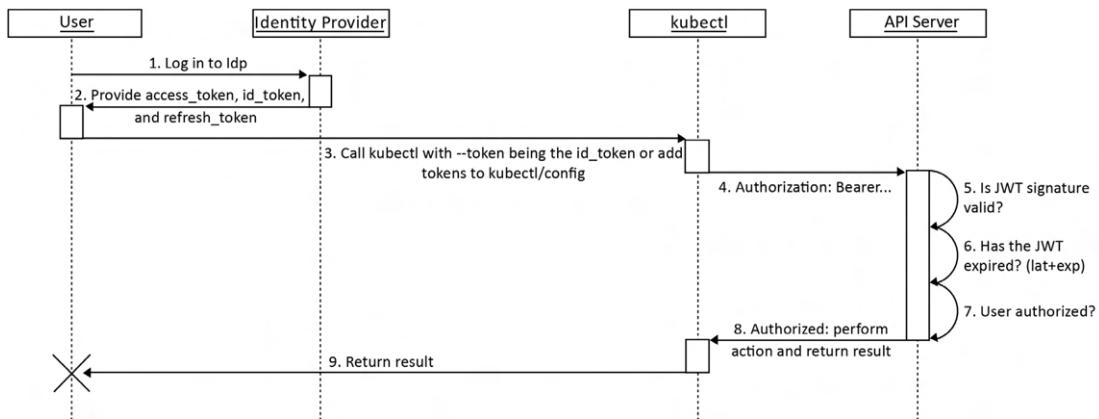


Figure 6.1: Kubernetes/kubectl OpenID Connect sequence diagram

The preceding diagram is from Kubernetes' authentication page at <https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens>. Authenticating a request involves the following:

- Log into your IdP:** This will be different for each IdP. This could involve providing a username and password to a form in a web browser, a multi-factor token, or a certificate. This will be specific to every implementation.
- Provide tokens to the user:** Once authenticated, the user needs a way to generate the tokens needed by kubectl to access the Kubernetes APIs. This can take the form of an application that makes it easy for the user to copy and paste them into the configuration file, or it can be a new file to download.
- This step is where `id_token` and `refresh_token` are added to the `kubectl` configuration. If the tokens were presented to the user in the browser, they can be manually added to the configuration. Alternatively, some solutions provide a new `kubectl` configuration to download at this step. There are also `kubectl` plugins that will launch a web browser to start the authentication process and, once completed, generate your configuration for you.
- Inject id\_token:** Once the `kubectl` command has been called, each API call includes an additional header, called the **Authorization** header, that includes `id_token`.
- JWT signature validation:** Once the API server receives `id_token` from the API call, it validates the signature against the public key provided by the identity provider. The API server will also validate whether the issuer matches the issuer for the API server configuration, and also that the recipient matches the client ID from the API server configuration.
- Check the JWT's expiration:** Tokens are only good for a limited amount of time. The API server ensures that the token hasn't expired. If it has expired, the API server will return with a 401 error code.
- Authorization check:** Now that the user has been authenticated, the API server will determine whether the user identified by the provided `id_token` is able to perform the requested action by matching the user's identifier and asserted groups to internal policies.

8. **Execute the API:** All checks are complete, and the API server executes the request, generating a response that will be sent back to kubectl.
9. **Format the response for the user:** Once the API call (or a series of API calls) is complete, the response in JSON is formatted and presented to the user by kubectl.

In general terms, authentication is the process of validating that you are you. Most of us encounter this when we put our username and password into a website; we're proving who we are. In the enterprise world, authorization then becomes the decision of whether we're allowed to do something. First, we authenticate, and then we authorize.

The standards built around API security don't assume authentication and go straight to authorization, based on some sort of token. It's not assumed that the caller has to be identified. For instance, when you use a physical key to open a door, the door doesn't know who you are, only that you have the right key. This terminology can become very confusing, so don't feel bad if you get a bit lost. You're in good company!

The `id_token` is self-contained; everything the API server needs to know about you is in that token. The API server verifies the `id_token` using the certificate provided by the identity provider and verifies that the token hasn't expired. As long as that all lines up, the API server will move on to authorizing your request based on its own RBAC configuration. We'll cover the details of that process later. Finally, assuming you're authorized, the API server provides a response.

Note that Kubernetes never sees your password or any other secret information that you, and only you, know. The only thing that's shared is the `id_token`, and that's ephemeral. This leads to several important points:

- Since Kubernetes never sees your password or other credentials, it can't compromise them. This can save you a tremendous amount of time working with your security team, as all the tasks and controls related to securing passwords can be skipped!
- The `id_token` is self-contained, which means that if it's compromised, there is nothing you can do, short of rekeying your identity provider, to stop it from being abused. This is why it's so important for your `id_token` to have a short lifespan. At 1–2 minutes, the likelihood that an attacker will be able to obtain an `id_token`, realize what it is, and abuse it is very low.

If, while performing its calls, kubectl finds that `id_token` has expired, it will attempt to refresh it by calling the IdP's token endpoint using `refresh_token`. If the user's session is still valid, the IdP will generate a new `id_token` and `refresh_token`, which kubectl will store for you in the kubectl configuration. This happens automatically with no user intervention. Additionally, a `refresh_token` has a one-time use, so if someone tries to use a previously used `refresh_token`, your IdP will fail the refresh process.

A sudden security event is bound to happen. Someone may need to be locked out immediately; it may be that they're being walked out or that their session has been compromised. Revocation of tokens is dependent on your IdP, so when choosing an IdP, make sure it supports some form of session revocation.

Finally, if the `refresh_token` has expired or the session has been revoked, the API server will return a `401 Unauthorized` message to indicate that it will no longer support the token.

We've spent a considerable amount of time examining the OIDC protocol. Now, let's take an in-depth look at the `id_token`.

## **id\_token**

An `id_token` is a JSON web token that is base64-encoded and digitally signed. The JSON contains a series of attributes, known as claims, in OIDC. There are some standard claims in the `id_token`, but for the most part, the claims you will be most concerned with are as follows:

- `iss`: The issuer, which MUST line up with the issuer in your `kubectl` configuration
- `aud`: Your client ID
- `sub`: Your unique identifier
- `groups`: Not a standard claim, but it should be populated with groups specifically related to your Kubernetes deployment

Many deployments attempt to identify you by your email address. This is an anti-pattern, as your email address is generally based on your name, and names can change. The `sub` claim is supposed to be a unique identifier that is immutable and will never change. This way, it doesn't matter if your email changes because your name changes. While this can make it harder to debug "who is cd25d24d-74b8-4cc4-8b8c-116bf4abbd26?", it will provide a cleaner and easier-to-maintain cluster.

There are several other claims that indicate when an `id_token` should no longer be accepted. These claims are all measured in seconds from epoch (January 1, 1970) UTC time:

- `exp`: When the `id_token` expires
- `iat`: When the `id_token` was created
- `nbf`: The absolute earliest an `id_token` should be allowed

Why doesn't a token just have a single expiration time?

It's unlikely that the clock on the system that created the `id_token` has the exact same time as the system that evaluates it. There's often a skew and, depending on how the clock is set, it may be a few minutes. Having a not-before in addition to an expiration gives some room for standard time deviation.

There are other claims in an `id_token` that don't really matter but are there for additional context. Examples include your name, contact information, organization, and so on.

While the primary use for tokens is to interact with the Kubernetes API server, they are not limited to only API interaction. In addition to going to the API server, webhook calls may also receive your `id_token`.

You may have deployed OPA as a validating webhook on a cluster. When someone submits a pod creation request, the webhook will receive the user's `id_token`, which can be used to make decisions. **Open Policy Agent (OPA)**, is a tool to validate and authorize requests. It's often deployed in Kubernetes as an admission controller webhook. If you haven't worked with OPA or admission controllers, we cover both in depth, starting in *Chapter 11, Extending Security Using Open Policy Agent*.

One example of when an admission controller would inspect the user's `id_token` is that you want to ensure that the PVCs are mapped to specific PVs based on the submitter's organization. The organization is included in the `id_token`, which is passed to Kubernetes, and then onto the OPA webhook. Since the token has been passed to the webhook, the information can then be used in your OPA policies.

We've spent an extensive amount of time on OpenID Connect and how it's used to authenticate API calls to your Kubernetes cluster. While it may be the best overall option, it's not the only one. In the next section, we'll look at other options and when they would be appropriate.

## Other authentication options

In this section, we focused on OIDC and presented reasons why it's the best mechanism for authentication. It is certainly not the only option, and we will cover the other options in this section and when they're appropriate.

### Certificates

This is generally everyone's first experience authenticating to a Kubernetes cluster.

Once a Kubernetes installation is complete, a pre-built `kubectl config` file that contains a certificate and private key is created and ready to be used. Where this file is created is dependent on the distribution. This file should only be used in "break glass in case of emergency" scenarios, where all other forms of authentication are not available. It should be controlled by your organization's standards for privileged access. When this configuration file is used, it doesn't identify the user and can easily be abused, since it doesn't allow for an easy audit trail.

While this is a standard use case for certificate authentication, it's not the only use case for certificate authentication. Certificate authentication, when done correctly, is one of the strongest recognized credentials in the industry.

Certificate authentication is used by the US Federal Government for its most important tasks. At a high level, certificate authentication involves using a client key and certificate to negotiate your HTTPS connection to the API server. The API server can get the certificate you used to establish the connection and validate it against a **Certificate Authority (CA)** certificate. Once verified, it maps attributes from the certificate to a user and groups the API server can recognize.

To get the security benefits of certificate authentication, the private key needs to be generated on isolated hardware, usually in the form of a smartcard, and never leave that hardware. A certificate signing request is generated and submitted to a CA that signs the public key, thus creating a certificate that is then installed on the dedicated hardware. At no point does the CA get the private key, so even if the CA were compromised, you couldn't gain the user's private key. If a certificate needs to be revoked, it's added to a revocation list that can either be pulled from an **LDAP** directory or a file, or it can be checked using the **OCSP** protocol.

This may look like an attractive option, so why shouldn't you use certificates with Kubernetes?

- Smartcard integration uses a standard called **PKCS11**, which is not supported by either `kubectl` or the API server.

- The API server has no way of checking certificate revocation lists or using OCSP, so once a certificate has been minted, there's no way to revoke it. Since the API server can't revoke it, anyone who has it can continue to use it until it expires.

Additionally, the process to correctly generate a key pair is rarely used. It requires a complex interface to be built that is difficult for users to use, combined with command-line tools that need to be run. To get around this, the certificate and key pair are generated for you, and you download them or they're emailed to you, negating the security of the process.

The other reason you shouldn't use certificate authentication for users is that it's difficult to leverage groups. While you can embed groups into the subject of the certificate, you can't revoke a certificate. So if a user's role changes, you can give them a new certificate, but you can't keep them from using the old one. While you could reference users directly in your `RoleBindings` and `ClusterRoleBindings`, this is an anti-pattern that will make it difficult to keep track of access across even small clusters.

As stated in the introduction to this section, using a certificate to authenticate in “break glass in case of emergency” situations is a good use of certificate authentication. It may be the only way to get into a cluster if all other authentication methods experience issues.

After certificates, the next most common alternative is to use `ServiceAccount` tokens. We'll walk through that next and why you shouldn't use them from outside of your cluster.

## Service accounts

A `ServiceAccount` is designed to provide an identity to containers running in a cluster so that when those containers call the API server, they can be authenticated and have RBAC rules applied. Unfortunately, users began using the tokens associated with `ServiceAccount` objects to access the API server from outside of the cluster, which is problematic for multiple reasons:

- **Secure transmission of the token:** Service accounts are self-contained and need nothing to unlock them or verify ownership, so if a token is taken in transit, you have no way of stopping its use. You could set up a system where a user logs in to download a file with the token in it, but you now have a much less secure version of OIDC.
- **No expiration:** When you decode a legacy service account token, there is nothing that tells you when the token expires. That's because the token never expires. You can revoke a token by deleting the service account and recreating it, but that means you need a system in place to do that. Again, you've built a less capable version of OIDC.
- **Auditing:** The service account can easily be handed out by the owner once the key has been retrieved. If multiple users use a single key, it becomes very difficult to audit the use of the account.

Beginning in Kubernetes 1.24, static `ServiceAccount` tokens were disabled by default and replaced with short-lived tokens that are “projected” into your containers, using the `TokenRequest` API. We'll cover these tokens in more detail in the next section. We're including instructions here for how to generate static tokens as an example of an anti-pattern. While there are some narrow use cases where static tokens are useful, they should be avoided for use from outside of your cluster. They're most often used by pipelines, and later in this chapter, we will explore alternative approaches.

Service accounts appear to provide an easy access method. Creating them is easy. The following commands create a service account object and a secret to go with it that stores the service account's token:

```
kubectl create sa mysa -n default
kubectl create -n default -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
  name: mysa-secret
  annotations:
    kubernetes.io/service-account.name: mysa
type: kubernetes.io/service-account-token
EOF
```

The above steps:

1. Create a ServiceAccount object
2. Create a Secret with a token that is bound to the ServiceAccount

Next, the following command will retrieve the service account's token in the JSON format and return only the value of the token. This token can then be used to access the API server:

```
kubectl get secret mysa-secret -o json | jq -r '.data.token' | base64 -d
```

To show an example of this, let's call the API endpoint directly, without providing any credentials (make sure you use the port for your own local control plane):

```
curl -v --insecure https://0.0.0.0:6443/api
```

You will receive the following:

```
.
.
.
{
  "kind": "Status",
  "apiVersion": "v1",
  "metadata": {
  },
  "status": "Failure",
  "message": "forbidden: User \\"system:anonymous\\" cannot get path \"/api\"",
  "reason": "Forbidden",
  "details": {
  },
  "code": 403
* Connection #0 to host 0.0.0.0 left intact
```

By default, most Kubernetes distributions do not allow anonymous access to the API server, so we received a 403 error because we didn't specify a user.

Now, let's add our service account to an API request:

```
export KUBE_AZ=$(kubectl get secret mysa-secret -o json | jq -r '.data.token' | base64 -d)
curl -H "Authorization: Bearer $KUBE_AZ" --insecure https://0.0.0.0:6443/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.17.0.3:6443"
    }
  ]
}
```

Success! We were able to use a static `ServiceAccount` token to authenticate to our API server. As we said earlier, this is an anti-pattern. In addition to the issues with the token itself that we covered, you can't put a service account into arbitrary groups. This means that RBAC bindings have to either be direct to the service account or use one of the pre-built groups that service accounts are a member of. We'll explore why this is an issue when we discuss authorization, but here's an example of why this is an issue: directly binding means that in order to know if a user should have access, you need to process each binding, looking for the user instead of simply looking in an external database that has users organized into groups, which increases compliance burdens.

Finally, service accounts were never designed to be used outside of the cluster. It's like using a hammer to drive in a screw. With enough muscle and aggravation, you will drive it in, but it won't be pretty and no one will be happy with the result.

Now that we have covered how `ServiceAccount` tokens work, and that you shouldn't use them for users, we'll explore next why you should leverage the `TokenRequest` API to generate short-lived tokens for your `ServiceAccounts`.

## TokenRequest API

The `TokenRequest` API is how `ServiceAccount` tokens are generated in Kubernetes 1.24+. This API eliminates the use of static legacy service accounts and instead projects accounts into your pods. These projected tokens are short-lived and unique for each individual pod. Finally, these tokens become invalid once the pods they're associated with are destroyed. This makes service account tokens embedded into a pod much more secure.

This API provides another great feature: you can use it with third-party services. One example is using HashiCorp's Vault secret management system to authenticate pods without having to do a token review API call against the API server to validate it. We'll explore this approach when we get to *Chapter 8, Managing Secrets*.

This feature makes it much easier, and more secure, for your pods to call external APIs.

The `TokenRequest` API lets you request a short-lived service account for a specific scope. While it provides slightly better security, since it will expire and has a limited scope, it's still bound to a service account, which means no groups, and there's still the issue of securely getting the token to the user and auditing its use.

Starting in 1.24, all service account tokens are projected into pods via the `TokenRequest` API by default. The new tokens are good for a year though, so not very short-lived! That said, even if a token is set up to expire quickly, the API server won't reject it. It will log that someone is using an expired token. This is intended to make the transition from unlimited-life tokens to short-lived tokens easier.

Some people may be tempted to use tokens for user authentication. However, tokens generated by the `TokenRequest` API are still built for pods to talk to your cluster or to talk to third-party APIs; they are not meant to be used by users. In order to use them, you need to create them and securely transfer them. Since they're still bearer tokens, this could lead to a loss of the token and an eventual breach. If you're in a situation where you need to use them because there's no other technical option:

1. Make the tokens as short-lived as possible
2. Create an automated rotation process
3. Make sure your SIEM monitors these accounts usages outside of expected scenarios

Similar to static `ServiceAccount` tokens, there are use cases where you may need a token that can be used from outside the cluster, such as bootstrapping integrations or simple testing. The `kubectl` command now includes the `token` sub-command that can generate a short-lived token for a `ServiceAccount` without having to create a static `Secret`:

```
$ export KUBE_AZ=$(kubectl create token mysa -n default)
$ curl -H "Authorization: Bearer $KUBE_AZ" --insecure \
https://0.0.0.0:6443/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.17.0.3:6443"
    }
  ]
}
```

Our token from `kubectl` is good for an hour. This can be adjusted, but this is a much better approach for the few use cases where an external token is needed than creating a static token.

## Custom authentication webhooks

If you already have an identity platform that doesn't use an existing standard, a custom authentication webhook will let you integrate it without having to customize the API server. This feature is commonly used by cloud providers who host managed Kubernetes instances.

You can define an authentication webhook that the API server will call with a token to validate it and get information about the user. Unless you manage a public cloud with a custom IAM token system that you are building a Kubernetes distribution for, don't do this. Writing your own authentication is like writing your own encryption – just don't do it. Every custom authentication system we've seen for Kubernetes boils down to either a pale imitation of OIDC or "pass the password." Much like the analogy of driving a screw in with a hammer, you could do it, but it will be very painful. This is mostly because instead of driving the screw through a board, you're more likely to drive it into your own foot.

So far, we've focused on the fundamentals of Kubernetes authentication, looking at both the recommended patterns and antipatterns. Next, let's put that theory into practice by configuring authentication in a Kubernetes cluster.

## Configuring KinD for OpenID Connect

For our example deployment, we will use a scenario from our customer, FooWidgets. FooWidgets has a Kubernetes cluster that they would like integrated using OIDC. The proposed solution needs to address the following requirements:

- Kubernetes must use our central authentication system, Active Directory
- We need to be able to map Active Directory groups into our RBAC RoleBinding objects
- Users need access to the Kubernetes Dashboard
- Users need to be able to use the CLI
- All enterprise compliance requirements must be met
- Additional cluster management applications need to be managed centrally as well

Let's explore each of these in detail and explain how we can address the customer's requirements.

## Addressing the requirements

Our enterprise's requirements require multiple moving parts, both inside and outside our cluster. We'll examine each of these components and how they relate to building an authenticated cluster.

## Using LDAP and Active Directory with Kubernetes

Most enterprises today use Active Directory from Microsoft™ to store information about users and their credentials. Depending on the size of your enterprise, it's not unusual to have multiple domains or forests where users' data is stored.

We'll need a solution that knows how to talk to each domain. Your enterprise may have one of many tools and products for OpenID Connect integration, or you may just want to connect via LDAP. **LDAP**, the **Lightweight Directory Access Protocol**, is a standard protocol that has been used for over 30 years and is still the standard way to talk directly to Active Directory. Using LDAP, you can look up users and validate their passwords. It's also the simplest way to start because it doesn't require integration with an identity provider. All you need is a service account and credentials!

For FooWidgets, we're going to connect directly to our Active Directory for all authentication.

Don't worry – you don't need Active Directory ready to go to run this exercise. We'll walk through deploying a demo directory into our KinD cluster.

## Mapping Active Directory groups to RBAC RoleBindings

This will become important when we start talking about authorization. Active Directory lists all the groups a user is a member of in the `memberOf` attribute. We can read this attribute directly from our logged-in user's account to get their groups. These groups will be embedded into our `id_token` in the `groups` claim and can be referenced directly in RBAC bindings. This allows us to centralize the management of authorizations instead of having to manually manipulate RBAC bindings, simplifying management and decreasing the number of objects we need to manage and maintain in our cluster.

## Kubernetes Dashboard access

The Dashboard is a powerful way to quickly access information about your cluster and make quick updates. Unlike what is commonly thought about the dashboard's security, when deployed correctly, it does not create any security issues. The proper way to deploy the dashboard is with no privileges, instead relying on the user's own credentials. We'll do this with a reverse proxy that injects the user's OIDC token on each request, which the dashboard will then use when it makes calls to the API server. Using this method, we'll be able to constrain access to our dashboard the same way we would with any other web application.

There are a few reasons why using the `kubectl` built-in proxy and port-forward isn't a great strategy for accessing the dashboard. Many enterprises will not install CLI utilities locally, forcing you to use a jump box to access privileged systems such as Kubernetes, meaning port forwarding won't work. Even if you can run `kubectl` locally, opening a port on loopback (`127.0.0.1`) means anything on your system can use it, not just you from your browser. While browsers have controls in place to keep you from accessing ports on loopback using a malicious script, that won't stop anything else on your workstation. Finally, it's just not a great user experience.

We'll dig into the details of how and why this works in *Chapter 10, Deploying a Secured Kubernetes Dashboard*.

## Kubernetes CLI access

Most developers want to be able to access `kubectl` and other tools that rely on the `kubectl` configuration. For instance, the Visual Studio Code Kubernetes plugin doesn't require any special configuration. It just uses the `kubectl` built-in configuration. Most enterprises tightly constrain what binaries you're able to install, so we want to minimize any additional tools and plugins we want to install.

## Enterprise compliance requirements

Being cloud-native doesn't mean you can ignore your enterprise's compliance requirements. Most enterprises have requirements such as having 20-minute idle timeouts, multi-factor authentication for privileged access, and so on. Any solution we put in place has to make it through the control spreadsheets needed to go live. Also, and this goes without saying, everything needs to be encrypted (and I do mean everything).

## Pulling it all together

To fulfill these requirements, we're going to use **OpenUnison**. This has prebuilt configurations to work with Kubernetes, the Dashboard, the CLI, and Active Directory.

It's also pretty quick to deploy, so we don't need to concentrate on provider-specific implementation details and instead can focus on Kubernetes' configuration options. Our architecture will look like this:

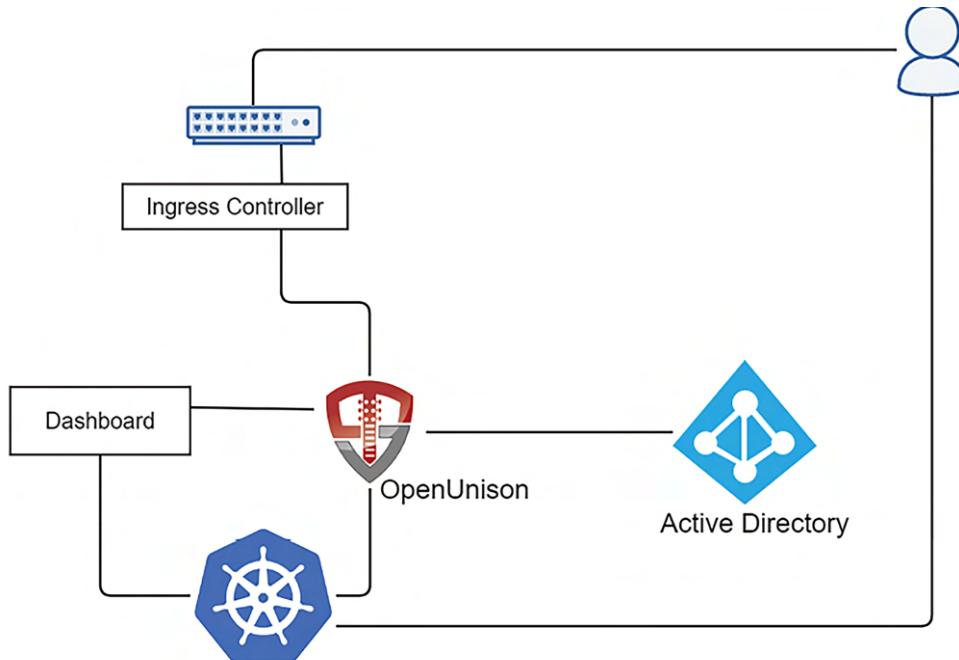


Figure 6.2: Authentication architecture

Although we're using an "Active Directory" in this instance, your enterprise might have an existing identity provider in place, such as **Okta**, **Entra** (formerly Azure Active Directory), **KeyCloak**, etc. In these instances, it's still a good idea to have an identity provider in-cluster to support not only SSO in your cluster but also your cluster management applications. As we continue through this book, we're going to be integrating monitoring systems, logging, GitOps systems, etc. It can be difficult from a management perspective to set up SSO with all of these applications, so having your own identity provider that you, as the cluster owner, control can give you greater flexibility over your clusters, making it easier to provide better security by integrating management applications with enterprise authentication, rather than relying on unauthenticated approaches like port-forwarding.

For our implementation, we're going to use two hostnames:

- `k8s.apps.X-X-X-X.nip.io`: Access to the OpenUnison portal, where we'll initiate our login and get our tokens
- `k8sdb.apps.X-X-X-X.nip.io`: Access to the Kubernetes dashboard

As a quick refresher, `nip.io` is a public DNS service that will return an IP address from the one embedded in your hostname. This is really useful in a lab environment where setting up DNS can be painful. In our examples, `X-X-X-X` is the IP of your Docker host.

When a user attempts to access `https://k8s.apps.X-X-X-X.nip.io/`, they'll be asked for their username and password. After the user hits submit, OpenUnison will look up the user against Active Directory, retrieving the user's profile information. At that point, OpenUnison will create user objects in the OpenUnison namespace to store the user's information and create OIDC sessions.

Earlier, we described how Kubernetes doesn't have user objects. Kubernetes lets you extend the base API with **Custom Resource Definitions (CRDs)**. OpenUnison defines a User CRD to help with high availability and to avoid needing a database to store state in. These user objects can't be used for RBAC.

Once the user is logged into OpenUnison, they can get their `kubectl` configuration to use the CLI or the Kubernetes Dashboard (<https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>) to access the cluster from their browser. Once the user is ready, they can log out of OpenUnison, which will end their session and invalidate their `refresh_token`, making it impossible for them to use `kubectl` or the dashboard until after they log in again. If they walk away from their desk for lunch without logging out, when they return, their `refresh_token` will have expired, so they'll no longer be able to interact with Kubernetes without logging back in.

Now that we have walked through how users will log in and interact with Kubernetes, we'll deploy OpenUnison and integrate it into the cluster for authentication.

## Deploying OpenUnison

We've automated the deployment for OpenUnison, so there aren't any manual steps. Since we want to start with a new cluster, we will delete the current cluster and execute the `create-cluster.sh` script in the `chapter2` folder to create a fresh KinD cluster. We have also added a script to the `chapter6` directory called `deploy_openunison_imp_noimpersonation.sh`. You can create the new cluster and integrate OIDC using the steps below:

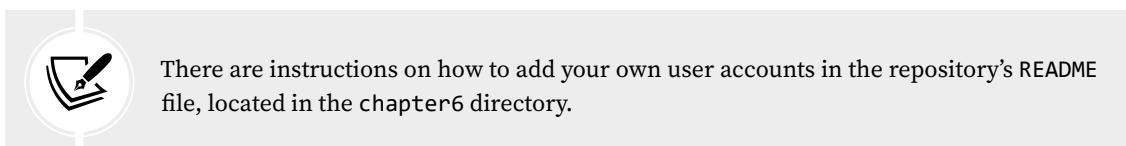
```
cd Kubernetes-An-Enterprise-Guide-Third-Edition/chapter2
kind delete cluster -n cluster01
./create-cluster.sh
cd ../chapter6/user-auth
./deploy_openunison_imp_noimpersonation.sh
```

This will take a few minutes, depending on your hardware. This script does several things:

1. Creates a stand-in “Active Directory” using a project called **ApacheDS**. You don’t need to know anything about ApacheDS other than it’s acting as our “Active Directory.”
2. Deploys the Kubernetes Dashboard version 2.7.
3. Downloads the `octl` utility and the OpenUnison helm charts.
4. Updates the `values.yaml` file for use with your Ubuntu VM’s IP.
5. Deploys OpenUnison.

You can log into the OIDC provider with any machine on your network by using the assigned `nip.io` address. Since we will test access using the dashboard, you can use any machine with a browser.

Navigate your browser to `network.openunison_host` in your `/tmp/openunison-values.yaml` file, which was created for you by running the above scripts. When prompted, use the username `mmosley` and the password `start123`, and then click on **Sign in**.



The screenshot shows a login interface. At the top center is the Tremolo Security logo, which consists of a stylized shield with a red and grey design. To the right of the logo, the words "TREMOLO SECURITY" are written in a bold, sans-serif font. Below the logo is a large, rounded rectangular button with the word "Login" in a large, bold, dark grey font. Underneath the "Login" button are two input fields. The first input field is labeled "User Name" and contains the text "mmosley". The second input field is labeled "Password" and contains six dots ("....."). At the bottom of the form are two red rectangular buttons: a smaller one on the left labeled "SIGN IN" and a larger one on the right labeled "RESET FORM".

Figure 6.3: OpenUnison login screen

When you do, you'll see this screen:

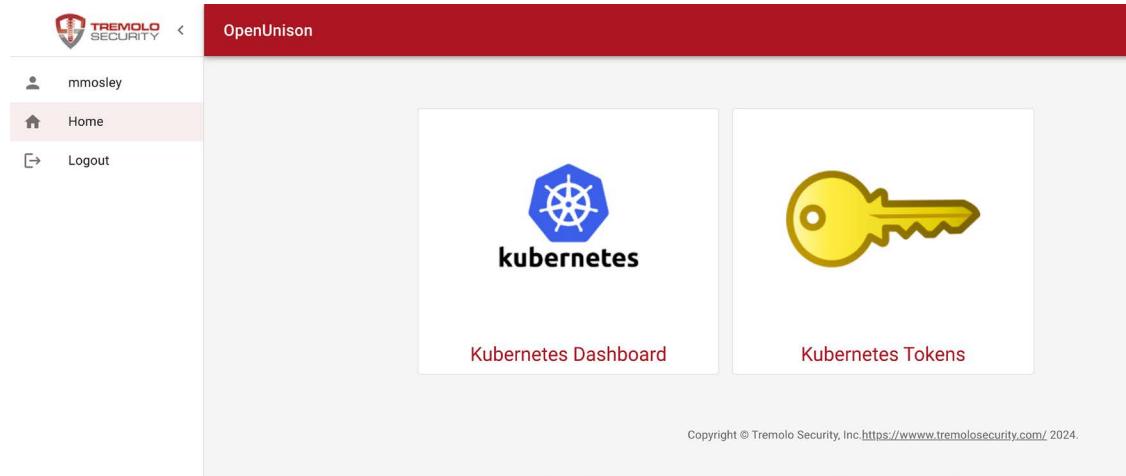


Figure 6.4: OpenUnison home screen

Let's test the OIDC provider by clicking on the **Kubernetes Dashboard** link. Don't panic when you see the initial dashboard screen – something like the following:

The screenshot shows the Kubernetes Dashboard. The top navigation bar includes a cluster dropdown set to "default", a search bar, and a user icon. The left sidebar is titled "Workloads" and lists several resource types: Cron Jobs, Daemon Sets, Deployments, Jobs, Pods, Replica Sets, Replication Controllers, Stateful Sets, Service, Ingresses, Ingress Classes, Services, Config and Storage, Config Maps, Persistent Volume Claims, Secrets, Storage Classes, Cluster, Cluster Role Bindings, Cluster Roles, Events, Namespaces, and Network Policies. The main content area displays tables for each of these resource types, showing columns such as Name, Images, Labels, Schedule, Suspend, Active, Last Schedule, Created, Node, Status, Restarts, CPU Usage (cores), Memory Usage (bytes), and Pods.

Figure 6.5: Kubernetes Dashboard before SSO integration has been completed with the API server

That looks like a lot of errors! We're in the dashboard, but nothing seems to be authorized. That's because the API server doesn't trust the tokens that have been generated by OpenUnison, yet. To resolve this, the next step is to tell Kubernetes to trust OpenUnison as its OpenID Connect Identity Provider.

## Configuring the Kubernetes API to use OIDC

At this point, you have deployed OpenUnison as an OIDC provider and it's working, but your Kubernetes cluster has not been configured to use it as a provider yet.

To configure the API server to use an OIDC provider, you need to add the OIDC options to the API server and provide the OIDC certificate so that the API will trust the OIDC provider.

Since we are using KinD, we can add the required options using a few `kubectl` and `docker` commands.

To provide the OIDC certificate to the API server, we need to retrieve the certificate and copy it over to the KinD master server. We can do this using two commands on the Docker host:

1. The first command extracts OpenUnison's TLS certificate from its secret. This is the same secret referenced by OpenUnison's Ingress object. We use the `jq` utility to extract the data from the secret and then Base64-decode it:

```
kubectl get secret ou-tls-certificate -n openunison -o json | jq -r  
'$.data["tls.crt"]' | base64 -d > ou-ca.pem
```

2. The second command will copy the certificate to the master server into the `/etc/kubernetes/pki` directory:

```
docker cp ou-ca.pem cluster01-control-plane:/etc/kubernetes/pki/ou-ca.pem
```

3. As we mentioned earlier, to integrate the API server with OIDC, we need to have the OIDC values for the API options. To list the options we will use, describe the `api-server-config` ConfigMap in the `openunison` namespace:

```
kubectl describe configmap api-server-config -n openunison  
Name:           api-server-config  
Namespace:      openunison  
Labels:         <none>  
Annotations:    <none>  
Data  
=====  
oidc-api-server-flags:  
----  
--oidc-issuer-url=https://k8sou.apps.192-168-2-131.nip.io/auth/idp/k8sIdp  
--oidc-client-id=Kubernetes  
--oidc-username-claim=sub  
--oidc-groups-claim=groups  
--oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
```

- Next, edit the API server configuration. OpenID Connect is configured by changing flags on the API server. This is why managed Kubernetes generally doesn't offer OpenID Connect as an option, but we'll cover that later in this chapter. Every distribution handles these changes differently, so check with your vendor's documentation. For KinD, shell into the control plane and update the manifest file:

```
docker exec -it cluster01-control-plane bash
apt-get update
apt-get install vim -y
vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

- Add the flags from the output of the ConfigMap under command. Make sure to add spacing and a dash (-) in front. Make sure to update the URLs to match yours. It should look something like this when you're done:

```
- --kubelet-preferred-address-types=InternalIP,ExternalIP,Hostname
- --oidc-issuer-url=https://k8sou.apps.192-168-2-131.nip.io/auth/idp/
k8sIdp
- --oidc-client-id=Kubernetes
- --oidc-username-claim=sub
- --oidc-groups-claim=groups
- --oidc-ca-file=/etc/kubernetes/pki/ou-ca.pem
- --proxy-client-cert-file=/etc/kubernetes/pki/front-proxy-client.crt
```

- Exit vim and the Docker environment (*Ctrl + D*), and then take a look at the api-server pod:

kubectl get pod kube-apiserver-cluster01-control-plane -n kube-system			
NAME	READY	STATUS	RESTARTS
AGE			
kube-apiserver-cluster-auth-control-plane	1/1	Running	0
73s			

Note that it's only 73s old. That's because KinD saw that there was a change in the manifest and restarted the API server.

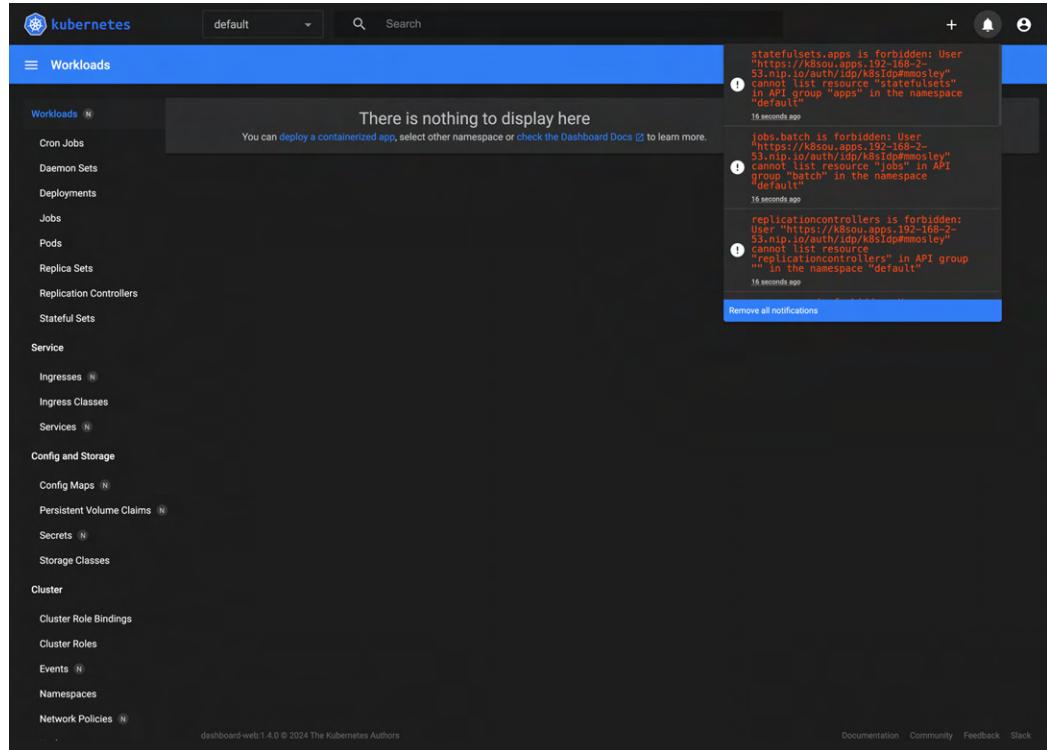
The API server pod is known as a static pod. This pod can't be changed directly; its configuration has to be changed from the manifest on disk. This gives you a process that's managed by the API server as a container, but without giving you a situation where you need to edit pod manifests in etcd directly if something goes wrong.

Once you have updated your API server flags, the next step is to verify that you can now log in to your cluster. Let's walk through those steps next.

## Verifying OIDC integration

Once OpenUnison and the API server have been integrated, we need to test that the connection is working:

1. To test the integration, log back into OpenUnison and click on the **Kubernetes Dashboard** link again.
2. Click on the bell in the upper right and you'll see a different error:



*Figure 6.6: SSO enabled but the user is not authorized to access any resources*

SSO between OpenUnison and Kubernetes is working! However, the new error, `service is forbidden: User https://...`, is an authorization error, **not** an authentication error. At this point, the API server knows who we are but isn't letting us access the APIs.

3. We'll dive into the details of RBAC and authorizations in the next chapter, but for now, create this RBAC binding:

```
kubectl create -f - <<EOF
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: ou-cluster-admins
subjects:
- kind: Group
  name: cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com
  apiGroup: rbac.authorization.k8s.io
roleRef:
```

```

kind: ClusterRole
name: cluster-admin
apiGroup: rbac.authorization.k8s.io
EOF
clusterrolebinding.rbac.authorization.k8s.io/ou-cluster-admins created

```

- Finally, go back to the Dashboard, and you'll see that you have full access to your cluster and all the error messages are gone.

The API server and OpenUnison are now connected. Additionally, an RBAC policy has been created to enable our test user to manage the cluster as an administrator. Access was verified by logging into the Kubernetes Dashboard, but most interactions will take place using the `kubectl` command. The next step is to verify that we're able to access the cluster using `kubectl`.

## Using your tokens with `kubectl`

This section assumes you have a machine on your network that has a browser and `kubectl` running.

Using the Dashboard has its use cases, but you will likely interact with the API server using `kubectl`, rather than the Dashboard, for the majority of your day. In this section, we will explain how to retrieve your JWT and how to add it to your Kubernetes config file so that you can use `kubectl`:

- You can retrieve your token from the OpenUnison dashboard. Navigate to the OpenUnison home page and click on the key that says **Kubernetes Tokens**. You'll see a screen that looks as follows:

**Kubernetes kubectl command**

Use this kubectl command to set your user in .kubectl/config. Refresh this screen to generate a new set of tokens. Logging out will clear all of your sessions.

**refresh\_token**

Run the kubectl command to set your user-context and server connect:

```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CERTIFICATE-----\r\nMIIDz+zCCAUoGwIBAgIGAZFBpT7aMA0GCSqGSIb3DQEBCwUAMIGCMScwJQYDV0QD\r\nDB5rHNNvdS5hchBzLjESM10xNjgtM101hySuaXaua8xExzARBgNVBAsCkt1YmVy\r\nbmVZQXh0jAMBGNVBAoBU15T3jnMRlwEOYDVQ0HDApNeSB0hdVzdgVymQkwBwYD\r\nVQ0QIDAAXEjAQBGNVBAYTUCU15Q291nRyeTAefw0yND4MTYNDQ0MDFaFw0tAT4\r\nMTYnDQ0MDfAMIGCMScwJQYDV0Q0DB5rHNNvdS5hchBzLjESM10xNjgtM101My5u\r\naxXuaa8xEzARBGNVBAoBzCkt1YmVby0Zk0XkxjAMBGNVBAoBU15T3jnMRlwEOYD\r\nVQ0QHDApIe5B0bhVzdgVymQkwBwYDQ0QIDAAXEjAQBgnNVBAYTCU15Q291bnRyeTCC\r\nA51w0QYJkZ1hvhcNAQEBBQAdggEPADCAQcCgqBAMKKPXvCU0XtMkLfQ0q9dk4J\r\ntzqmvounaYezpGhpWPUlZaa5yCnLzWlWx0kep10fSTzW11n07n0Lw/giwl24\r\nt0WfqnAd1QkOx3J1Pp8w1z8Ty13UbLBVwyJn15rYcrxgLo150aDc75Rh\r\r9Huyd6jxy83robjEDB0tknJt4pGn24zuK6jNKFHEGrHGmrjd0j0/c7/2Infq\rj7D+209zkK4Cc/w/fvEOCt39ZRpxxTSVj8avkRo0Ym+zcz7xyBLTq9/X0/Qgoofe\rxmAyhLZZD72fp+33abATV6e7VMGjhKFTNU96RwNXICiiIxHuSSbYDeq1mV3b10C\rAwAAAa01MMhxFgYDVR0LAQ/BawGcYIKwBBQUHAEwDgYDV0RFPAHQ/BAQDAg9Q\rMEKA1Q0RCMEEChms4c291lmFwCMH0tKtyLT20C0ylTU2L5pc5p4Teazhz\rZG1uyXBwcy4x0T1MTY4L1TNTMu0mlwLm1vMA0GCSqGSIb3D0EBCwUA41BAQC7\rSFs+AfwNP7Pgjgj0Esq2g8t85Pb4kRBGqvLSY43pr0thType++Yba+bFkrzVwLj\rxtJ9DzaAHg6EE5kyxQMg4FwdCujUX0DL1KHdq+rJa+GGqCYThfNc6fknrhAj\rCsxnknCPry/4Gvh0/oYgwfRVEejG+g20k9AO1G8jf/e8Zx+Hq4lwL7568jRwENka\rwPQpUDrpwHouc2mlw20qgAlZoH120pk0f849CV39NFD109Ex0q1NpjKhcSiud\rRHRSSq0su9Weux9zXtb08bratSrCeUVPsj3BcAYAY9c1cdR5FgNvbnn7ndwQY\rOnqV3jR0VwsxN9V2fvgw\r-----END CERTIFICATE-----
```

**id\_token**

**cert**

Figure 6.7: OpenUnison `kubectl` configuration tool

OpenUnison provides a command line that you can copy and paste into your host session that adds all the required information to your config.

2. First, click on the double documents button next to the **kubectl Command** (or **kubectl Windows Command** if you're on Windows) to copy your `kubectl` command into your buffer. Leave the web browser open in the background.
3. You may want to back up your original config file before pasting the `kubectl` command from OpenUnison:

```
export KUBECONFIG=$(mktemp)
kubectl get nodes
W0804 13:43:26.624417 3878806 loader.go:222] Config not found: /tmp/tmp.tqcXxwBh0H
to the server localhost:8080 was refused - did you specify the right host
or port?
```

4. Then, go to your host console and paste the command into the console (the following output has been shortened, but your paste will start with the same output):

```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CER. . .
Cluster "no-impersonation" set.
Context "no-impersonation" created
User "mmosley@no-impersonation" set.
Switched to context "no-impersonation".
```

5. Now, verify that you can view the cluster nodes using `kubectl get nodes`:

```
kubectl get nodes
NAME                  STATUS   ROLES      AGE      VERSION
cluster01-control-plane  Ready    control-plane  7m47s  v1.27.3
cluster01-worker        Ready    <none>     7m26s  v1.27.3
```

6. You're now using your login credentials instead of the master certificate! As you work, the session will refresh. You can verify your identity using the `kubectl auth whoami` command:

```
kubectl auth whoami
ATTRIBUTE  VALUE
Username    https://k8sou.apps.192-168-2-82.nip.io/auth/idp/
k8sIdp#mmosley
Groups      [cn=group2,ou=Groups,DC=domain,DC=com cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com system:authenticated]
```

This command will tell you who the API server thinks you are, including your groups. This can be very handy when debugging authorizations.



When I first began working with Kubernetes in 2015, the first issue I opened was for this feature while I was debugging the integration of OpenUnison and Kubernetes. I was thrilled to see it implemented initially in 1.0.26 and for it to go GA in 1.0.28. It's a beta feature in 1.27, and we have pre-configured our KinD cluster to support it. If you want to use this feature with other clusters, you may need to work with your vendor, since it requires a command-line argument for the API server.

7. Log out of OpenUnison and watch the list of nodes. Within a minute or two, your token will expire and no longer work:

```
kubectl get nodes  
Unable to connect to the server: failed to refresh token: oauth2: cannot  
fetch token: 401 Unauthorized
```

Congratulations! You've now set up your cluster so that it does the following:

- Authenticates using LDAP, using your enterprise's existing authentication system
- Uses groups from your centralized authentication system to authorize access to Kubernetes (we'll get into the details of how in the next chapter)
- Gives user access to both the CLI and the dashboard using the centralized credentials
- Maintains your enterprise's compliance requirements by having short-lived tokens that provide a way to time out
- Ensures everything uses TLS, from the user's browser to the Ingress Controller, to OpenUnison, the Dashboard, and finally, the API server

You've integrated most of the advice from this chapter into your cluster. You've also made it easier to access because you don't need to have a pre-configured configuration file anymore.

Next, you'll learn how to integrate centralized authentication into your managed clusters.

## Introducing impersonation to integrate authentication with cloud-managed clusters

It's very popular to use managed Kubernetes services from cloud vendors such as Google, Amazon, Microsoft, and DigitalOcean (among many others).

When it comes to these services, they are generally very quick to get up and running, and they all share a common thread: they mostly don't support OpenID Connect (Amazon's EKS does support OpenID Connect now, but the cluster must be running on a public network and have a commercially signed TLS certificate).

Earlier in this chapter, we talked about how Kubernetes supports custom authentication solutions through webhooks and that you should never, ever, use this approach unless you are a public cloud provider or some other host of Kubernetes systems. It turns out that pretty much every cloud vendor has its own approach to using these webhooks that uses its own identity and access management implementations. In that case, why not just use what the vendor provides? There are several reasons why you may not want to use a cloud vendor's IAM system:

- **Technical:** You may want to support features not offered by the cloud vendor, such as the dashboard, in a secure fashion.
- **Organizational:** Tightly coupling access to managed Kubernetes with that cloud's IAM puts an additional burden on the cloud team, which means that they may not want to manage access to your clusters.
- **User experience:** Your developers and admins may have to work across multiple clouds. Providing a consistent login experience makes it easier for them and requires learning fewer tools.
- **Security and compliance:** The cloud implementation may not offer choices that line up with your enterprise's security requirements, such as short-lived tokens and idle timeouts.

All that being said, there may be reasons to use the cloud vendor's implementation. However, you'll need to balance out the requirements. If you want to continue to use centralized authentication and authorization with hosted Kubernetes, you'll need to learn how to work with Impersonation.

## What is Impersonation?

Kubernetes **Impersonation** is a way of telling the API server who you are without knowing your credentials or forcing the API server to trust an OpenID Connect IdP. This is useful when you can't configure OpenID Connect, as is generally the case with managed Kubernetes offerings, or you want to support multiple access from multiple identity providers.

When you use `kubectl`, instead of the API server receiving your `id_token` directly, it will receive a service account or identifying certificate that will be authorized to impersonate users, as well as a set of headers that tell the API server who the proxy acts on behalf of:

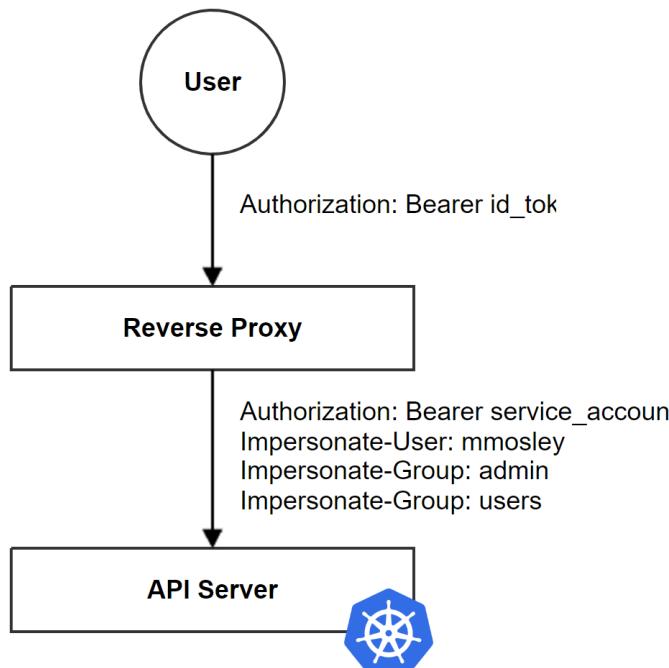


Figure 6.8: Diagram of how a user interacts with the API server when using Impersonation

The reverse proxy is responsible for determining how to map from the `id_token`, which the user provides (or any other token, for that matter) to the `Impersonate-User` and `Impersonate-Group` HTTP headers. The dashboard should never be deployed with a privileged identity, which the ability to impersonate falls under.

To allow Impersonation with the 2.x dashboard, use a similar model, but instead of going to the API server, you go to the dashboard:

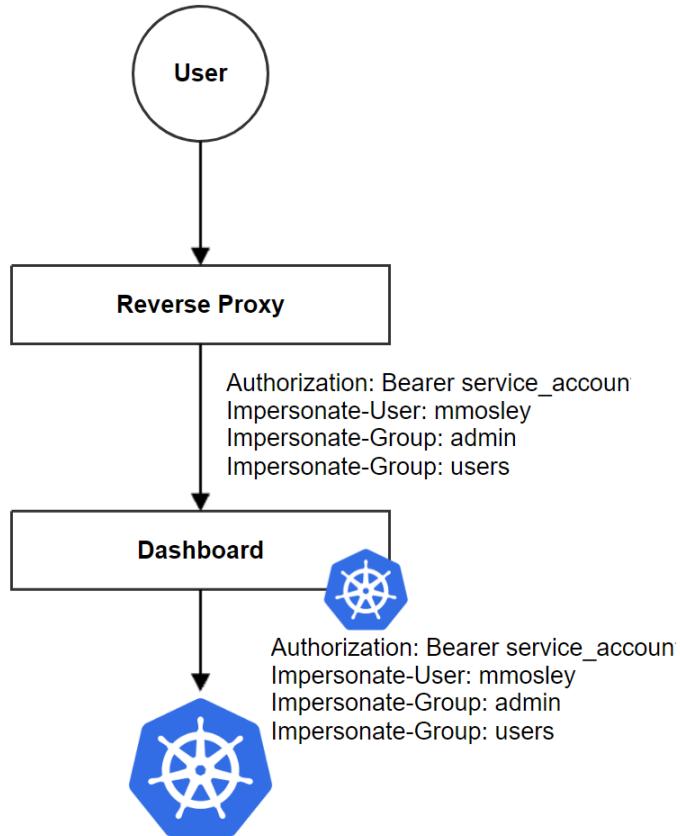


Figure 6.9: Kubernetes Dashboard with Impersonation

The user interacts with the reverse proxy just like any web application. The reverse proxy uses its own service account and adds the impersonation headers. The dashboard passes this information through to the API server on all requests. The dashboard never has its own identity.

Now that we see what impersonation is, and how it can help us secure access to the Kubernetes Dashboard and Kubernetes APIs, we'll walk through what you need to think about from a security perspective when implementing it.

## Security considerations

The service account has a certain superpower: it can be used to impersonate **anyone** (depending on your RBAC definitions). If you’re running your reverse proxy from inside the cluster, a service account is OK, especially if combined with the `TokenRequest` API to keep the token short-lived.

Earlier in the chapter, we talked about the legacy tokens for `ServiceAccount` objects having no expiration. That’s important here because if you’re hosting your reverse proxy off-cluster, then if it were compromised, someone could use that service account to access the API service as anyone. Make sure you’re rotating that service account often. If you’re running the proxy off-cluster, it’s probably best to use a shorter-lived certificate instead of a service account.

When running the proxy on a cluster, you want to make sure it’s locked down. It should run in its own namespace at a minimum, not `kube-system` either. You want to minimize the number of people who have access. Using multi-factor authentication to get to that namespace is always a good idea, as is using network policies that control what pods can reach out to the reverse proxy.

Based on the concepts we’ve just learned about regarding impersonation, the next step is to update our cluster’s configuration to use impersonation instead of using OpenID Connect directly. You don’t need a cloud-managed cluster to work with impersonation.

## Configuring your cluster for impersonation

Let’s deploy an impersonating proxy for our cluster. Just like integrating our cluster directly into OpenUnison using OpenID Connect, we’ve automated the deployment so that you don’t need to manually configure OpenUnison. We’ll clear out our old cluster and start afresh:

```
cd Kubernetes-An-Enterprise-Guide-Third-Edition/chapter2
kind delete cluster -n cluster01
./create-cluster.sh
cd ../chapter6/user-auth
./deploy_openunison_imp_implementation.sh
```

The differences between this script and our original script are:

- Configuring OpenUnison to generate `NetworkPolicy` objects to limit access to just requests from our NGINX Ingress controller and the API server
- Configuring OpenUnison’s `ServiceAccount` token to only be valid for 10 minutes instead of the typical hour or day
- Configuring the OpenUnison `values.yaml` to deploy the `kube-oidc-proxy` to handle incoming API server requests
- Creating the `cluster-admin ClusterRoleBinding` so that your user can work with your cluster

Once the script is finished running, you can log in with the same account, `mmosley`, as before.

The OpenUnison helm charts create NetworkPolicies and constraints the lifetime of its ServiceAccount token to line up with the security best practices we discussed above. It's important that we keep any system that shouldn't interact with our impersonation proxy from doing so to cut down on the potential attack surface, ensuring that any tokens that are minted with the ability to impersonate other users expire quickly.

Next, we'll walk through testing our impersonation-based integration.

## Testing Impersonation

Now, let's test our Impersonation setup. Follow these steps:

1. In a browser, enter the URL for your OpenUnison deployment. This is the same URL you used for your initial OIDC deployment.
2. Log into OpenUnison, and then click on the dashboard.
3. Click on the little circular icon in the upper right-hand corner to see who you're logged in as.
4. Next, go back to the main OpenUnison dashboard and click on the **Kubernetes Tokens** badge.
5. Note that the `--server` flag being passed to `kubectl` no longer has an IP. Instead, it has the hostname from `network.api_server_host` in the `/tmp/openunison-values.yaml` file. This is Impersonation. Instead of interacting directly with the API server, you're now interacting with `kube-oidc-proxy`'s reverse proxy.
6. Finally, let's copy and paste our `kubectl` command from the OpenUnison tokens screen into a shell:

```
export TMP_CERT=$(mktemp) && echo -e "-----BEGIN CERTIFI...
Cluster "impersonation" set.
Context "impersonation" created.
User "mmosley@impersonation" set.
Switched to context "impersonation".
```

7. To verify you have access, list the cluster nodes:

```
kubectl get nodes
NAME                  STATUS   ROLES      AGE      VERSION
cluster01-control-plane  Ready    control-plane  37m     v1.27.3
cluster01-worker        Ready    <none>     37m     v1.27.3
```

8. Just as with OIDC integration, you can use `kubectl auth whoami` to verify your identity:

```
kubectl auth whoami
ATTRIBUTE  VALUE
Username    mmosley
Groups      [cn=group2,ou=Groups,DC=domain,DC=com cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com system:authenticated]
```

The main difference between your identity when using impersonation instead of OIDC integration is that your username doesn't have the identity provider's URL in the front.

- Just like when you integrated the original deployment of OpenID Connect, once you've logged out of the OpenUnison page, within a minute or two, the tokens will expire and you won't be able to refresh them:

```
kubectl get nodes  
Unable to connect to the server: failed to refresh token: oauth2: cannot  
fetch token: 401 Unauthorized
```

You've now validated that your cluster works correctly with Impersonation. Instead of authenticating directly to the API server, the impersonating reverse proxy (OpenUnison) forwards all requests to the API server with the correct impersonation headers. You're still meeting your enterprise's needs by providing both a login and logout process and integrating your Active Directory groups.

You'll also notice that you can now access your cluster from any system on your network! This might make doing the rest of the examples throughout the book easier.

Impersonation is far more than accessing your cluster. Next, we'll look at how to use impersonation from `kubectl get` debug authorization policies.

## Using Impersonation for Debugging

Impersonation can be used to debug authentication and authorization configurations. This will become more useful when you begin writing RBAC policies. As an administrator, you can use impersonation from the `kubectl` command by adding the `--as` and `--as-groups` parameters to run a command as someone else. For instance, if you ran `kubectl get nodes` as a random user from your command line, it would fail:

```
kubectl get nodes --as somerandomuser  
Error from server (Forbidden): nodes is forbidden: User "somerandomuser" cannot  
list resource "nodes" in API group "" at the cluster scope
```

However, if we add our administrative group:

```
kubectl get nodes --as somerandomuser --as-group=cn=k8s-cluster-admins,ou=Groups,  
s,DC=domain,DC=com  
NAME STATUS ROLES AGE VERSION  
cluster01-control-plane Ready control-plane 17m v1.27.3  
cluster01-worker Ready <none> 17m v1.27.3
```

You can see that it worked. That's because the API server interpreted our user as being a member of the group `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com`, which we created an RBAC binding for. In fact, if we run `kubectl auth whoami` with these parameters, we'll just see how the API server sees us:

```
kubectl auth whoami --as=someuser --as-group=cn=k8s-cluster-admins,ou=Groups,DC  
=domain,DC=com
```

ATTRIBUTE	VALUE
-----------	-------

```

Username                      someuser
Groups                         [cn=k8s-cluster-
 admins,ou=Groups,DC=domain,DC=com system:authenticated]
Extra: originaluser.jetstack.io-groups [cn=group2,ou=Groups,DC=domain,DC=com
cn=k8s
cluster-admins,ou=Groups,DC=domain,DC=com]
Extra: originaluser.jetstack.io-user   [mmosley]

```

In the above example, the API server sees the request as being from *someuser*, with the appropriate group based on the impersonation headers sent by `kubectl`.

The additional `Extra` attributes are there because while we are performing an impersonation request from `kubectl` -> the `kube-oidc-proxy`, `kube-oidc-proxy` does a separate impersonation with new headers, adding the `extra-info` headers to be included, so the audit logs show who the original user who made the request was. Before forwarding the request to the API server, the `kube-oidc-proxy` first performs a `SubjectAccessReview` to make sure that the user `mmosley` with their groups is allowed to impersonate `someuser` and the group.

We were able to quickly configure impersonation using OpenUnison, where most of the details of the implementation were hidden from you. What if you want to configure an impersonating proxy without OpenUnison?

## Configuring Impersonation without OpenUnison

OpenUnison automated a couple of key steps to get impersonation working. You can use any reverse proxy that can generate the correct headers. There are three critical items to understand when doing this on your own: RBAC, default groups, and inbound impersonation.

## Impersonation RBAC policies

RBAC will be covered in the next chapter, but for now, the correct policy to authorize a service account for Impersonation is as follows:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
- apiGroups:
  - ""
resources:
- users
- groups
verbs:
- impersonate

```

To constrain what accounts can be impersonated, add `resourceNames` to your rule. For instance, if you only want to allow the impersonation of the user `mmosley`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: impersonator
rules:
- apiGroups:
  - ""
  resources:
  - users    resourceNames:
  - mmosley
  verbs:
  - impersonate
```

The first `ClusterRole` above is what tells Kubernetes that a member can impersonate all users and groups (or specific users or groups if `resourceNames` is specified). Be very careful as to which accounts are granted this `ClusterRole`, as it makes you essentially a `cluster-admin` because you could impersonate the `system:masters` group, for example, bypassing RBAC and allowing anyone who is authorized by this role to become a global administrator and compromise your cluster however they wish.

When configuring the impersonation of specific users and groups, break the `ClusterRole` into one `ClusterRole` for each. This way, you won't have someone impersonating a group with the name of the user creating unintended consequences.

With RBAC having been configured, the next requirement is adding default groups to impersonation requests.

## Default groups

When impersonating a user, Kubernetes does not add the default group, `system:authenticated`, to the list of impersonated groups. When using a reverse proxy that doesn't specifically know to add the header for this group, configure the proxy to add it manually. Otherwise, simple acts such as calling the `/api` endpoint will fail, as this will be unauthorized for anyone except cluster administrators.

We've focused the bulk of this chapter on authenticating users who will interact with the API server. A major advantage of Kubernetes and the APIs it provides is to automate your systems. Next, we'll look at how you apply what we've learned so far to authenticating those automated systems.

## Inbound Impersonation

We've shown how to use `kubectl` with the `--as` and `--as-group` parameters to impersonate users for debugging. If you're using impersonation to manage access to your clusters, how does your impersonating proxy know that the user attempting to impersonate another user is, in fact, authorized to do so? In Kubernetes, you need to build a `ClusterRole` and a `ClusterRoleBinding` to enable impersonation for a specific user to a specific user, but how does your proxy know that you can impersonate someone?

In our previous example:

```
kubectl auth whoami --as=someuser --as-group=cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com
ATTRIBUTE          VALUE
Username           someuser
Groups             [cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com system:authenticated]
Extra: originaluser.jetstack.io-groups [cn=group2,ou=Groups,DC=domain,DC=com cn=k8s
cluster-admins,ou=Groups,DC=domain,DC=com]
Extra: originaluser.jetstack.io-user   [mmosley]
```

We see `mmosley` impersonated `someuser`. Kubernetes allowed this impersonation because `mmosley` is a member of the group `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com`, which has a `ClusterRoleBinding` to the `cluster-admin` `ClusterRole`. However, this request went through `kube-oidc-proxy`, so how did `kube-oidc-proxy` know that the cluster would authorize the request? On each request to `kube-oidc-proxy` that includes impersonation headers, a `SubjectAccessReview` is created to check if `mmosley` is allowed to impersonate `someuser`. If this check fails, the `kube-oidc-proxy` denies the request.

Your impersonating proxy will need to make the same choice. There are three approaches:

- **Delete and ignore all inbound impersonation headers:** Your proxy will ignore and remove all inbound headers for impersonation, making the `--as` and `--as-group` flags useless. This locks down access but limits functionality.
- **Maintain a custom authorization scheme:** Before generating impersonation headers, a proxy can have its own authorization system to determine which users are allowed to impersonate other users. This means maintaining an additional authorization system, which can lead to issues with misconfiguration and, eventually, breaches.
- **Query Kubernetes for Authorization Decisions:** This is what `kube-oidc-proxy` and Pinniped (a tool from VMware that serves a similar role to OpenUnison) use to make sure that the inbound impersonation is authorized. This is best, as it uses the same rules as your cluster to manage access, simplifying management and making it less likely a misconfiguration will lead to a breach.

Even once you've authorized an inbound impersonation, it's important to log that the impersonation has occurred. The `kube-oidc-proxy` project does this in two places:

- **Proxy logs:** Each inbound impersonation gets logged to the console (which should be captured by a log aggregator)
- **API Server Audit Logs:** The extra-info headers tell the API server who the original user was which is included in the audit logs. We'll see how to set up and inspect the audit logs in the next section.

Inbound impersonation is a very difficult process to manage. If it's something you want to allow, you should stick with a purpose-built impersonating proxy. Otherwise, it's best to just strip all inbound impersonation headers to avoid an account takeover.

So far, we've only discussed users as a whole, without adding any context. Many enterprises require that users who interact with a cluster to perform administrative work have privileges beyond their typical account. Next, we'll look at how to implement privileged access management in Kubernetes.

## Privileged Access to Clusters

In addition to managing authentication, most enterprises require a concept of "privileged access management," where not only is access limited by the user but also by time. Most enterprises require a change control process of some kind to ensure that changes to production systems are tracked and approved. This requirement generally comes from any of the various compliance and regulatory frameworks needed in large enterprises.

There are generally three ways to manage privileged access in Kubernetes, and we'll cover all three with their benefits and drawbacks.

### Using a Privileged User Account

It is common for enterprises to require that administrators have two accounts, one for day-to-day tasks and one to make administrative changes. This approach is generally implemented using a **Privilege Access Manager (PAM)** that generates a new password for the user when they're authorized to do their work. This approach enables compliance with most frameworks because there's a process by which someone has to approve the administrative account's unlocking inside of the PAM. Once the admin has completed their work, they check the account back into the PAM, which locks it. Alternatively, a time limit is usually set for how long the account can be checked out for, and when that time expires, the account is automatically locked by the PAM.

The major benefit of this approach is that the management of the privileged accounts is done outside of Kubernetes. It's someone else's responsibility and eliminates something that cluster owners need to manage, either via the previously mentioned PAM or some other engine. It's important to note that as the cluster manager, you're still responsible for authorizing access, so the same recommendations from this chapter apply.

Another reason for this approach is to protect against phishing attacks against administrators. For instance, if your cluster is integrated with your Active Directory in a way that allows desktop SSO, a bad actor could send your admins an email that runs a command as that user, without having to even know the admin's credentials! If you are at least forcing a password, there's an additional step an attacker needs to take.

There are arguments to be made that this isn't the most efficient or secure approach, but it's often what's already in place. You will find it much easier to work within existing frameworks than trying to reinvent them.

## Impersonating a Privileged User

Instead of using an external PAM to unlock users via passwords, another approach is to impersonate a privileged user using the `--as` command line parameter for `kubectl`. The idea is to simulate the Unix `sudo` command to escalate your privileges, to protect against accidental administrative actions.

This approach is more likely to do more harm than good. To make it work, you need at least one RBAC `ClusterRole` and `ClusterRoleBinding` for each user to maintain individual privileged accounts. If you have 100 admins, that's 200 additional objects to create even before you authorize access to resources. In addition to creating those objects, you need to delete them when the time comes. While automation can help, the proliferation of objects makes it easier to hide misconfigurations. The fewer objects, the better.

Any security that is too complicated to be easily tracked is more likely to create security holes. In this case, you may decide you're going to cut down on the objects created by only creating one `ClusterRole` for impersonation and a single `ClusterRoleBinding` with multiple `Subjects`. This doesn't really cut down on the complexity of managing this solution because:

- You still have to manage a `Subjects` list that can grow very quickly
- Your privileged users all now look to have the same identity as the API server, losing a considerable amount of granularity and value

It's important to note that the API server does track and log who the original requestor was, but that's now in a different field which your systems need to look for.

This additional work provides little, if any, benefit. You can't effectively time-box access without some kind of additional automation, and just requiring the addition of a command-line parameter to `kubectl` isn't likely to stop someone from hitting the up arrow to find the previous command they ran that has the `--as` parameter, even if they didn't mean to.

This approach is more trouble than it's worth. It won't provide any meaningful security but will complicate your cluster's management in ways that are more likely to create security holes than plug them.

## Temporarily Authorizing Privilege

Assuming you write your RBAC policies based on groups, the only thing you really need to do to escalate privileges is temporarily assign a user to a privileged group. The workflow would look similar to using a privileged account, but instead of having an entirely separate account, you use your standard account. As an example, in our current cluster, let's assume that `mmosley` was NOT a member of the AD group `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com`. An external workflow engine would add them after the approval of whatever work needs to be done. Once provisioned, `mmosley` performs their tasks, and when completed, their membership to `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com` is revoked.

This gives us the same benefits of having a privileged account, without having to have an additional account to manage. There are multiple risks associated with this approach:

- Phishing: If you're using your standard account that's used for everyday tasks like email, then there's a higher risk that your credentials will be stolen.

- Overstaying your welcome: A long-lived credential, such as a token or a certificate, may grant access beyond when policy dictates that access has expired.

To combat these risks, it's important that privileged users are:

- Required to re-authenticate: Making sure that the administrator has to re-enter credentials helps protect against malicious scripts and executables.
- Use multi-factor authentication: Requiring an admin to provide a second factor, preferably one that can't be phished, will protect against most attacks.
- Use short-lived tokens: What's the point of a four-hour change window if your token is good for eight hours?

With these additional mitigations in place, privileged authorization puts the least amount of work on cluster owners because everything is externalized. Just authorize by group!

While this provides the best user experience, most large enterprises are likely to have a privileged access manager already, making that the most likely approach.

Having walked through multiple ways of authenticating users who interact with our clusters, the next step is to look at how pipelines and automation need to authenticate.

## Authenticating from pipelines

This chapter so far has focused exclusively on authentication to Kubernetes by users. Whether an operator or a developer, a user will often interact with a cluster to update objects, debug issues, view logs, and so on. However, this doesn't quite handle all use cases. Most Kubernetes deployments are partnered with pipelines, a process by which code is moved from source to binaries to containers and, ultimately, into a running cluster. We'll cover pipelines in more detail in *Chapter 18, Provisioning a Multitenant Platform*. For now, the main question is, "How will your pipeline talk to Kubernetes securely?"

If your pipeline runs in the same cluster that is being updated, this is a simple question to answer. You would grant access to the pipeline's service account via RBAC to do what it needs to do. This is why service accounts exist – to provide identities to processes inside the cluster.

What if your pipeline runs outside of the cluster? Kubernetes is an API, and all the options presented in this chapter apply to a pipeline as they would to a user. Legacy service account tokens don't provide an expiration and can easily be abused. The `TokenRequest` API could give you a short-lived token, but you still need to be authenticated to get it. If your cluster runs on the same cloud provider as your pipeline, you may be able to use its integrated IAM system. For instance, you can generate an IAM role in **Amazon CodeBuild** that can talk to an EKS cluster without having a static service account. The same is true for Azure DevOps and AKS.

If a cloud's IAM capabilities won't cover your needs, there are three options. The first is to dynamically generate a token for a pipeline the same way you would for a user, by authenticating to an identity provider and then using the returned `id_token` with your API calls. The second is to generate a certificate that can be used with your API server. Finally, you can leverage impersonation to authenticate your pipeline's token. Let's look at all three options and see how our pipelines can use them.

## Using tokens

Kubernetes doesn't distinguish between an API call from a human or a pipeline. A short-lived token is a great way to interact with your API server as a pipeline, given the risks we have provided throughout this chapter of potentially losing a token. Most of the client SDKs for Kubernetes know how to refresh these tokens. The biggest issue is, how do you get a token your pipeline can use?

Most enterprises already have some kind of service account management system. Here, the term "service account" is generic and means an account used by a service of some kind, instead of being the `ServiceAccount` object in Kubernetes. These service account management systems often have their own way of handling tasks, such as credential rotation and authorization management. They also have their own compliance tools, making it easier to get through your security review processes!

Assuming that you have an enterprise service account for your pipeline, how do you translate that credential into a token? We generate tokens based on credentials in our OIDC integrated identity provider; it would be great to use that from our pipelines too! With OpenUnison, this is pretty easy because the page that gave us our token is just a frontend for an API. The next question to answer is how to authenticate to OpenUnison. We could write some code to simulate a browser and reverse-engineer the login process, but that's just ugly. And if the form changes, our code will break. It would be better to configure the API to authenticate with something that is more API-friendly, such as HTTP Basic authentication.

OpenUnison can be extended by creating configuration custom resources. In fact, most of OpenUnison is configured using these custom resources. The current token service assumes you are authenticating using the default OpenUnison form login mechanism, instead of a basic authentication that would be helpful from a pipeline. In order to tell OpenUnison to support API authentication, we need to tell it to:

- Enable authentication via HTTP Basic authentication by defining an authentication mechanism
- Create an authentication chain that uses the basic authentication mechanism to complete the authentication process
- Define an application that can provide the token API, authenticating using the newly created chain

We won't go through the details of how to make this work in OpenUnison, instead focusing on the end results. The `chapter6` folder contains a Helm chart that was created for you to configure this API. Run it using the same `openunison-values.yaml` file you used to deploy OpenUnison:

```
cd chapter6/pipelines
helm install orchestra-token-api token-login -n openunison -f /tmp/openunison-
values.yaml
NAME: orchestra-token-api
LAST DEPLOYED: Mon Jul 24 18:47:04 2023
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

Once deployed, we can test it using `curl`:

```
$ export KUBE_AZ=$(curl --insecure -u 'pipeline_svc_account:start123' \
https://k8sou.apps.192-168-2-114.nip.io/k8s-api-token/token/user\
| jq -r '.token.id_token')
curl --insecure -H "Authorization: Bearer $KUBE_AZ" https://k8sapi.apps.192-
168-2-114.nip.io/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "172.18.0.2:6443"
    }
  ]
}
```



If you're using direct integration with OpenID Connect, replace `k8sapi.apps.192-168-2-114.nip.io` with `0.0.0.0:6443` to run the `curl` command directly against the API server.

Now, wait a minute or two and try the `curl` command again, and you'll see that you're not authenticated anymore. This example is great if you're running a single command, but most pipelines run multiple steps, and a single token's lifetime isn't enough. We could write code to make use of the `refresh_token`, but most of the SDKs will do that for us. Instead of getting just the `id_token`, let's generate an entire `kubectl` configuration:

```
$ export KUBECONFIG=$(mktemp)
$ kubectl get nodes
The connection to the server localhost:8080 was refused - did you specify the
right host or port?
curl --insecure -u 'pipeline_svc_account:start123' https://k8sou.apps.192-168-
2-114.nip.io/k8s-api-token/token/user 2>/dev/null | jq -r '.token["kubectl
Command"]' | bash
Cluster "impersonation" set.
Context "impersonation" created.
User "pipelinex-95-xsvcx-95-xaccount@impersonation" set.
Switched to context "impersonation".
```

```
$ kubectl get nodes
NAME                  STATUS   ROLES      AGE      VERSION
cluster01-control-plane  Ready    control-plane  130m    v1.27.3
cluster01-worker        Ready    <none>     129m    v1.27.3
```

We're getting a short-lived token securely, while also interacting with the API server using our standard tools! This solution only works if your service accounts are stored and accessed via an LDAP directory. If that's not the case, you can extend OpenUnison's configuration to support any number of configuration options. To learn more, visit OpenUnison's documentation at <https://openunison.github.io/>.

This solution is specific to OpenUnison because there is no standard to convert a user's credentials into an `id_token`. That is a detail left to each identity provider. Your identity provider may have an API to generate an `id_token` easily, but it's more likely you'll need something to act as a broker, since an identity provider won't know how to generate a full `kubectl` configuration.

## Using certificates

The preceding process works well but requires OpenUnison or something similar. If you wanted to take a vendor-neutral approach, you could use certificates as your credential instead of trying to generate a token. Earlier in the chapter, I said that certificate authentication should be avoided for users because of Kubernetes' lack of revocation support and the fact that most certificates aren't deployed correctly. Both of these issues are generally easier to mitigate with pipelines because the deployment can be automated.

If your enterprise requires you to use a central store for service accounts, this approach may not be possible. Another potential issue with this approach is that you may want to use an enterprise CA to generate the certificates for service accounts, but Kubernetes doesn't know how to trust third-party CAs. There are active discussions about enabling the feature, but it's not there yet.

Finally, you can't generate certificates for many managed clusters. Most managed Kubernetes distributions, such as EKS, do not make the private keys needed to sign requests via the built-in API available to clusters directly. In that case, you'll be unable to mint certificates that will be accepted by your cluster.

With all that said, let's walk through the process:

1. First, we'll generate a keypair and certificate signing request (CSR):

```
$ openssl req -out sa_cert.csr \
-new -newkey rsa:2048 -nodes -keyout sa_cert.key \
-subj '/O=k8s/O=sa-cluster-admins/CN=sa-cert/'
Generating a RSA private key
.....+++++
.....+++++
writing new private key to 'sa_cert.key'
-----
```

2. Next, we'll submit the CSR to Kubernetes:

```
$ cat <<EOF | kubectl apply -f -
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: sa-cert
spec:
  request: $(cat sa_cert.csr | base64 | tr -d '\n')
  signerName: kubernetes.io/kube-apiserver-client
  usages:
    - digital signature
    - key encipherment
    - client auth
EOF
```

3. Once the CSR is submitted to Kubernetes, we need to approve the submission:

```
$ kubectl certificate approve sa-cert
certificatesigningrequest.certificates.k8s.io/sa-cert approved
```

4. After being approved, we download the minted certificate into a pem file:

```
$ kubectl get csr sa-cert -o jsonpath='{.status.certificate}' | base64
--decode > sa_cert.crt
```

5. Next, we'll configure kubectl to use our newly approved certificate:

```
$ cp ~/.kube/config ./sa-config
$ export KUBECONFIG=./sa-config
$ kubectl config set-credentials kind-cluster01 --client-key=./sa_cert.
key \
  --client-certificate=./sa_cert.crt
$ kubectl get nodes
Error from server (Forbidden): nodes is forbidden: User "sa-cert" cannot
list resource "nodes" in API group "" at the cluster scope
```

The API server has accepted our certificate but has not authorized it. Our CSR had an o in the subject called `sa-cluster-admins`, which Kubernetes translates to “the user `sa-cert` is in the group `sa-cluster-admins`.” We need to authorize that group to be a cluster admin next:

```
$ export KUBECONFIG=
$ kubectl create -f chapter6/pipelines/sa-cluster-admins.yaml
$ export KUBECONFIG=./sa-config
$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
------	--------	-------	-----	---------

cluster01-control-plane	Ready	control-plane	138m	v1.27.3
cluster01-worker	Ready	<none>	138m	v1.27.3

You now have a key pair that can be used from your pipelines with your cluster! Beware while automating this process. The CSR submitted to the API server can set any groups it wants, including `system:masters`. If a certificate is minted with `system:masters` as an `o` in the subject, it will not only be able to do anything on your cluster; it will also bypass all RBAC authorization. In fact, it will bypass all authorization!

If you're going to go down the certificate route, think about potential alternatives, such as using certificates with your identity provider instead of going directly to the API server. This is similar to our token-based authentication, but instead of using a username and password in HTTP Basic authentication, you use a certificate. This gives you a strong credential that can be issued by your enterprise certificate authority while avoiding having to use passwords.

Next, we'll explore how to authenticate a pipeline using its own identity.

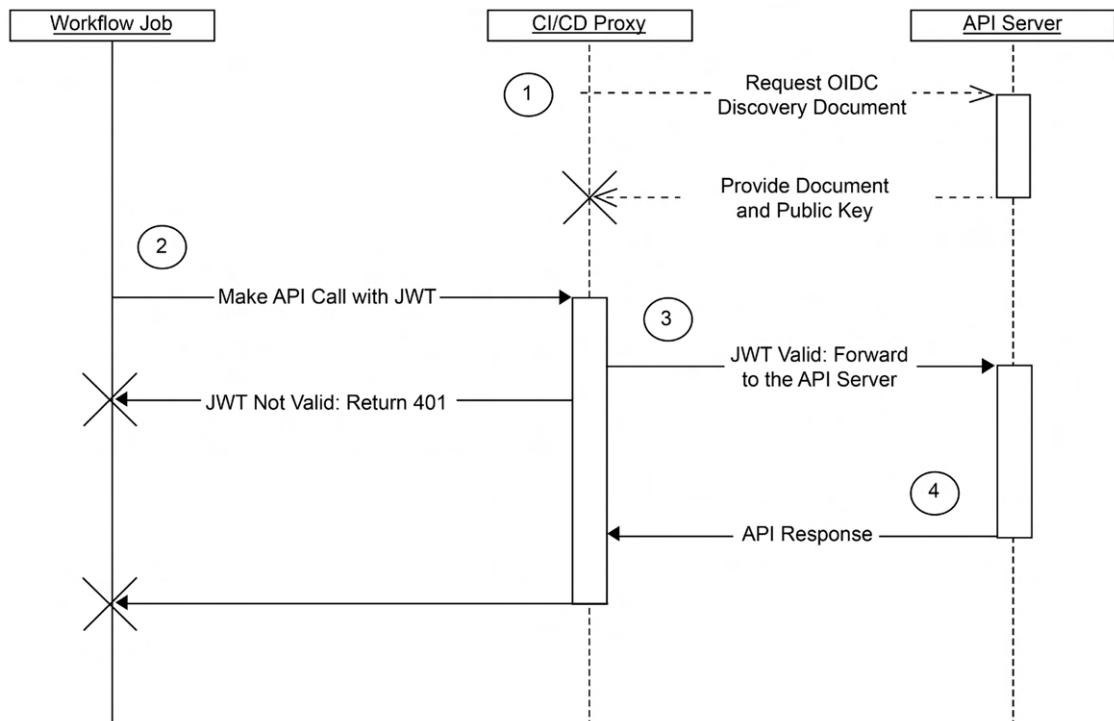
## Using a pipeline's identity

Over the last year or two, discussions about increased supply chain security have become a front-and-center topic for Kubernetes and security professionals. Part of that discussion has led to more pipeline systems providing unique identities to workflows that can be used to interact with remote systems, such as a Kubernetes cluster. This offers the best approach because each workflow is unique, and it can have a short-lived token that doesn't require a shared secret between the Kubernetes cluster and the workflow.

The challenge with using a workflow's identity with a Kubernetes cluster is that a cluster can only accept a single OpenID Connect issuer, and managed clusters aren't even capable of that. Earlier, we explored how your clusters can use impersonation to authenticate API requests to your cluster without enabling OpenID Connect directly in the API server flags. It turns out that this approach works well with CI/CD pipelines too. Instead of configuring your impersonating proxy to trust the identity provider that issues tokens for your users, you can configure it to trust the identity provider that issues tokens for your workflows.

We'll demonstrate this using the CI/CD Proxy (<https://cicd-proxy.github.io>). This is a collection of Helm charts that Tremolo Security built around the `kube-oidc-proxy` project to simplify integration with pipelines. The `kube-oidc-proxy` was created by JetStack, but development ended in early 2021. Tremolo Security forked the project, adding several features, and has since kept it up to date with dependencies and bug fixes as needed. If you ran the lab to deploy authentication with impersonation earlier in this chapter, you've already run Tremolo's `kube-oidc-proxy`. The OpenUnison Helm charts automate its integration for you.

We're going to simulate a workflow deleting some pods in our cluster using the CI/CD Proxy. While we'll be working with GitLab later in the book, that's a very heavy deployment just to show how pipelines can securely authenticate. To simulate our workflow, we're going to run a simple Job that will have a token mounted via the TokenRequest API, with our CI/CD proxy as the audience, instead of the API server. Our CI/CD proxy will then impersonate the ServiceAccount in the projected token's sub-claim, which will be allowed to delete pods in our namespace. The CI/CD proxy will be configured to trust our cluster's OIDC discovery URL, completing the circle of trust. Let's walk through how these trusts come together.



*Figure 6.10: Workflow authentication sequence*

The above diagram shows the following sequence of events:

1. When the CI/CD proxy starts, it reaches out to the cluster's OIDC discovery document to pull in the correct keys to validate inbound tokens. Since we're trusting our own cluster's tokens, we're using <https://kubernetes.default.svc.cluster.local/> as our issuer, so we'll pull in <https://kubernetes.default.svc.cluster.local/.well-known/openid-configuration>.

- When our workflow Job starts, it will have a token projected into it that will have our CI/CD proxy as an audience. This is in addition to the ServiceAccount token that every pod is provided by default. If we inspect this token, we'll see how it differs from the standard token.

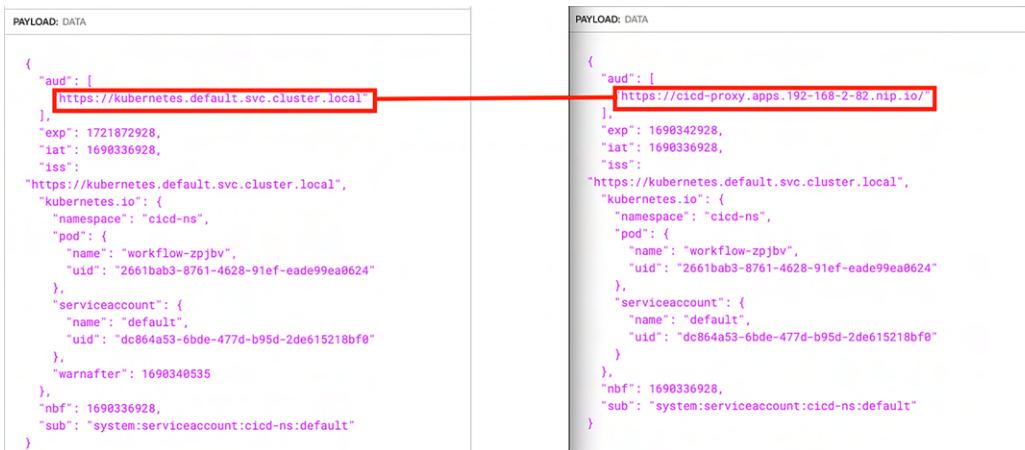


Figure 6.11: Comparing tokens

The above image is a side-by-side comparison of the typical ServiceAccount token on the left and a token meant for the CI/CD proxy on the right. Both are bound to the specific pod, but the left-hand side is meant for the API server where whereas the right-hand token is meant for our proxy. They can't be used interchangeably, even though they're signed by the same set of keys and have the same issuer. If you try to use the token on the right with the API server, it will be rejected for having an invalid audience. The same would be true if you tried to use the token on the left with our proxy. The other major difference between the two tokens is that the expiration of the token on the right is only 10 minutes after creation. This means that if an attacker were to get access to this token, they'd only have 10 minutes to use it, increasing the security of the token.

- Once our Job makes a call using `kubectl` to our proxy, not directly to an API server, the proxy checks the token to make sure it was signed correctly and built correctly. The proxy then forwards the request to the API server but with its own token and the addition of the impersonation headers.
- Finally, the API server acts on the request as if it were made by our Job.

Throughout this transaction, there are no shared secrets that need to be distributed or rotated. There's very little that can be compromised. Since the OIDC discovery document is controlled by our identity provider, if the keys need to be rotated, our proxy will pick it up. Having walked through the theory, let's deploy our example.

First, start with a fresh cluster:

```

cd Kubernetes-An-Enterprise-Guide-Third-Edition/chapter2
kind delete cluster -n cluster01
./create-cluster.sh

```

Once the cluster is created, let's deploy the CI/CD proxy. We didn't want to get bogged down with specific steps, so we automated the deployment:

```
cd ../chapter6/pipelines/cicd-proxy  
./deploy-proxy.sh
```

This script will take a minute or two to run. It does a few things:

1. Deploys the cert-manager project from JetStack and creates an internal CA that we'll use to sign certificates
2. Enables anonymous access to the API server's OIDC discovery document
3. Deploys the CI/CD proxy using Tremolo Security's Helm charts
4. Creates a target namespace and Deployment we can use to test deleting pods in
5. Creates an RBAC binding for our impersonated user to be able to list and delete pods in our target namespace

Once everything is deployed, the next step is to create our Job and check the logs:

```
./run_workflow.sh  
  
kubectl logs -l job-name=workflow -n cicd-ns  
User "remote" set.  
Context "remote" created.  
Switched to context "remote".  
pod "test-pods-777b69dc55-4bmwd" deleted
```

We can see that we were able to delete our pods, using the projected token!

This seems like quite a bit of work just to delete a Pod. It might have been easier to just create a ServiceAccount token and store it somewhere our workflow could access it. However, that would be a security and Kubernetes anti-pattern. It means that so long as the pod exists in the cluster's etcd database, it could be used without restriction. You could create a rotation system, but just like with custom authentication, you're now creating a pale imitation of OpenID Connect's existing security. You're also building out additional automation that also needs to be secured. So what looks like quite a bit of additional work will actually save you time and make your security team happy!

Having discussed how to properly authenticate to your cluster from your pipeline, let's examine some anti-patterns with pipeline authentication.

## Avoiding anti-patterns

It turns out most of the anti-patterns that apply to user authentication also apply to pipeline authentication. Given the nature of code which authenticates, there are some specific things to look out for.

First, don't use a person's account for a pipeline. It will likely violate your enterprise's policies and can expose your account, and maybe your employer, to issues. Your enterprise account (which is assigned to everyone else in the enterprise) generally has several rules attached to it. Simply using it in code can breach these rules. The other anti-patterns we'll discuss add to the risk.

Next, never put your service account's credentials into Git, even when encrypted. It's popular to include credentials directly in objects stored in Git because you now have change control, but it's just so easy to accidentally push a Git repository out to a public space. Much of security is about protecting users from accidents that can leak sensitive information. Even encrypted credentials in Git can be abused if the encryption keys are also stored in Git. Every cloud provider has a secret management system that will synchronize your credentials into Kubernetes Secret objects. You can do this with Vault as well, which we'll do later in this book. This is a much better approach, as these tools are specifically designed to manage sensitive data. Git is meant to make it easy to share and collaborate, which makes for poor secret management.

Finally, don't use legacy service account tokens from outside of your cluster. I know that I've said this a dozen times in this chapter, but it's incredibly important. When using a bearer token, anything that carries that token is a potential attack vector. There have been network providers that leak tokens, for example. It's a common anti-pattern. If a vendor tells you to generate a service account token, push back – you're putting your enterprise's data at risk.

## Summary

This chapter detailed how Kubernetes identifies users and what groups their members are in. We detailed how the API server interacts with identities and explored several options for authentication. Finally, we detailed the OpenID Connect protocol and how it's applied to Kubernetes.

Learning how Kubernetes authenticates users and the details of the OpenID Connect protocol is an important part of building security in a cluster. Understanding the details and how they apply to common enterprise requirements will help you decide the best way to authenticate to clusters, and also provide justification regarding why the anti-patterns we explored should be avoided.

In the next chapter, we'll apply our authentication process to authorizing access to Kubernetes resources. Knowing who somebody is isn't enough to secure your clusters. You also need to control what they have access to.

## Questions

1. OpenID Connect is a standard protocol with extensive peer review and usage.
  - a. True
  - b. False
2. Which token does Kubernetes use to authorize your access to an API?
  - a. `access_token`
  - b. `id_token`
  - c. `refresh_token`
  - d. `certificate_token`

3. In which situation is certificate authentication a good idea?
  - a. Day-to-day usage by administrators and developers
  - b. Access from external CI/CD pipelines and other services
  - c. Break glass in case of emergency, when all other authentication solutions are unavailable
4. How should you identify users accessing your cluster?
  - a. Email address
  - b. Unix login ID
  - c. Windows login ID
  - d. An immutable ID not based on a user's name
5. Where are OpenID Connect configuration options set in Kubernetes?
  - a. Depends on the distribution
  - b. In a ConfigMap object
  - c. In a secret
  - d. Set as flags on the Kubernetes API server executable
6. When using Impersonation with your cluster, the groups your user brings are the only ones needed.
  - a. True
  - b. False
7. The dashboard should have its own privileged identity to work properly.
  - a. True
  - b. False

## Answers

1. a: True
2. b: id\_token
3. c: Break glass in case of emergency, when all other authentication solutions are unavailable
4. d: An immutable ID not based on a user's name
5. d: Set as flags on the Kubernetes API server executable
6. b: False
7. b: False



# 7

## RBAC Policies and Auditing

Authentication is only the first step in managing access to a cluster. Once access to a cluster is granted, it's important to limit what accounts can do, depending on whether an account is for an automated system or a user. Authorizing access to resources is an important part of protecting against both accidental issues and bad actors looking to abuse a cluster.

In this chapter, we're going to detail how Kubernetes authorizes access via its **Role-Based Access Control (RBAC)** model. The first part of this chapter will be a deep dive into how Kubernetes RBAC is configured, what options are available, and mapping the theory onto practical examples. Debugging and troubleshooting RBAC policies will be the focus of the second half.

In this chapter, we will cover the following topics:

- Introduction to RBAC
- Mapping enterprise identities to Kubernetes to authorize access to resources
- Implementing namespace multi-tenancy
- Kubernetes auditing
- Using `audit2rbac` to debug policies

Once you have completed this chapter, you'll have the tools needed to manage access to your cluster via Kubernetes' integrated RBAC model and debug issues when they arise. Next, let's dive into the technical requirements for this chapter.

### Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, though 8 GB is suggested.
- Scripts from the `chapter7` folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Introduction to RBAC

RBAC stands for Role-Based Access Control. At its core is the idea of building permission sets, which are called **Roles**, and lists of subjects (users) that those permissions apply to. In this chapter, we'll walk through building roles and their corresponding bindings to build out the permissions in our clusters.

### What's a Role?

In Kubernetes, a **Role** is a way to tie together permissions into an object standardized to a specific schema. By codifying **Roles** into this schema, you're able to standardize and automate their creation and management.

Roles have rules, which are a collection of resources and verbs. Working backward, we have the following:

- **Verbs:** The actions that can be taken on an API, such as reading (get), writing (create, update, patch, and delete), or listing and watching.
- **Resources:** Names of APIs to apply the verbs to, such as services, endpoints, and so on. Specific sub-resources, such as logs and status, may be listed as well. Specific resources can be named to provide very specific permissions on an object.

A Role does not say who can perform the verbs on the resources—that is handled by `RoleBindings` and `ClusterRoleBindings`. We will learn more about these in the *RoleBindings and ClusterRoleBindings* section.

The term “role” can have multiple meanings, and RBAC is often used in other contexts. In the enterprise world, the term “role” is often associated with a business role and used to convey entitlements to that role instead of a specific person.

### Role-Based Access Control

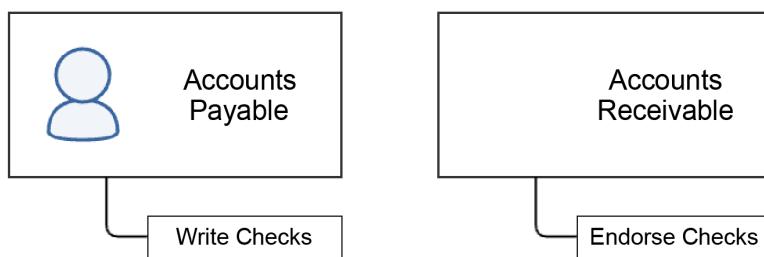


Figure 7.1: RBAC versus entitlement-based access control

As an example, in *Figure 7.1*, a user is a member of the “role” for accounts payable. Being a member of this role automatically provides the entitlement to “write checks.” If our user’s job changes to accounts receivable, their permissions will change automatically because the permissions are tied to the user’s role. In enterprise RBAC models, what ties a user to a “role” is generally some context instead of a specific group membership or attribute value. For instance, users might be located in different parts of the corporate directory based on their “role.”

This is different from how Kubernetes uses the term “Role” to mean a list of permissions, and those permissions aren’t tied together because of a business role but because of a technical requirement. As we’ll see when we get to bindings, Kubernetes Roles are tightly bound to accounts and groups, and while the permissions of a Role are grouped together for a specific function, that function is defined at a lower, technical level than “enterprise” roles.

Now that we’ve differentiated between what the “enterprise” definition of a role is from how Kubernetes defines a Role, let’s dive into how you build a Role.

Each resource that a role will be built from is identified by the following:

- `apiGroups`: A list of groups the resources are a member of
- `resources`: The name of the object type for the resource (and potentially sub-resources)
- `resourceNames`: An optional list of specific objects to apply this rule to

Each rule *must* have a list of `apiGroups` and `resources`. `resourceNames` is optional.

Once the resource is identified in a rule, verbs can be specified. A verb is an action that can be taken on the resource, providing access to the object in Kubernetes.

If the desired access to an object should be all, you do not need to add each verb; instead, the wildcard character may be used to identify all the verbs, resources, or `apiGroups`.

## Identifying a Role

The Kubernetes authorization page (<https://kubernetes.io/docs/reference/access-authn-authz/rbac/>) uses the following Role as an example to allow someone to get the details of a pod and its logs:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-and-pod-logs-reader
rules:
  - apiGroups: [""]
    resources: ["pods", "pods/log"]
    verbs: ["get", "list"]
```

Before defining what a Role manages, it’s important to note that Role objects are namespaced, so the `namespace` the Role is created in means that the permissions it defines apply only within its own namespace. In this example, the Role only applies to the `default` namespace.

Working backward to determine how this Role was defined, we will start with `resources`, since it is the easiest aspect to find. All objects in Kubernetes are represented by URLs. If you wanted to pull all the information about the pods in the `default` namespace, you would call the `/api/v1/namespaces/default/pods` URL, and if you wanted the logs for a specific pod, you would call the `/api/v1/namespaces/default/pods/mypod/log` URL.

The URL pattern will be true of all namespace-scoped objects. `pods` lines up to resources, as does `pods/log`. When trying to identify which resources you want to authorize, use the `api-reference` document from the Kubernetes API documentation at <https://kubernetes.io/docs/reference/#api-reference>.

If you are trying to access an additional path component after the name of the object (such as with `status` and `logs` on `pods`), it needs to be explicitly authorized. Authorizing `Pods` does not immediately authorize `logs` or `status`.

Based on the use of URL mapping to resources, your next thought may be that the `verbs` field is going to be HTTP verbs. This is not the case. There is no `GET` verb in Kubernetes. Verbs are instead defined by the schema of the object in the API server. The good news is that there's a static mapping between HTTP verbs and RBAC verbs (<https://kubernetes.io/docs/reference/access-authn-authz/authorization/#determine-the-request-verb>). Looking at this URL, notice that there are verbs on top of the HTTP verbs for **impersonation**. That's because the RBAC model is used beyond authorizing specific APIs and is also used to authorize who can impersonate users. The focus of this chapter is going to be on the standard HTTP verb mappings.

The final component to identify is `apiGroups`. APIs will be in an API group and that group will be part of their URL. You can find the group by looking at the API documentation for the object you are looking to authorize or by using the `kubectl api-resources` command. For instance, to get `apiGroups` for the `Ingress` object, you could run:

```
kubectl api-resources -o wide | grep Ingress
ingressclasses networking.k8s.io/v1 false IngressClass [create delete delete-
collection get list patch update watch]
ingresses ing networking.k8s.io/v1 true Ingress [create delete deletecollection
get list patch update watch]
```

The second result gives you what you would see in the `apiVersion` of a YAML version of an `Ingress` object. Use this for `apiGroups`, but without the version number. To apply a `Role` to an `Ingress` object, the `apiGroups` would be `networking.k8s.io`.

The inconsistencies in the RBAC model can make debugging difficult, to say the least. The last lab in this chapter will walk through the debugging process and take much of the guesswork out of defining your rules.

Now that we've defined the contents of a `Role` and how to define specific permissions, it's important to note that `Roles` can be applied at both the namespace and cluster level.

## Roles versus ClusterRoles

RBAC rules can be scoped either to specific namespaces or to the entire cluster. Taking the preceding example, if we defined it as a `ClusterRole` instead of a `Role`, and removed the namespace, we would have a `Role` that authorizes someone to get the details and logs of all pods across the cluster. This new `Role` could alternatively be used in individual namespaces to assign the permissions to the pods in a specific namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-pod-and-pod-logs-reader
rules:
- apiGroups: [""]
  resources: ["pods", "pods/log"]
  verbs: ["get", "list"]
```

Whether this permission is applied globally across a cluster or within the scope of a specific namespace depends on how it's bound to the subjects it applies to. This will be covered in the *RoleBindings* and *ClusterRoleBindings* section.

In addition to applying a set of rules across the cluster, *ClusterRoles* are used to apply rules to resources that aren't mapped to a namespace, such as *PersistentVolume* and *StorageClass* objects.

After learning how a *Role* is defined, let's explore the different ways *Roles* can be designed for specific purposes. In the next sections, we'll look at different patterns for defining *Roles* and their application in a cluster.

## Negative Roles

One of the most common requests for authorization is “*Can I write a Role that lets me do everything EXCEPT xyz?*”. In RBAC, the answer is NO. RBAC requires either every resource to be allowed or specific resources and verbs to be enumerated. There are two reasons for this in RBAC:

- **Better security through simplicity:** Being able to enforce a rule that says *every Secret except this one* requires a much more complex evaluation engine than RBAC provides. The more complex an engine, the harder it is to test and validate, and the easier it is to break. A simpler engine is just simpler to code and keep secure.
- **Unintended consequences:** Allowing someone to do everything *except xyz* leaves the door open for issues in unintended ways as the cluster grows and new capabilities are added.

On the first point, an engine with this capability is difficult to build and maintain. It also makes the rules much harder to keep track of. To express this type of rule, you need to not only have authorization rules but also an order to those rules. For instance, to say *I want to allow everything except this Secret*, you would first need a rule that says *allow everything* and then a rule that says *deny this secret*. If you switch the rules to say *deny this secret* then *allow everything*, the first rule would be overridden. You could assign priorities to different rules, but that now makes it even more complex.

There are ways to implement this pattern, either by using a custom authorization webhook or by using a controller to dynamically generate RBAC *Role* objects. These should both be considered security anti-patterns and so won't be covered in this chapter.

The second point deals with unintended consequences. It's becoming more popular to support the provisioning of infrastructure that isn't Kubernetes using the operator pattern, where a custom controller looks for new instances of a **CustomResourceDefinition (CRD)** to provision infrastructure such as databases.

Amazon Web Services publishes an operator for this purpose (<https://github.com/aws/aws-controllers-k8s>). These operators run in their own namespaces with administrative credentials for their cloud looking for new instances of their objects to provision resources. If you have a security model that allows everything “except...”, then once deployed, anyone in your cluster can provision cloud resources that have real costs and can create security holes. Enumerating your resources, from a security perspective, is an important part of knowing what is running and who has access.

The trend in Kubernetes clusters is to provide more control over infrastructure outside of the cluster via the custom resource API. You can provision anything from VMs to additional nodes, to any kind of API-driven cloud infrastructure. There are other tools you can use besides RBAC to mitigate the risk of someone creating a resource they shouldn’t, but these should be secondary measures.

So far, we’ve looked at how to create permissions for specific use cases. What happens if you need some flexibility to be able to define permissions more dynamically than the static lists we’re providing now? Next, we’ll discover how to use aggregated ClusterRoles to provide a dynamic approach to permission lists.

## Aggregated ClusterRoles

ClusterRoles can become confusing quickly and be difficult to maintain. It’s best to break them up into smaller ClusterRoles that can be combined as needed. Take the `admin` ClusterRole, which is designed to let someone do generally anything inside of a specific namespace. When we look at the `admin` ClusterRole, it enumerates just about every resource there is. You may think someone wrote this ClusterRole so that it would contain all those resources, but that would be really inefficient, and what happens as new resource types get added to Kubernetes? The `admin` ClusterRole is an aggregated ClusterRole. Take a look at the ClusterRole:

```
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: admin
  labels:
    kubernetes.io/bootstrapping: rbac-defaults
  annotations:
    rbac.authorization.kubernetes.io/autoupdate: 'true'
rules:
  .
  .
  .
aggregationRule:
  clusterRoleSelectors:
    - matchLabels:
        rbac.authorization.k8s.io/aggregate-to-admin: 'true'
```

The key is the `aggregationRule` section. This section tells Kubernetes to combine the rules for all `ClusterRoles` where the `rbac.authorization.k8s.io/aggregate-to-admin` label is true. When a new CRD is created, an admin is not able to create instances of that CRD without adding a new `ClusterRole` that includes this label. To allow namespace admin users to create an instance of the new `myapi/superwidget` objects, create a new `ClusterRole`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: aggregate-superwidget-admin
  labels:
    # Add these permissions to the "admin" default role.
    rbac.authorization.k8s.io/aggregate-to-admin: "true"
rules:
- apiGroups: ["myapi"]
  resources: ["superwidgets"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

The next time you look at the `admin` `ClusterRole`, it will include `myapi/superwidgets`. You can also reference this `ClusterRole` directly for more specific permissions.

So far, we've focussed on creating permission lists via `Roles` and `ClusterRoles`. Next, we'll work on assigning those permissions to users and services.

## RoleBindings and ClusterRoleBindings

Once a permission is defined, it needs to be assigned to something to enable it. “Something” can be a user, a group, or a service account. These options are referred to as `subjects`. Just as with `Roles` and `ClusterRoles`, a `RoleBinding` binds a `Role` or `ClusterRole` to a specific namespace, and a `ClusterRoleBinding` will apply a `ClusterRole` across the cluster. A binding can have many subjects but may only reference a single `Role` or `ClusterRole`. To assign the `pod-and-pod-logs-reader` `Role` created earlier in this chapter to a service account called `mysa` in the default namespace, a user named `podreader`, or anyone with the `podreaders` group, create a `RoleBinding`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-and-pod-logs-reader
  namespace: default
subjects:
- kind: ServiceAccount
  name: mysa
  namespace: default
  apiGroup: rbac.authorization.k8s.io
- kind: User
```

```

  name: podreader
- kind: Group
  name: podreaders
roleRef:
  kind: Role
  name: pod-and-pod-logs-reader
  apiGroup: rbac.authorization.k8s.io

```

The preceding RoleBinding lists three different subjects:

- **ServiceAccount:** Any service account in the cluster can be authorized to a RoleBinding. The namespace must be included since a RoleBinding can authorize a service account in any namespace, not just the one the RoleBinding is defined in.
- **User:** A user is asserted by the authentication process. Remember from *Chapter 6, Integrating Authentication into Your Cluster*, that there are no objects in Kubernetes that represent users.
- **Group:** Just as with users, groups are asserted as part of the authentication process and have no object associated with them.

Finally, the Role we created earlier is referenced. In a similar fashion, to assign the same subjects the ability to read pods and their logs across the cluster, a ClusterRoleBinding can be created to reference the `cluster-pod-and-pod-logs-reader` ClusterRole created earlier in the chapter:

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-pod-and-pod-logs-reader
subjects:
- kind: ServiceAccount
  name: mysa
  namespace: default
  apiGroup: rbac.authorization.k8s.io
- kind: User
  name: podreader
- kind: Group
  name: podreaders
roleRef:
  kind: ClusterRole
  name: cluster-pod-and-pod-logs-reader
  apiGroup: rbac.authorization.k8s.io

```

The ClusterRoleBinding is bound to the same subjects but is bound to a ClusterRole instead of a namespace-bound Role. Now, instead of being able to read pod details and Pod/logs in the default namespace, these users can read all pod details and Pod/logs in all namespaces.

Till now, the focus has been on combining a `Role` with a `RoleBinding` and a `ClusterRole` with a `ClusterRoleBinding`. If you want to define the same permissions that are scoped to multiple namespaces, you'll want a way to do that without reproducing the same `Role` every time. Next, we'll cover how to simplify `Role` management by combining `ClusterRoles` and `RoleBindings`.

## Combining ClusterRoles and RoleBindings

We have a use case where a log aggregator wants to pull logs from pod in multiple namespaces, but not all namespaces. A `ClusterRoleBinding` is too broad. While the `Role` could be recreated in each namespace, this is inefficient and a maintenance headache. Instead, define a `ClusterRole` but reference it from a `RoleBinding` in the applicable namespaces. This allows the reuse of permission definitions while still applying those permissions to specific namespaces. In general, note the following:

- `ClusterRole + ClusterRoleBinding` = cluster-wide permission
- `ClusterRole + RoleBinding` = namespace-specific permission

To apply our `ClusterRoleBinding` in a specific namespace, create a `Role`, referencing the `ClusterRole` instead of a namespaced `Role` object:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-and-pod-logs-reader
  namespace: default
subjects:
- kind: ServiceAccount
  name: mysa
  namespace: default
  apiGroup: rbac.authorization.k8s.io
- kind: User
  name: podreader
- kind: Group
  name: podreaders
roleRef:
  kind: ClusterRole
  name: cluster-pod-and-pod-logs-reader
  apiGroup: rbac.authorization.k8s.io
```

The preceding `RoleBinding` lets us reuse the existing `ClusterRole`. This cuts down on the number of objects that need to be tracked in the cluster and makes it easier to update permissions across the cluster if the `ClusterRole` permissions need to change.

Having built our permissions and defined how to assign them, next, we'll look at how to map enterprise identities into cluster policies.

## Mapping enterprise identities to Kubernetes to authorize access to resources

One of the benefits of centralizing authentication is leveraging the enterprise's existing identities instead of having to create new credentials that users who interact with your clusters need to remember. It's important to know how to map your policies to these centralized users. In *Chapter 6, Integrating Authentication into Your Cluster*, you created a cluster and integrated it with an "enterprise Active Directory." To finish the integration, the following `ClusterRoleBinding` was created:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: ou-cluster-admins
subjects:
- kind: Group
  name: cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

This binding allows all users that are members of the `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com` group to have full cluster access. At the time, the focus was on authentication, so there weren't many details provided as to why this binding was created.

What if we wanted to authorize our users directly? That way, we will have control over who has access to our cluster. Our RBAC `ClusterRoleBinding` would look different:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: ou-cluster-admins
subjects:
- kind: User
  name: https://k8sou.apps.192-168-2-131.nip.io/auth/idp/k8sIdp#mmosley
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
```

Using the same `ClusterRole` as before, this `ClusterRoleBinding` will assign the `cluster-admin` privileges only to my testing user.

The first issue to point out is that the user has the URL of our **OpenID Connect** issuer in front of the username. When OpenID Connect was first introduced, it was thought that Kubernetes would integrate with multiple identity providers and different types of identity providers, so the developers wanted you to be able to easily distinguish between users from different identity sources. For instance, `mmosley` in domain 1 is a different user than `mmosley` in domain 2. To ensure that a user's identity doesn't collide with another user across identity providers, Kubernetes requires the identity provider's issuer to be prepended to your username. This rule doesn't apply if the username claim defined in your API server flags is `mail`. It also doesn't apply if you're using certificates or impersonation.

Beyond the inconsistent implementation requirements, this approach can cause problems in a few ways:

- **Changing your identity provider URL:** Today, you're using an identity provider at one URL, but tomorrow you decide to move it. Now, you need to go through every `ClusterRoleBinding` and update them.
- **Audits:** You can't query for all `RoleBindings` associated with a user. You need to instead enumerate every binding.
- **Large bindings:** Depending on how many users you have, your bindings can become quite large and difficult to track.

While there are tools you can use to help manage these issues, it's much easier to associate your bindings with groups instead of individual users. You could use the `mail` attribute to avoid the URL prefix, but that is considered an anti-pattern and will result in equally difficult changes to your cluster if an email address changes for any reason.

So far in this chapter, we have learned how to define access policies and map those policies to enterprise users. Next, we need to determine how clusters will be divided into tenants.

## Implementing namespace multi-tenancy

Clusters deployed for multiple stakeholders, or tenants, should be divided up by namespace. This is the boundary that was designed in Kubernetes from the very beginning. When deploying namespaces, there are generally two `ClusterRoles` that are assigned to users in the namespace:

- `admin`: This aggregated `ClusterRole` provides access to every verb and nearly every resource that ships with Kubernetes, making the `admin` user the ruler of their namespace. The exception to this is any namespace-scoped object that could affect the entire cluster, such as `ResourceQuotas`.
- `edit`: Similar to `admin`, but without the ability to create RBAC Roles or `RoleBindings`.

It's important to note that the `admin` `ClusterRole` can't make changes to the namespace object by itself. Namespaces are cluster-wide resources, so they can only be assigned permissions via a `ClusterRoleBinding`.

Depending on your strategy for multi-tenancy, the `admin ClusterRole` may not be appropriate. The ability to generate RBAC Role and RoleBinding objects means that a namespace admin may grant themselves the ability to change resource quotas. This is where RBAC tends to fall apart and needs some additional options:

- **Don't grant access to Kubernetes:** Many cluster owners want to keep Kubernetes out of the hands of their users and limit their interaction with external CI/CD tools. This works well with microservices but begins to fall apart on multiple lines. First, more legacy applications being moved into Kubernetes means more legacy administrators needing to directly access their namespace. Second, if the Kubernetes team keeps users out of the clusters, they are now responsible. The people who own Kubernetes may not want to be the reason things aren't happening the way application owners want them to and, often, the application owners want to be able to control their own infrastructure to ensure they can handle any situation that impacts their own performance.
- **Treat access as privileged:** Most enterprises require a privileged user to access infrastructure. This is typically done using a privileged access model where an admin has a separate account that needs to be "checked out" in order to use it and is only authorized at certain times, as approved by a "change board" or process. The use of these accounts is closely monitored. This is a good approach if you already have a system in place, especially one that integrates with your enterprise's central authentication system.
- **Give each tenant a cluster:** This model moves multi-tenancy from the cluster to the infrastructure layer. You haven't eliminated the problem, only moved where it is addressed. This can lead to sprawl that becomes unmanageable and costs can skyrocket depending on how you are implementing Kubernetes. In *Chapter 9, Building Multitenant Clusters with vClusters*, we'll explore how we can give each tenant its own cluster without having to worry about the sprawl as much.
- **Admission controllers:** These augment RBAC by limiting which objects can be created. For instance, an admission controller can decide to block an RBAC policy from being created, even if RBAC explicitly allows it. This topic will be covered in *Chapter 11, Extending Security Using Open Policy Agent*.

In addition to authorizing access to namespaces and resources, a multi-tenant solution needs to know how to provision tenants. This topic will be covered in the final chapters – *Chapter 18, Provisioning a Multitenant Platform*, and *Chapter 19, Building a Developer Portal*.

Now that we have a strategy for implementing authorization policies, we'll need a way to debug those policies as we create them and also to know when those policies are violated. Kubernetes provides an audit capability that will be the focus of the next section, where we will add the audit log to our Kind cluster and debug the implementation of RBAC policies.

## Kubernetes auditing

The Kubernetes audit log is where you track what is happening in your cluster from an API perspective. It's in JSON format, which makes reading it directly more difficult, but makes it much easier to parse using tools such as OpenSearch. In *Chapter 15, Managing Clusters and Workloads*, we will cover how to create a full logging system using the OpenSearch stack.

## Creating an audit policy

A policy file is used to control what events are recorded and where to store the logs, which can be a standard log file or a webhook. We have included an example audit policy in the `chapter7` directory of the GitHub repository, and we will apply it to the KinD cluster that we have been using throughout the book.

An **audit policy** is a collection of rules that tell the API server which API calls to log and how. When Kubernetes parses the policy file, all rules are applied in order and only the initial matching policy event will be applied. If you have more than one rule for a certain event, you may not receive the expected data in your log files. For this reason, you need to be careful that your events are created correctly.

Policies use the `audit.k8s.io` API and the manifest kind of `Policy`. The following example shows the beginning of a policy file:

```
apiVersion: audit.k8s.io/v1
kind: Policy
rules:
  - level: Request
    userGroups: ["system:nodes"]
    verbs: ["update", "patch"]
    resources:
      - group: "" # core
        resources: ["nodes/status", "pods/status"]
    omitStages:
      - "RequestReceived"
```

While a policy file may look like a standard Kubernetes manifest, you do not apply it using `kubectl`. A policy file is used with the `--audit-policy-file` API flag on the API server(s). This will be explained in the *Enabling auditing on a cluster* section.

To understand the rule and what it will log, we will go through each section in detail.

The first section of the rule is `level`, which determines the type of information that will be logged for the event. There are four levels that can be assigned to events:

Audit level	Logging details
None	Does not log any data
Metadata	Only logs metadata – does not include the request or the request response
Request	Logs metadata and the request, but not the request response
RequestResponse	Logs metadata, the request, and the request response

Table 7.1: Kubernetes auditing levels

The `userGroups`, `verbs`, and `resources` values tell the API server the object and action that will trigger the auditing event. In this example, only requests from `system:nodes` that attempt an action of `update` or `patch` on a `node/status` or `pod/status` on the core API will create an event.

`omitStages` tells the API server to skip any logging events during a stage, which helps you limit the amount of data that is logged. There are four stages that an API request goes through:

API stage	Stage details
<code>RequestReceived</code>	This is the stage where the API receives a request.
<code>ResponseStarted</code>	This stage is only used with certain requests and it starts before the response is sent in the <code>ResponseComplete</code> stage.
<code>ResponseComplete</code>	This is the stage where the API server responds to a request.
<code>Panic</code>	Event created if a panic occurs.

Table 7.2: Auditing stages

In our example, we have set the event to ignore the `RequestReceived` event, which tells the API server not to log any data for the incoming API request.

Every organization has its own auditing policy, and policy files can become long and complex. Don't be afraid to set up a policy that logs everything until you get a handle on the types of events that you can create. Logging everything is not a good practice since the log files become very large. Even when pushing logs into an external system, like `OpenSearch`, there's still a cost in processing and management. Fine-tuning an audit policy is a skill that is learned over time and as you learn more about the API server, you will start to learn what events are most valuable to audit.

Policy files are just the start of enabling cluster auditing, and now that we have an understanding of the policy file, let's explain how to enable auditing on a cluster.

## Enabling auditing on a cluster

Enabling auditing is specific to each distribution of Kubernetes. In this section, we will enable the audit log in KinD to understand the low-level steps. As a quick refresher, the finished product of the last chapter was a KinD cluster with impersonation enabled (instead of directly integrating with OpenID Connect). The rest of the steps and examples in this chapter assume this cluster is being used. Start with a fresh cluster and deploy `OpenUnison` with impersonation from *Chapter 6*:

```
cd Kubernetes-An-Enterprise-Guide-Third-Edition/chapter2
kind delete cluster -n cluster01
./create-cluster.sh
cd ../chapter6/user-auth
./deploy_openunison_imp_imersonation.sh
```

Next, we're going to configure the API server to send audit log data to a file. This is more complex than setting a switch because kubeadm, the installer that KinD is built on, runs the API server as a static pod(s). The API server is a container inside of Kubernetes! This means that in order for us to tell the API server where to write log data to, we first have to have storage to write it to and then configure the API server's pod to use that location as a volume. We're going to walk through this process manually to give you experience with modifying the API server's context.

You can follow the steps in this section manually or you can execute the included script, `enable-auditing.sh`, in the `chapter7` directory of the GitHub repository:

1. First, copy the example audit policy from the `chapter7` directory to the API server:

```
$ cd chapter7
$ docker exec -ti cluster01-control-plane mkdir /etc/kubernetes/audit
$ docker cp cm/k8s-audit-policy.yaml cluster01-control-plane:/etc/kubernetes/audit/
```

2. Next, create the directories to store the audit log and policy configuration on the API server. We will exec into the container since we need to modify the API server file in the next step:

```
$ docker exec -ti cluster01-control-plane mkdir /var/log/k8s
```

At this point, you have the audit policy on the API server and you can enable the API options to use the file.

3. On the API server, edit the `kubeadm` configuration file (you will need to install an editor such as vi by running `apt-get update; apt-get install vim`), `/etc/kubernetes/manifests/kube-apiserver.yaml`, which is the same file that we updated to enable OpenID Connect. To enable auditing, we need to add three values.

It's important to note that many Kubernetes clusters may only require the file and the API options. We need the second and third steps since we are using a KinD cluster for our testing.

First, add bold command-line flags for the API server that enable the audit logs. Along with the policy file, we can add options to control the log file rotation, retention, and maximum size:

```
- --tls-private-key-file=/etc/kubernetes/pki/apiserver.key
- --audit-log-path=/var/log/k8s/audit.log
- --audit-log-maxage=1
- --audit-log-maxbackup=10
- --audit-log-maxsize=10
- --audit-policy-file=/etc/kubernetes/audit/k8s-audit-policy.yaml
```

Notice that the option is pointing to the policy file that you copied over in the previous step.

4. Next, add the bold directories that store the policy configuration and the resulting logs to the `volumeMounts` section:

```

- mountPath: /usr/share/ca-certificates
  name: usr-share-ca-certificates
  readOnly: true
- mountPath: /var/log/k8s
  name: var-log-k8s
  readOnly: false
- mountPath: /etc/kubernetes/audit
  name: etc-kubernetes-audit
  readOnly: true

```

5. Finally, add the bold `hostPath` configurations to the `volumes` section so that Kubernetes knows where to mount the local paths:

```

- hostPath:
    path: /usr/share/ca-certificates
    type: DirectoryOrCreate
  name: usr-share-ca-certificates
- hostPath:
    path: /var/log/k8s
    type: DirectoryOrCreate
  name: var-log-k8s
- hostPath:
    path: /etc/kubernetes/audit
    type: DirectoryOrCreate
  name: etc-kubernetes-audit

```

6. Save and exit the file.
7. Like all API option changes, you need to restart the API server for the changes to take effect; however, KinD will detect that the file has changed and restart the API server's pod automatically.

Exit the attached shell and check the pods in the `kube-system` namespace:

```
$ kubectl get pod kube-apiserver-cluster01-control-plane -n kube-system
NAME                           READY   STATUS    RESTARTS
AGE
kube-apiserver-cluster01-control-plane   1/1     Running      0
47s
```

The API server is highlighted to have been running for only 47 seconds, showing that it successfully restarted.

Having verified that the API server is running, let's look at the audit log to verify that it's working correctly. To check the log, you can use `docker exec` to tail `audit.log`:

```
$ docker exec cluster01-control-plane tail /var/log/k8s/audit.log
```

This command generates the following log data:

```
{"kind": "Event", "apiVersion": "audit.k8s.io/v1", "level": "Metadata", "auditID": "451ddf5d-763c-4d7c-9d89-7afc6232e2dc", "stage": "ResponseComplete", "requestURI": "/apis/discovery.k8s.io/v1/namespaces/default/endpointslices/kubernetes", "verb": "get", "user": {"username": "system:apiserver", "uid": "7e02462c-26d1-4349-92ec-edf46af2ab31"}, "groups": ["system:masters"], "sourceIPs": [":1"], "userAgent": "kube-apiserver/v1.21.1 (linux/amd64) kubernetes/5e58841", "objectRef": {"resource": "endpointslices", "namespace": "default", "name": "kubernetes", "apiGroup": "discovery.k8s.io", "apiVersion": "v1"}, "responseStatus": {"metadata": {}, "code": 200}, "requestReceivedTimestamp": "2021-07-12T08:53:55.345776Z", "stageTimeStamp": "2021-07-12T08:53:55.365609Z", "annotations": {"authorization.k8s.io/decision": "allow", "authorization.k8s.io/reason": ""}}
```

There is quite a bit of information in this JSON, and it would be challenging to find a specific event by looking at a log file directly. Luckily, now that you have auditing enabled, you can forward events to a central logging server. We will do this in *Chapter 15, Monitoring Clusters and Workloads with Prometheus*, where we will deploy an EFK stack.

Now that we have auditing enabled, the next step is to practice debugging RBAC policies.

## Using audit2rbac to debug policies

There is a tool called `audit2rbac` that can reverse-engineer errors in the audit log into RBAC policy objects. In this section, we'll use this tool to generate an RBAC policy after discovering that one of our users can't perform an action they need to be able to do. This is a typical RBAC debugging process and learning how to use this tool can save you hours trying to isolate RBAC issues:

1. In the previous chapter, a generic RBAC policy was created to allow all members of the `cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com` group to be administrators in our cluster. If you're logged into OpenUnison, log out.
2. Now, log in again with the username `jjackson` and the password `start123`.
3. Next, click on **Sign In**. Once you're logged in, go to the dashboard. Just as when OpenUnison was first deployed, there won't be any namespaces or other information because the RBAC policy for cluster administrators doesn't apply anymore.
4. Next, copy your `kubectl` configuration from the token screen, making sure to paste it into a window that isn't your main KinD terminal so you do not overwrite your master configuration.
5. Once your tokens are set, attempt to create a namespace called `not-going-to-work`:

```
PS C:\Users\mlb> kubectl create ns not-going-to-work
Error from server (Forbidden): namespaces is forbidden: User "jjackson"
cannot create resource "namespaces" in API group "" at the cluster scope
```

There's enough information here to reverse-engineer an RBAC policy.

6. In order to eliminate this error message, create a ClusterRole with a resource for "namespaces", apiGroups set to "", and a verb of "create" using your KinD administrative user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: cluster-create-ns
rules:
- apiGroups: [""]
  resources: ["namespaces"]
  verbs: ["create"]
```

7. Next, create a ClusterRoleBinding for the user and this ClusterRole:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: cluster-create-ns
subjects:
- kind: User
  name: jjackson
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: ClusterRole
  name: cluster-create-ns
  apiGroup: rbac.authorization.k8s.io
```

8. Once the ClusterRole and ClusterRoleBinding are created, try running the command again, and it will work:

```
PS C:\Users\mlb> kubectl create ns not-going-to-work
namespace/not-going-to-work created
```

Unfortunately, this is not likely how most RBAC debugging will go. Most of the time, debugging RBAC will not be this clear or simple. Typically, debugging RBAC means getting unexpected error messages between systems. For instance, if you're deploying the kube-prometheus project for monitoring, you'll generally want to monitor by Service objects, not by explicitly naming Pods. In order to do this, the Prometheus ServiceAccount needs to be able to list the Service objects in the namespace of the service you want to monitor. Prometheus won't tell you this needs to happen; you just won't see your services listed. A better way to debug is to use a tool that knows how to read the audit log and can reverse-engineer a set of roles and bindings based on the failures in the log.

The audit2rbac tool is the best way to do this. It will read the audit log and give you a set of policies that will work. It may not be the exact policy that's needed, but it will provide a good starting point. Let's try it out:

- First, attach a shell to the control-plane container and download the tool from GitHub (<https://github.com/liggitt/audit2rbac/releases>):

```
root@cluster01-control-plane:/# curl -L https://github.com/liggitt/audit-2rbac/releases/download/v0.8.0/audit2rbac-linux-amd64.tar.gz 2>/dev/null > audit2rbac-linux-amd64.tar.gz  
root@cluster01-control-plane:/# tar -xvzf audit2rbac-linux-amd64.tar.gz
```

- Before using the tool, make sure to close the browser with the Kubernetes dashboard in it to avoid polluting the logs. Also, remove the cluster-create-ns ClusterRole and ClusterRoleBinding created previously. Finally, try creating the still-not-going-to-work namespace:

```
PS C:\Users\mlb> kubectl create ns still-not-going-to-work  
Error from server (Forbidden): namespaces is forbidden: User "jjackson"  
cannot create resource "namespaces" in API group "" at the cluster scope
```

- Next, use the audit2rbac tool to look for any failures for your test user:

```
root@cluster01-control-plane:/# ./audit2rbac --filename=/var/log/k8s/audit.log --user=jjackson  
Opening audit source...  
Loading events...  
Evaluating API calls...  
Generating roles...  
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRole  
metadata:  
  annotations:  
    audit2rbac.liggitt.net/version: v0.8.0  
  labels:  
    audit2rbac.liggitt.net/generated: "true"      audit2rbac.liggitt.net/  
  user: jjackson  
  name: audit2rbac:jjackson  
rules:  
  - apiGroups:  
    - ""    resources:  
    - namespaces  
    verbs:  
    - create  
---  
apiVersion: rbac.authorization.k8s.io/v1  
kind: ClusterRoleBinding  
metadata:  
  annotations:
```

```
audit2rbac.liggitt.net/version: v0.8.0
labels:
  audit2rbac.liggitt.net/generated: "true"      audit2rbac.liggitt.net/
user: jjackson
  name: audit2rbac:jjackson
roleRef:|
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole|
  name: audit2rbac:jjackson
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: jjackson
Complete!
```

4. This command generated a policy that will allow the test user to create namespaces. This becomes an anti-pattern, though, of explicitly authorizing access to users.
5. In order to better leverage this policy, it would be better to use our group:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: create-ns-audit2rbac
rules:
- apiGroups:
  - ""
  resources:
  - namespaces
  verbs:
  - create
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: create-ns-audit2rbac
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: create-ns-audit2rbac
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: cn=k8s-create-ns,ou=Groups,DC=domain,DC=com
```

The major change is highlighted. Instead of referencing the user directly, the `ClusterRoleBinding` is now referencing the `cn=k8s-create-ns,ou=Groups,DC=domain,DC=com` group so that any member of that group can now create namespaces.

## Summary

This chapter's focus was on RBAC policy creation and debugging. We explored how Kubernetes defines authorization policies and how it applies those policies to enterprise users. We also looked at how these policies can be used to enable multi-tenancy in your cluster. Finally, we enabled the audit log in our KinD cluster and learned how to use the `audit2rbac` tool to debug RBAC issues.

Using Kubernetes' built-in RBAC policy management objects lets you enable access that's needed for operational and development tasks in your clusters. Knowing how to design policies can help limit the impact of issues, providing the confidence to let users do more on their own.

In the next chapter, *Chapter 8, Managing Secrets*, we'll learn how Kubernetes manages secret data and how you should integrate external secrets into your clusters using HashiCorp Vault and the External Secrets Operator.

## Questions

1. True or false – ABAC is the preferred method of authorizing access to Kubernetes clusters.
  - a. True
  - b. False
2. What are the three components of a Role?
  - a. Subject, noun, and verb
  - b. Resource, action, and group
  - c. `apiGroups`, resources, and verbs
  - d. Group, resource, and sub-resource
3. Where can you go to look up resource information?
  - a. Kubernetes API reference
  - b. The library
  - c. Tutorials and blog posts
4. How can you reuse Roles across namespaces?
  - a. You can't; you need to re-create them.
  - b. Define a `ClusterRole` and reference it in each namespace as a `RoleBinding`.
  - c. Reference the `Role` in one namespace with the `RoleBindings` of other namespaces.
  - d. None of the above.

5. How should bindings reference users?
  - a. Directly, listing every user.
  - b. RoleBindings should only reference service accounts.
  - c. Only ClusterRoleBindings should reference users.
  - d. Whenever possible, RoleBindings and ClusterRoleBindings should reference groups.
6. True or false – RBAC can be used to authorize access to everything except for one resource.
  - a. True
  - b. False
7. True or false – RBAC is the only method of authorization in Kubernetes.
  - a. True
  - b. False

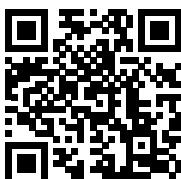
## Answers

1. a: false
2. b: Resource, action, and group
3. a: Kubernetes API reference
4. b: Define a ClusterRole and reference it in each namespace as a RoleBinding.
5. d: Whenever possible, RoleBindings and ClusterRoleBindings should reference groups.
6. b: False
7. b: False

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 8

## Managing Secrets

Everyone has secrets, and Kubernetes clusters are no different. **Secrets** can be used to store credentials for connecting to databases, private keys for encryption or authentication, or anything else that's deemed confidential. In this chapter, we'll explore why secret data has to be handled differently than other configuration data, how to model threats against your cluster's secrets, and different ways to integrate external secret managers into your clusters.

In *Chapter 6, Integrating Authentication into Your Cluster*, we created some secrets for **OpenUnison**. These Secrets were simple Kubernetes objects and weren't treated any differently than we'd treat other configuration data. This makes it difficult to follow common enterprise requirements for secret data, such as periodic rotation and tracking usage. It's important to understand why enterprises generally have these requirements and how to implement them. It's also important to be able to model threats from a realistic perspective and to avoid creating security holes by trying to make things more secure.

This chapter will walk through why you need to treat secret data different from other configuration data and provide you the tools you'll need to determine your secrets management requirements and to build out your secrets management platform. We'll cover:

- Examining the difference between Secrets and Configuration Data
- Understanding Secrets Managers
- Integrating Secrets into your Deployments

### Technical Requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 4 GB of RAM, though 8 GB is suggested
- Scripts from the `chapter8` folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Getting Help

We do our best to test everything, but there are sometimes half a dozen systems or more in our integration labs. Given the fluid nature of technology, sometimes things that work in our environment don't work in yours. Don't worry, we're here to help! Open an issue on our GitHub repo at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/issues> and we'll be happy to help you out!

## Examining the difference between Secrets and Configuration Data

What makes a Secret different than the configuration data stored in a ConfigMap or a CRD? From a Kubernetes perspective, the only real difference is that both ConfigMaps and CRDs are represented as text, whereas a Secret is represented as a **base64** encoded string, allowing secrets to contain binary data.

If you are new to base64, it is an encoding process, known for using a 64-character set that converts binary data into an ASCII character string. This provides a reliable method to send binary information during transmission as text, which is beneficial when direct binary support is unsupported or where the risk of data corruption in a plain text transmission is a concern, making it useful for transmitting images, audio, and binary files.



There can be some confusion between the terms encoding and encryption. Encryption requires a key to decode, while encoding does not. While encoding might provide some obscurity to text, it doesn't protect it. If you didn't need a key to encode your data, it's not encrypted.

Now, let's look at a Secret object:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: "2023-07-30T00:04:51Z"
  name: orchestra-secrets-source
  namespace: openunison
  resourceVersion: "2958"
  uid: 2236389e-e751-4030-8d51-96325d302815
type: Opaque
data:
  AD_BIND_PASSWORD: JHRhcNQxMjM=
  K8S_DB_SECRET: VHloc...
  OU_JDBC_PASSWORD: c3RhcnR0MTIz
  SMTP_PASSWORD: c3RhcnQxMjM=
  unisonKeystorePassword: R2VHV3...
```

This looks very similar to a `ConfigMap`, but there are two differences:

- The addition of the `type` directive tells Kubernetes what kind of Secret this is.
- All of the fields in the `data` section are base64 encoded.

The `type` directive tells Kubernetes what kind of `Secret` you're creating. In this case, `type Opaque` means that there is no format to the `Secret`'s `data` section. This will likely be the most common type you will see.

There are no requirements for the `type` directive; you can specify whatever you want if you wish to provide your own value. That said, if you do provide one of Kubernetes' pre-defined types, the cluster will validate that the format matches. You can find the list of pre-defined types in Kubernetes' `Secret` documentation: <https://kubernetes.io/docs/concepts/configuration/secret/#secret-types>

For instance, if you were to set the `type` to `kubernetes.io/tls` your `Secret` must have a key called `tls.crt` (i.e., a base64-encoded PEM encoded certificate) and a key called `tls.key` (i.e., a base64 encoded PEM private key), otherwise the API server will fail to create your `Secret` and give you an error. Here's an example:

```
$ kubectl create -f - <<EOF
heredoc> apiVersion: v1
kind: Secret
metadata:
  name: not-tls
  namespace: default
type: kubernetes.io/tls
data:
  AD_BIND_PASSWORD: JHRhcNQxMjM=
EOF
The Secret "not-tls" is invalid:
 * data[tls.crt]: Required value
 * data[tls.key]: Required value
```

The base64 encoding of data is for a very simple reason, but is also the source of considerable confusion. Kubernetes `Secret` data *must* be base64 encoded because secret data is often case sensitive or binary and so must be encoded to ensure that it survives the translation from YAML -JSON -binary storage.

It's important to understand how YAML gets stored in the Kubernetes API server to understand why.

When we work with a YAML file, the text is represented in the file by a byte (or more). This YAML is then converted to JSON by `kubectl` for interacting with the Kubernetes API. The JSON that is sent to the API server is then translated into a binary format when it is stored. The problem we run into with many text-based formats is that there are multiple ways to represent text-based data in a binary format.

For instance, **UTF-8**, which is one of the most common encodings, can use from one to four bytes to represent a certain character. **UTF-16** uses one to four 16-bit “code units”. **ASCII** can only encode the English alphabet, Arabic numerals, and common English punctuation. If the encoding from YAML -JSON - binary and back involves switches encoding types, data can be lost or corrupted.

Preserving binary data in text is where the base64 standard comes in. Base64 allows any data to be stored as ASCII text, which is a subset universally across different encoding types. This means that the base64-encoded data can be reliably transmitted across encoding types.

If you’re still skeptical of why base64 encoding your secret data is important, have you ever copied a file created on Windows to a Linux system and started seeing ^M in the text? That’s an additional risk of traversing systems: different systems represent new lines with different control characters. Base64 encoding your secret data means that the information in the YAML file is the same as what’s stored byte-for-byte.

One thing that’s incredibly important to understand is that base64 encoding is not encryption. There is no security benefit to encoding, it will not stop someone from snooping on your secrets.

Now that we know why `Secret` data is base64 encoded, why are `Secrets` their own objects? Why don’t we just base64 encode `ConfigMaps`? The answer is to more easily use RBAC to restrict access. In the last chapter, we explored Kubernetes’ RBAC system for authorizing access to resources. In the chapter before that, we explored how Kubernetes creates `ServiceAccount` tokens, which can be stored in `Secret` objects. Combining the knowledge from these two chapters, we see how storing sensitive data in a `ConfigMap` can generate unintended consequences when we consider the `view ClusterRole`, which is intended to give read-only access to a namespace. This `ClusterRole` does not include the `Secret` type, so `view` will allow you to read `ConfigMaps`, view pod status, etc., but not read a `Secret`. This is because a `Secret` might contain a token for a `ServiceAccount`, which is bound to a higher-privilege Role or `ClusterRole`, so a user with read-only access can escalate that access if we’re not careful. If secret data were stored in `ConfigMaps`, RBAC would either need to support some way to exclude resources or enumerate specific `ConfigMap` objects that should be allowed to be viewable by the `view ClusterRole`, making it likely to not be used properly.

Now that we know what makes a `Secret` different than other configuration objects, we will explore why you need to treat `Secret` objects differently than `ConfigMaps` and CRDs.

## Managing Secrets in an Enterprise

The title of this book includes the word “enterprise,” and secret data management is a big area where this is important. Most enterprises have very specific rules around secrets management. Some of these rules involve being able to audit when a secret is used, and others require that secrets be rotated on a periodic basis. Following these rules is a process referred to as “compliance” and is often one of the largest cost drivers for any enterprise deployment.

Security and compliance are often grouped together, but they do not mean the same thing. It’s quite easy to build a 100% compliant system that makes your auditors happy, but will fail to secure your applications and data. That’s why it’s important to understand why you’re building certain features into your platforms. You need to ask yourself, are they satisfying compliance, security, or both?

In order to answer this question, you need to understand the threats to your data and systems. This process is referred to as `threat modeling`, and several books have been written on the topic. In this chapter, we’re going to build a very basic threat model for Kubernetes secrets based on where they are in the application’s deployment. We’ll start with secrets at rest.

## Threats to Secrets at Rest

When a secret, such as a credential or a key, is in storage it is referred to being “at rest,” since data is not being moved. Nearly every compliance framework requires that secrets at rest be encrypted. This makes sense; why wouldn’t you encrypt your data at rest? In Kubernetes, you can configure etcd to encrypt data at rest and you may think you’ve not only met a compliance requirement (often referred to as “checking the box”) but increased the security of your cluster! The truth is much more complicated.

Before we dive into how Kubernetes encrypts data at rest, let’s do a quick recap of how encryption works. All encryption involves three basic components:

- **Data:** Either encrypted (cipher text) or decrypted (plain text).
- **Key:** Used to encrypt your plain text to cipher text and to decrypt your cipher text to plain text.
- **Algorithm:** Some process to combine the key and data to encrypt or decrypt it.

Every encryption class or book will teach you that the key is always a secret while hiding the algorithm should never be relied upon for secrecy. This means that if an attacker knows you’re using AES-256, it really doesn’t matter because it’s the key that’s the secret.

What’s important here is that if you’re using encryption, you must have a key available to encrypt and decrypt the data. There is a great deal of nuance in how the different algorithms work, differences between block and stream ciphers, how keys are generated, etc. That nuance isn’t important to this discussion, no matter how fascinating it is. The fact that you need your key and data in the same place at the same time limits the security impact of encrypting data at rest because the security is only increased if the data and the key are separate.

With that sidebar in encryption completed, you might start to see the issue with encrypting data in Kubernetes’ database. Kubernetes does support encrypting data and we’re not going to cover it here because of the complexities, other than to describe it at a high level.

Kubernetes encryption works by configuring an `EncryptionConfiguration` object that identifies what data is encrypted and with what keys. This object is accessible from the host running your API servers. Do you see the flaw with this? If someone has access to the cluster, they have the keys!

If your etcd instances are run on different servers there’s some additional security, but does that benefit offset the risks involved when you need to decrypt and re-encrypt for key rotation? That’s a decision you need to make on your own.

Does encrypting your data at rest make your clusters less secure? Consider the “**CIA triad** of security” that’s most often used to describe the security requirements and impacts on a system. CIA stands for:

- **Confidentiality:** Can we ensure we’re the only ones who have access to this data?
- **Integrity:** How sure are we that the data hasn’t been tampered with?
- **Availability:** Is the data available when we need it?

Encrypting the data helps with the C+I portion of the CIA triad, assuming we can trust the keys (we’ll talk about that in a moment). If the key rotation requires downtime, or has a high risk of an outage, our “A” can be impacted.



There's an argument that encryption doesn't provide data integrity assurances because encrypted data can be corrupted and you need signatures to validate data. **Signatures** are just a form of encryption with a private key that can be validated with a public key, so it's all still encryption. If someone can tamper with encrypted data in a way that bypasses your keys, or using your keys, then the data can't be trusted. This is why I include integrity as a benefit of encryption.

Speaking of the keys, maybe we don't store the keys locally. Kubernetes does support external key management systems. We'll dive into the details of how your cluster authenticates with a vault or **KMS** later in the chapter. For now, what's important is that to make sure that generated authentication tokens are being given to the correct system, you need a local key to use for authenticating, leading to the same impact as having the decryption key local to your cluster: a local compromise means an attacker will have both the keys and the encrypted data at the same time and so can decrypt it.

So, you've gone through the process of encrypting data in your cluster and you've checked the box. Is your cluster secure? This is where security and compliance aren't the same thing. If you've deployed this encrypted database onto a system with a single user account that's shared across multiple users, you've created a new way to get attacked while thinking you're secure! It's important to note that most compliance frameworks still require some form of authorization management, but many vendors push this off to another system and often the answer is "we keep the keys in a vault," creating a circular compliance issue. These complexities are what make securing Kubernetes and being compliant so difficult.

Having examined how we can approach encryption of Kubernetes' data at rest, we'll next explore threats to Kubernetes' secrets in transit between systems.

## Threats to Secrets in Transit

After spending time looking at the issues with encrypting data at rest you may think that the same issues apply to data in transit. The keys need to be in the same place as the data, so why bother?

It turns out this isn't the case. Kubernetes and API driven certificate authorities like **JetStack's cert-manager** make certificate management pretty much non-existent. We already deployed cert-manager in the authentication chapter with an internal certificate when we tested out pipeline authentication. We deployed cert-manager with a private key and a self-signed root certificate that is good for ten years. We trust that certificate throughout our cluster and configure our Ingress objects to use that internal CA to generate three-month certificates. A combination of NGINX and cert-manager make sure that we don't ever think about renewing certificates.

For intracluster communications, you can use the same approach, or you can deploy a service mesh like Istio to generate certificates and provide TLS. We'll dive into this later in the book.

From an availability standpoint, data in transit is much more ephemeral than data at rest. If there is a break in availability due to an expired certificate, there are technologies to perform retries that can be used to mitigate this risk.

The point is there's no reason not to encrypt your data in transit. It's true that the CA and private key are still in the cluster, so a compromised cluster leads to decryptable traffic, but the likelihood of availability going down due to key rotation is much reduced, making this a much easier decision.

If encrypting data in transit increases security, is it compliant? This is where we get the opposite of the "data at rest" scenario. From a technical perspective, a certificate authority is easy to build. Back before we had cert-manager or Kubernetes, I built a simple API-based CA for a customer that wanted to lock down APIs from a mobile app using Java and the `openssl` command. Building a compliant CA is much harder. It often involves volumes of management and regulations. For this reason, while most large enterprises have an internal CA that could be used, you can't use it from within Kubernetes. If your cluster doesn't match all the rules of your CA, it invalidates the compliance of those rules and breaks your compliance.

The compromise that is often struck is for your ingress controller to have a wildcard certificate while an internal CA is used for communications within a cluster.

There is a strong argument that the increase in automation overcomes the introduced weakness of an in-cluster key for a CA, but as compliance is often a legal requirement those arguments generally fail. This is why it's so important to understand the difference between security and compliance. In these two use cases we've shown how they conflict and why you need to understand those conflicts to make design choices.

Having walked through how to encrypt secret data in transit, the last scenario to explore is secrets when they're used in your applications.

## Protecting Secrets in Your Applications

Let's walk through a potential, and all too common, scenario. You've built a "secure" cluster. Your secret data is all stored in a well-designed secrets manager. You're following all the guidance on how to manage that data. Every connection is encrypted. Your application loads, gets a password, and connects to a database. It turns out that two years ago someone found a flaw in your parsing library that lets an attacker open a shell on your app and get access to that password and since you need to talk to that database they can connect and extract all the data!

What went wrong? Going back to the previous two scenarios, we established multiple times that you must have the secret in hand to use it. This means that if your application has a security flaw, it doesn't matter how well designed your secrets management process is, it becomes the weakest link.

This doesn't mean we should abandon secrets management. Supply chain security is its own focus and one we will cover later in this book. The point is that when you consider how to build your secrets management systems and processes, remember that your application is probably the easiest place to lose control and you must plan accordingly. For instance, adding additional layers that impact automation will not likely buy you additional security, but you may push your developers into spending time to work around your systems or drive up costs unnecessarily.

Now that we've walked through how secrets can be attacked, we can explore how secrets managers work and look at different strategies for managing secret data in your clusters.

# Understanding Secrets Managers

We've covered what makes Secrets special and how to approach secret data, now we need to talk about how to manage them. There are four ways most clusters manage Secrets:

- **Kubernetes Secrets:** Storing all secrets as Secret objects without any kind of external management.
- **Sealed Secrets:** Secret data is encrypted in files stored in Git.
- **External Secrets Manager:** An external service, such as HashiCorp's Vault or a cloud-based secrets manager, is used to store secrets for your cluster.
- **Hybrid:** By syncing secret data from an external secrets manager into generic Kubernetes Secret objects you get an approach that allows for the Secrets API while still maintaining your source of truth about secret data outside of your cluster.

Let's walk through each approach to managing secrets.

## Storing Secrets as Secret Objects

The first option seems like the easiest. Leveraging Kubernetes Secret objects provides several benefits:

- There's a standard API for accessing Secret objects.
- The API can be restricted via RBAC, mostly.
- There are multiple ways for containers to access Secret objects without having to make API calls.

The last two points can be a double-edged sword. When Kubernetes was first created, one of the goals was to allow application developers to run a workload on Kubernetes without the application knowing anything about Kubernetes. This meant that having a standard Secret API was less important than having an easy way for applications to access the secret data. To this end, Kubernetes made the easiest path to accessing secret data either mounting the secrets as virtual files in the container or setting them as environment variables. We'll discuss the benefits and risks of both approaches later in this chapter. The impact of this design decision is that while you can limit who can access a Secret via an API using RBAC, you can't limit who can mount a Secret into a pod within a Namespace.



This point been said previously in this book and will be repeated often. The Namespace is the security boundary in Kubernetes. If you want to limit access to specific Secrets within a Namespace, it's time to create a new Namespace.

In April 2022, Mac Chaffee wrote a great blog post titled *Plain Kubernetes Secrets are fine* (<https://www.macchaffee.com/blog/2022/k8s-secrets/>) where he gave a great summary of why, from a security standpoint, it's OK to use plain Kubernetes Secrets. The blog post points out that you need to model the threats to your secret data before assuming a path forward for securing them. You may recognize many of the same arguments from this blog post in the previous section. Mac did a better job of articulating what I always thought to be true, and I really enjoyed his approach. The "too long, didn't read" of the post is:

- Secret managers, like Vault, are rarely deployed in a way that adds any additional security over any other key/value database.

- Encrypting secrets at rest doesn't accomplish anything.
- Your application is the most likely spot where you'll lose a secret.

If a Kubernetes Secret is fine, why are we bothering with secrets managers at all? There are two reasons: compliance and GitOps.

From a compliance perspective, most compliance frameworks require that you not only know when secret data changes, but also when it's used. For instance, NIST-800-53 requires that you continuously monitor the usage of credentials (which makes up the bulk of secret data). While you could set up logging in Kubernetes to track this, it makes it much easier to audit by having it in a central location.

The next reason why we should evaluate a secrets manager is GitOps. In the last two chapters, we're going to explore GitOps, a large part of which is storing our configuration in a Git repository. You should never, ever, EVER, EVER store secret data in a Git repository either in plain text or as encrypted data. Git repositories are designed to be easily forked. Once forked, you've lost control of that repository. Going back to compliance, this is a big risk as you have no way of knowing if a developer forked your internal repository and accidentally pushed it to a public GitHub repository. There are other reasons why keeping secrets in Git should be considered an anti-pattern, but we'll cover that when we talk about sealed secrets. Using a secrets manager allows us to externalize our secret data from our cluster, even though Secret objects are most likely fine for most clusters.

Having looked at why regular Kubernetes Secrets are usually fine from a security standpoint, let's look at what sealed secrets are and why they're an anti-pattern.

## Sealed Secrets

If you are externalizing your Kubernetes manifests into a Git repository, you may be tempted to store sensitive secret data there too. You don't want anyone to get that data though, so you decide to encrypt it. Now, however, you need to decrypt it to get it back into your clusters. Bitnami (now owned by VMware) released a tool called Sealed Secrets (<https://github.com/bitnami-labs/sealed-secrets>) that does just this. You install the operator into your cluster and when it sees a SealedSecret, it decrypts it for you.

This seems like a simple and elegant solution for externalizing secret data securely. Unfortunately, its apparent simplicity is what leads to this solution being an anti-pattern.

The first issue with Sealed Secrets is that secret data is stored in Git. We pointed out in the previous section that this is a bad idea from a security perspective. One of the main goals of secrets management is being able to track the usage of secrets. It's incredibly easy for a developer to push an internal repository to a public service like GitHub or GitLab. Take this simple command line:

```
$ git remote set-url origin https://github.com/new-repository.git  
$ git push
```

So long as the user is signed into GitHub, your internal repo is now public! In theory you could limit access to public Git repositories, but that would probably be counterproductive. My general advice is never put something in Git that you wouldn't want on GitHub. Once the code is on GitHub, or any other remote repository, you've lost all control.

Your first response to this issue of pushing to public repositories may be: “But it’s encrypted!” As a species, we’re bad at keeping secrets. As an industry, we’re very bad at protecting the keys used to encrypt secrets. If you’ve lost the repository, there’s a good chance you’ll also lose the keys.



*Three may keep a secret, if two are dead – Benjamin Franklin*

---

This isn’t just true in the Kubernetes world, but really any technology. This is why it’s so important to plan for losing secrets and being able to quickly change them. If your Git repository with secrets, whether encrypted or not, were to be pushed outside of your enterprise, you’d want to:

1. Go through all of the Sealed Secrets and generate new secret data (i.e. credentials).
2. Generate a new encryption key.
3. Re-encrypt and re-post all of the Sealed Secrets to Git.

Depending on the size of your clusters and how well you manage your keys this could become a monumental task very quickly. It turns out a secret manager is pretty good at handling this failure mode.

In addition to being able to handle the failure mode of losing your repository, you need to account for losing the keys used to encrypt and decrypt your secrets. If you lose the key used to encrypt the secrets and lose the secrets...suffice it to say that you’re setting yourself up for a bad week, or what some call a *resume-building event*.

While Sealed Secrets appears to be a simple way to handle secrets management, they fail to account for failure in a way that would be acceptable in the aftermath of most breaches. While you wouldn’t want to store your secret data in Git, it is acceptable to store metadata about secrets in Git. We’ll see in the next section that secret managers can be integrated into your cluster using metadata that describes where to get secret data without having any secrets in the repository.

## External Secrets Managers

In the last section we discussed why storing *Secrets* in Git, whether encrypted or not, is an antipattern. We also discussed that you may want to externalize your secret data management to make compliance and *GitOps* easier. The most common approach to satisfying these requirements is a secrets manager.

Secrets managers are key/value databases that have some additional features not often found in generic key/value databases:

- **Authentication:** Secrets managers can generally authenticate with multiple methods. The best solutions allow you to use either your Pod’s credentials directly or using a credential derived from it. This allows your secret manager to track which workloads are working with secret data and provide richer policies for managing access to that data.
- **Policies:** Most secret managers provide a richer policy framework than generic databases. When combined with flexible authentication, options for the secrets manager can help lock down secrets to the workload while also tracking usage without an administrator getting involved with each onboarding.

- **Auditing:** In addition to tracking changes, secrets managers track reads as well. This is a key compliance requirement.

The authentication tools for secrets managers are important. It doesn't really add much to your security or your compliance if you're using a generic credential to access your secrets manager. Back in *Chapter 6, Integrating Authentication into Your Cluster*, we discussed how each pod gets a unique identity based on their ServiceAccount and how that ServiceAccount can be validated either by using the cluster's OIDC discovery document or by submitting a TokenReview to validate that the token is still valid. This token should be used when communicating with your secret manager. If you're running on a cloud-managed Kubernetes, this can also be an identity supplied by your cloud. The point is, you're using a local identity, not a static key. This local identity is what shows up in your audit logs, allowing your security team and auditors to know who is accessing secrets.

Finally, utilizing your Pod's identity to access your secrets manager makes onboarding and automation easier. We'll look at multiple forms of multitenancy later in the book, which all have automation in common. Most secrets managers make it easier to design policies that allow for segmenting secrets access based on information in the authentication token, such as the namespace. This means you don't need to make API calls to your secrets manager whenever you onboard a new tenant.

There are several secrets managers available; every major cloud has its own offering and there are several open-source managers:

- **HashiCorp Vault:** <https://github.com/hashicorp/vault>
- **CyberArk Conjur:** <https://github.com/cyberark/conjur>
- **VMware Tanzu Secrets Manager:** <https://github.com/vmware-tanzu/secrets-manager>

We don't want you to have to sign up for a cloud service, so for the examples in this chapter (and whenever we need secrets for the rest of the book), we'll use HashiCorp's Vault.



In August, 2023, HashiCorp announced a change of license for its projects, including Vault, from the Mozilla Public License (MPL) to the Business Source License (BSDL). While the BSDL is not an approved Open Source™ license from the Open Source Institute, it does allow for free use in both production and non-production environments. We decided to continue with Vault because even though the community around HashiCorp's projects is making calls for forks or moves, enterprises have invested hundreds of thousands of dollars between software, people, and automation for Vault deployments. It's still the most common secrets manager and likely will be for some time.

To deploy Vault, start with a fresh cluster and run the `chapter8/vault/deploy_vault.sh` script:

```
$ cd chapter8/vault/  
$ ./deploy_vault.sh
```

This script deploys vault into your cluster by:

1. Deploying cert-manager with a self-signed CA for ingress certificates
2. Installing the Vault Helm chart

3. Deploying Vault into the cluster with the UI and with ClusterIP Service objects
4. Retrieving the keys used to unseal the Vault database
5. Unsealing the Vault database
6. Deploying Ingress objects you can access the UI and web services via NGINX

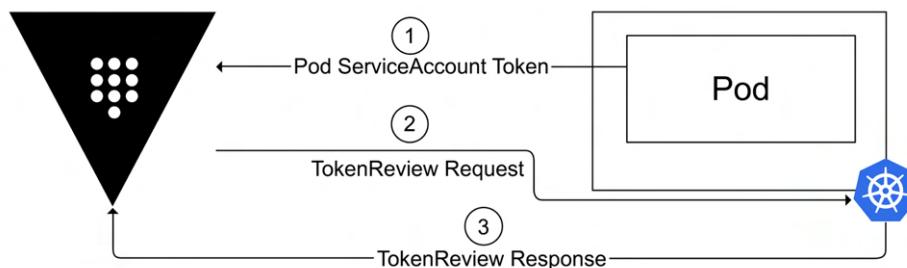
Vault encrypts its data, so when you start the pod you need to “unseal” it so it can be managed. You can log in to your Vault instance by first retrieving the token from the `~/unseal-keys.json` file generated by the deployment:

```
$ jq -r '.root_token' < ~/unseal-keys.json
hvs.OFotf6LlPTI1bRhqNDMyYf7N
```

Next, use this token to log in to Vault by going to `https://vault.apps.IP.nip.io`, where IP is the IP address of your Kubernetes cluster with dashes instead of dots. For instance, our cluster is at 192.168.2.82 so our Vault URL is `https://vault.apps.192-168-2-82.nip.io/`.

We’re not going to dig too deeply into how Vault is configured outside of specific examples to illustrate how to integrate an external secrets manager into your cluster. If you want to go through all its options, the documentation is available online at <https://developer.hashicorp.com/vault/docs>. It’s also important to point out that this isn’t a production-capable deployment either, since it’s not highly available nor are there any processes built around starting and managing onboarding.

With our Vault deployed, the next step is to integrate Vault into our cluster. Earlier, we discussed that it’s important to use the Pod’s identity to interact with your secrets manager. We’ll do that with Vault by configuring Vault to submit a `TokenReview` to our cluster to validate that the token was issued by our cluster and that the pod that the identity is tied to is still running:



*Figure 8.1: Vault integration with Kubernetes*

The above diagram shows the flow:

1. A pod makes a request to the Vault API using its ServiceAccount token projected via the `TokenRequest` API
2. Vault submits a `TokenReview` request to the API server with the Pod’s token
3. The API server validates whether the token is still valid or not.

Using the above process gives us the ability to validate a Pod's token, getting confidence that the pod it's assigned to is still valid. If an attacker were to exfiltrate a Pod's token and attempt to use it after the Pod's been destroyed, then `TokenReview` will be rejected.



In addition to using the `TokenReview` API, Vault can be configured to use OIDC to validate tokens without a callback to the API server. We aren't going down this route because we want Vault to validate that the pod associated with the token is still valid.

To integrate our Vault into our cluster, run `chapter8/vault/vault_integrate_cluster.sh`:

```
$ cd chapter8/vault/  
$ ./vault_integrate_cluster.sh
```

This script will integrate your Vault deployment with the KinD cluster by:

1. Creating the `vault-integration` namespace and the `vault-client` ServiceAccount
2. Creating a ClusterRoleBinding for the `vault-client` ServiceAccount to the `system:auth-delegator` ClusterRole
3. Creating a token for the `vault-client` ServiceAccount that's good for about a year
4. Creating an Ingress for our API server so that Vault can communicate with it
5. Creating a Vault Kubernetes authentication configuration with our cluster

The Vault deployment is being treated as a stand-alone deployment, even though it's running on our cluster because Vault is often run this way in enterprise deployments. Vault is a complex system that requires highly specialized knowledge to run, so it's much easier to centralize Vault knowledge into a centralized team.

You might also notice that we're creating a token for Vault to use to interact with your cluster that's good for a year. This violates our goal of using short-lived tokens. This is a chicken-and-egg problem, because Vault needs to authenticate to Kubernetes in order to validate the token. The cluster could be configured to allow anonymous `TokenReview` access, which would leave it opened to potential escalation attacks.

It would be great if Vault supported using its own OIDC tokens to talk to Kubernetes like a workload we defined in *Chapter 6, Integrating Authentication into Your Cluster*, but that's not a capability at this time.

We've covered what makes an external secrets manager different than other key/value stores and deployed HashiCorp's vault. Vault's been deployed and integrated into the cluster. You now have a foundation for working with externalized secrets and exploring the different ways to integrate that foundation into your cluster. Next, we'll cover a hybrid approach between using an external secrets manager and native Kubernetes Secrets.

## Using a Hybrid of External Secrets Management and Secret Objects

So far, we have covered the use of generic Kubernetes Secret objects, the antipattern of storing secret data in encrypted files in git, and finally using an external secrets manager. Since we've already established that plain Kubernetes Secrets are likely not a substantial risk based on our threat model, but we prefer to externalize secrets into a tool like Vault, it would be great if we could use the Kubernetes Secrets API to access secret data in external vaults.

The use of the `Secret` API to access external secret data is unlikely to ever happen. However, we can synchronize secret data from our Vault into Kubernetes `Secret` objects. This hybrid approach allows us to get the best of both approaches:

- **Centralized secret data:** The source of truth for our secret data is our vault. The data in the `Secret` object is a replica of that data.
- **Metadata can be stored in git:** The metadata that's used to describe where the secret data is stored is not itself secret. It can be stored in Git without the same adverse consequences of storing the actual secrets in Git.
- **Audit Data:** The access audit logs can be configured in both the API server's access logs and the vault's access logs.

There are multiple projects that support synchronizing secrets from a vault into Kubernetes. The two that come up the most are:

- **Kubernetes Secret Store CSI Driver** (<https://secrets-store-csi-driver.sigs.k8s.io/introduction>): The Secret Store CSI driver is a **special interest group (SIG)** in the Kubernetes project provides a storage driver for accessing secret stores such as Vault. It includes a synchronization engine that will generate generic `Secret` objects. The main challenge of using this project is that before you can synchronize from a vault into a `Secret`, you need to mount it a pod.
- **External Secrets Operator** (<https://external-secrets.io/latest/>): The External Secrets Operator project provides a direct synchronization of secret data into `Secret` objects.

This section will focus on using the **External Secrets Operator** project. We chose to use External Secrets Operator because it doesn't require a pod to first mount the secret data before synchronizing it into a `Secret`. First, deploy the synchronization operator:

```
$ cd chapter8/external-secrets  
$ ./install_external_secrets.sh
```

The above script does several things:

- Deploys **External Secrets Operator**
- Creates a Namespace to store the synchronized `Secret` object
- Creates a `ServiceAccount` to access Vault
- Creates a secret password in Vault
- Creates a policy in Vault to access the secret with the above `ServiceAccount`
- Creates a `ExternalSecret` object to tell the operator where and how to synchronize our secret data into our cluster

There's quite a bit going on here. The operator itself is deployed via a Helm chart. The namespace where we keep the Secret and its associated ServiceAccount builds off of our Vault integration to allow pods to use a specific identity instead of using a static ServiceAccount's token. After creating the Namespace and ServiceAccount, a Vault policy is created to allow the ServiceAccount to read the secret data. Finally, a SecretStore and ExternalSecret object is created to tell the operator how to synchronize the secret data. Let's take a look at these objects. First, we created the SecretStore to tell the operator where the Vault is and how to access it:

```
apiVersion: external-secrets.io/v1beta1
kind: SecretStore
metadata:
  name: vault-backend
  namespace: my-ext-secret
spec:
  provider:
    vault:
      server: "https://vault.apps.192-168-2-82.nip.io"
      path: "secret"
      version: "v1"
    caProvider:
      type: "ConfigMap"
      name: "cacerts"
      key: "tls.crt"
      namespace: "my-ext-secret"
    auth:
      mountPath: "kubernetes"
      serviceAccountRef:
        name: "ext-secret-vault"
```

The first object, SecretStore, tells **External Secrets Operator** where the secrets are stored and how to access them. In this case, we're connecting to Vault using the URL `https://vault.apps.192-168-2-82.nip.io` using the certificate stored in the cacerts ConfigMap to trust for TLS. The auth section tells the operator how to authenticate, using a token for the ServiceAccount `ext-secret-vault`. With the SecretStore defined, the next step is to begin defining what Secret objects need to be created.

In order to synchronize secret data into Secret objects, there needs to be an ExternalSecret object:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: my-external-secret
  namespace: my-ext-secret
spec:
  refreshInterval: 1m
```

```
secretStoreRef:  
  kind: SecretStore  
  name: vault-backend  
target:  
  name: secret-to-be-created  
  creationPolicy: Owner  
data:  
  - secretKey: somepassword  
    remoteRef:  
      key: /data/extsecret/config  
      property: some-password
```

The `ExternalSecret` object defines how to synchronize data from your vault into your cluster. Here, the data is being pulled from the `SecretStore` that was created to communicate with the Vault deployment. This object tells **External Secrets Operator** to create the `somesecret` key on the Secret `secret-to-be-created` from the `/data/extsecret/config` object in Vault, getting the value from the `some-password` property.

To give some context, here's the Vault configuration from the script:

```
vault kv put secret/data/extsecret/config some-password=mysupersecretp@ssw0rd
```

Once the synchronization process runs, we see that there's now data from Vault in our Secret:

```
$ kubectl get secret secret-to-be-created -n my-ext-secret -o json | jq -r  
'.data.somepassword' | base64 -d
```

```
mysupersecretp@ssw0rd
```

Given the approach provided by the **External Secrets Operator** project, metadata for where and how to access secret data can be created and stored in a git repository without adverse security impacts. Clusters are able to access secrets using the well-defined Secrets API while still getting the benefits of externalizing their secret data.

In the next section, we're going to take the secret data that is now available in our cluster and look at how to consume that data from workloads.

## Integrating Secrets into Your Deployments

So far, this chapter has been focused on how to store and manage secret data. We've covered different strategies for managing secrets with their associated risks and benefits. In this section, the focus will be on consuming that secret data in your workloads.

There are four ways that a workload can consume secret data:

- **Volume mounts:** Similar to reading a file from a `PersistentVolumeClaim`, secrets can be mounted to a pod and be accessed as a file. This approach can be used with both external secrets and with `Secret` objects. This is generally the preferred approach when working with security teams. If a `Secret` is updated while a pod is running, the volume will eventually get updated, though this can take some time based on your Kubernetes distribution.
- **Environment variables:** Secret data can be injected into environment variables and consumed from the workload like any other environment variable. This is often referred to as “insecure” since it’s a common practice for application developers to dump environment variables for debugging purposes. It’s not an uncommon occurrence for debugging components to be accidentally kept in production that are leaking environment variables. It’s better to avoid this approach if possible. It’s important to note that if a `Secret` is updated while a pod is running, the environment variable in the running pod is not updated.
- **Secrets API:** Kubernetes is an API and `Secrets` can be accessed directly via the `Secret API`. This approach provides more flexibility than either environment variables or volume mounts, but requires knowledge of how to call the API. If you need to be able to dynamically retrieve `Secrets`, this is a good approach but is probably overkill for most applications.
- **Vault API:** Every external vault provides an API. While we’re using tools like `External Secrets Operator` or a sidecar to interact with these APIs, there’s nothing stopping a developer from calling these APIs on their own. It would cut down on the external configuration, but at the cost of tightly binding your system to a particular project or vendor.

Next, we’re going to walk through these options to see how they’re implemented.

## Volume Mounts

The preferred way to add `Secrets` to your workloads is to treat them as files and load them into your application. This approach has multiple advantages over the other approaches listed above:

- **Less likely to be leaked during debugging:** There’s nothing that stops a developer from printing the contents of a file to the logs or an output stream, but a call to the `env` command won’t automatically print out secret data.
- **Can be updated:** When a file is updated, that update is reflected in your pod. The same is true for secret data that is mounted via a volume. If the application in your pod knows to look for updates, it will eventually get them.
- **Richer options:** A configuration file mounted onto a volume can be more than name/value pairs. It can be full configuration files, simplifying management.

Mounting a secret as a volume has been a feature of Kubernetes since `Secrets` were available. In this section we’ll walk through mounting both generic Kubernetes `Secret` objects and interacting with our Vault deployment directly using the `Vault Sidecar`.

## Using Kubernetes Secrets

Mounting a Kubernetes Secret as a volume into your pods is a matter of naming the Secret in your spec. For instance, if you created the pod from `chapter8/integration/volumes/volume-secrets.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test-volume
  name: test-volume-secrets
  namespace: my-ext-secret
spec:
  containers:
  - image: busybox
    name: test
    resources: {}
    command:
    - sh
    - -c
    - 'cat /etc/secrets/somepassword'
  volumeMounts:
  - name: mypassword
    mountPath: /etc/secrets
  volumes:
  - name: mypassword
    secret:
      secretName: secret-to-be-created
  dnsPolicy: ClusterFirst
  restartPolicy: Never
```

It will generate the following log:

```
$ kubectl logs test-volume-secrets -n my-ext-secret
myN3wP@ssw0rd
```

This pod added the Secret we synchronized from Vault directly to our pod. We can update that secret in Vault and see what happens to the mounted value in a longer-running pod. First, create the pod in `chapter8/integration/volumes/volume-secrets-watch.yaml`:

```
$ cd chapter8/integration/volumes
$ kubectl create -f ./volume-secrets-watch.yaml
$ kubectl logs -f test-volumes-secrets-watch -n my-ext-secret
Fri Sep 15 14:52:45 UTC 2023
```

```
myN3wP@ssw0rd
-----
Fri Sep 15 14:52:46 UTC 2023
myN3wP@ssw0rd
-----
Fri Sep 15 14:52:47 UTC 2023
myN3wP@ssw0rd
-----
```

Now that we're watching our mounted volume, let's update the secret in Vault:

```
$ . ./vault/vault_cli.sh
$ vault kv put secret/data/extsecret/config some-password=An0therPassw0rd
Success! Data written to: secret/data/extsecret/config
$ kubectl get secret secret-to-be-created -n my-ext-secret -o json | jq -r
'.data.somepassword' | base64 -d
An0therPassw0rd
```

It may take a minute or two, but the `Secret` in our cluster gets synchronized. Next, let's look at our running pod:

```
$ kubectl logs -f test-volumes-secrets-watch -n my-ext-secret
Fri Sep 15 14:52:45 UTC 2023
myN3wP@ssw0rd
-----
.
.
.
Fri Sep 15 14:56:33 UTC 2023
An0therPassw0rd
-----
Fri Sep 15 14:56:34 UTC 2023
An0therPassw0rd
-----
Fri Sep 15 14:56:35 UTC 2023
An0therPassw0rd
-----
```

We can see that the volume did get updated. So, if we do have a long running pod, we can watch mounted volumes to look for updates.

In this section, we integrated a `Secret` directly into our pod. When using an external secrets vault, such as HashiCorp's Vault, this requires synchronizing the `Secret` using a tool like **External Secrets Operator**. Next, we'll create a volume directly from Vault using Vault's injector sidecar.

## Using Vault's Sidecar Injector

In the previous section, we integrated a generic Kubernetes Secret into our pod. In this section, we'll integrate directly with Vault using its injector sidecar.



A **sidecar** is a special container that runs along with your primary workload to perform additional functions transparently and independently of your main workload. The sidecar pattern allows you to create containers that intercept network traffic, manage logs, or in the case of Vault, inject secrets. Starting in 1.28, sidecars moved from being a well-known pattern to becoming a first-class configuration option. This approach is still very new and has not yet been adopted by most implementations. You can learn more about the changes to sidecars in the Kubernetes blog at <https://kubernetes.io/blog/2023/08/25/native-sidecar-containers/>.

Vault's **sidecar injector** has two primary components that let us inject secret data into our pods:

- **Injector Mutating Admission Controller:** We'll cover admission controllers in more detail in *Chapter 11, Extending Security Using Open Policy Agent*. For now, what you need to know about this controller is that it looks for pods to be created with specific annotations to configure the sidecar that interacts with the Vault service.
- **Sidecar:** The sidecar does the work of interacting with our Vault deployment. You don't need to configure the sidecar manually, the admission controller mutator will do that for you based on annotations.

First, let's look at our pod that is getting secret data directly from Vault:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: watch-vault-volume
  name: test-vault-vault-watch
  namespace: my-ext-secret
  annotations:
    vault.hashicorp.com/agent-inject: "true"
    vault.hashicorp.com/service: "https://vault.apps.192-168-2-82.nip.io"

    vault.hashicorp.com/log-level: trace
    vault.hashicorp.com/role: extsecret
    vault.hashicorp.com/tls-skip-verify: "true"
    vault.hashicorp.com/agent-inject-secret-myenv: 'secret/data/extsecret/
config'
    vault.hashicorp.com/secret-volume-path-myenv: '/etc/secrets'
    vault.hashicorp.com/agent-inject-template-myenv: |
      {{- with secret "secret/data/extsecret/config" -}}
```

```

MY_SECRET_PASSWORD="{{ index .Data "some-password" }}"
{{- end }}

spec:
  containers:
    - image: ubuntu:22.04
      name: test
      resources: {}
      command:
        - bash
        - -c
        - 'while [[ 1 == 1 ]]; do date && cat /etc/secrets/myenv && echo "" && echo
"-----" && sleep 1; done'
      dnsPolicy: ClusterFirst
      restartPolicy: Never
      serviceAccountName: ext-secret-vault
      serviceAccount: ext-secret-vault

```

Let's focus on the Vault connection options:

```

vault.hashicorp.com/agent-inject: "true"
vault.hashicorp.com/service: "https://vault.apps.192-168-2-82.nip.io" vault.
hashicorp.com/log-level: trace
vault.hashicorp.com/role: extsecret
vault.hashicorp.com/tls-skip-verify: "true"

```

The first annotation tells the admission controller that we want to generate the sidecar configuration for this pod. The next annotation tells Vault where the Vault service is. We then enable detailed logging and then set the role for authentication. This role in Vault was created in `chapter8/external-secrets/install_external_secrets.sh` with the following:

```

vault write auth/kubernetes/role/extsecret \
  bound_service_account_names=ext-secret-vault \
  bound_service_account_namespaces=my-ext-secret \
  policies=extsecret \
  ttl=24h

```

The role maps our ServiceAccount to an allowed policy. Finally, we tell the agent to skip TLS verification. In a production deployment, you don't want to skip TLS verification. We could mount our CA certificate, and that's what you'll want to do in production.

Notice, we didn't specify a key or a Secret for authentication. That's because we're using our Pod's own identity. The `serviceAccount` and `serviceAccountName` options on the pod dictate which identity is used. When we configured external secrets, we use the `ext-secret-vault` ServiceAccount, so we reused that identity here.

Having defined how we talk to Vault, next let's look at how we define our data:

```
vault.hashicorp.com/agent-inject-secret-myenv: 'secret/data/extsecret/config'

vault.hashicorp.com/secret-volume-path-myenv: '/etc/secrets'

vault.hashicorp.com/agent-inject-template-myenv: |
  {{- with secret "secret/data/extsecret/config" -}}
  MY_SECRET_PASSWORD="{{ index .Data "some-password" }}"
  {{- end }}
```

The first annotation says, “Create a configuration called `myenv` that points to the Vault object `/secret/data/extsecret/config`.” `myenv` is like a variable that allows you to track configuration options across multiple annotations. The next annotation says that we want to place everything into the `/etc/secrets/myenv` file. If we didn’t specify this annotation, the sidecar would have put the resulting `myenv` file in the `/vault/secrets` directory.

The final annotation creates the content of the `myenv` file. If the syntax looks like Helm, that’s because it uses the same template engine. Here, we’re creating a file with a name/value pair. This can be a configuration file template too.

Now that we’ve walked through the configuration, let’s create the object:

```
$ cd chapter8/integration/volumes
$ ./create-vault.sh
$ kubectl logs -f test-vault-vault-watch -n my-ext-secret
Defaulted container "test" out of: test, vault-agent, vault-agent-init (init)
Fri Sep 15 18:24:09 UTC 2023
MY_SECRET_PASSWORD="An0therPassw0rd"

-----
Fri Sep 15 18:24:10 UTC 2023
MY_SECRET_PASSWORD="An0therPassw0rd"
```

Our original workload is unaware of Vault, but is able to retrieve the secret data from the generated template. There are scenarios where Vault will refresh the template, but they’re Vault-specific configuration options that we’re not going to dive into.

The annotation-based injection of sidecars is a common pattern amongst secrets management integrations with Kubernetes. If you’re looking to integrate other external secrets management systems, this is a consistent approach to use.

We’ve looked at how to use volumes to bind secrets to workloads using both standard Kubernetes `Secret` objects and our external Vault. Both approaches allow you to externalize your secrets. Next, we’ll look at how to inject secret data as environment variables.

## Environment Variables

Using environment variables is the easiest way to consume any data. Every language and platform has a standard way to access an environment variable. The downside to this ease of access is that it's common for developers to print out or dump all their environment variables for debugging. This means that the data could end up in a log, or even worse a debugging webpage that prints out environment variables. This mechanism will often be flagged by security teams, so it should be avoided if possible. That said, some workloads require environment variables, so let's look at how to integrate both plain Kubernetes Secrets and secrets from our external Vault into a pod.

### Using Kubernetes Secrets

Kubernetes can plug a Secret object's data directly into an environment variable inside of the Pod definition of the container. Here's a simple example:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: test
  name: test-envvars-secrets
  namespace: my-ext-secret
spec:
  containers:
  - image: busybox
    name: test
    resources: {}
    command:
    - env
    env:
    - name: MY_SECRET_PASSWORD
      valueFrom:
        secretKeyRef:
          name: secret-to-be-created
          key: somepassword
  dnsPolicy: ClusterFirst
  restartPolicy: Never
```

Looking at `.spec.containers[name=test].env`, you can see that we create an environment variable from an existing Secret. The command of this container is simply the `env` command, which prints out all the environment variables. To see this container in action, apply the YAML from the `chapter8/integration/envvars/envvars-secrets.yaml`:

```
$ cd chapter8/integration/envvars
$ kubectl create -f ./envvars-secrets.yaml
```

```
$ kubectl logs -f -l run=test -n my-ext-secret
MY_SECRET_PASSWORD=An0therPassw0rd
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
HOME=/root
```

What happens if we update our Secret? Kubernetes won't update the environment variable. Let's verify this point. First, create a pod that watches our environment variables:

```
$ cd chapter8/integration/envvars
$ kubectl create -f ./envvars-secrets-watch.yaml
$ kubectl logs -f -l run=watch-env -n my-ext-secret
Fri Sep 15 12:43:30 UTC 2023
MY_SECRET_PASSWORD=An0therPassw0rd
Fri Sep 15 12:43:31 UTC 2023
MY_SECRET_PASSWORD=An0therPassw0rd
```

This pod will continuously echo the environment variable we created. Next, let's update Vault:

```
$ . . . /vault/vault_cli.sh
$ vault kv put secret/data/extsecret/config some-password=myN3wP@ssw0rd
Success! Data written to: secret/data/extsecret/config
$ kubectl get secret secret-to-be-created -n my-ext-secret -o json | jq -r
'.data.somepassword' | base64 -d
myN3wP@ssw0rd
```

It can take up to a minute for our new password to be synchronized into the Secret. Wait until that synchronization is complete. Once synchronized, let's watch our logs:

```
ubuntu@book-v3:~/Kubernetes-An-Enterprise-Guide-Third-Edition/chapter8/vault$ kubectl logs -f -l run=watch-env -n my-ext-secret
Fri Sep 15 12:48:41 UTC 2023
MY_SECRET_PASSWORD=An0therPassw0rd
Fri Sep 15 12:48:42 UTC 2023
MY_SECRET_PASSWORD=An0therPassw0rd
Fri Sep 15 12:48:43 UTC 2023
MY_SECRET_PASSWORD=An0therPassw0rd
```

The environment variables in the pod are not updated. You can continue to let this run, but it won't change. You'll need something to watch the Secret and restart your workload if you want to support dynamic changes with environment variables.

Now that we're able to incorporate a Kubernetes Secret as an environment variable, next, we'll work with the Vault sidecar to integrate that same variable into a pod.

## Using the Vault Sidecar

The Vault sidecar doesn't support injecting environment variables directly, because sidecar images can't share environment variables. If your pod does require environment variables, you need to generate a file that has a script that exports those variables. Here's an example:

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    run: watch-vault-env
  name: test-envvars-vault-watch
  namespace: my-ext-secret
  annotations:
    vault.hashicorp.com/agent-inject: "true"
    vault.hashicorp.com/log-level: trace
    vault.hashicorp.com/role: extsecret
    vault.hashicorp.com/tls-skip-verify: "true"
    vault.hashicorp.com/agent-inject-secret-myenv: 'secret/data/extsecret/
config'
    vault.hashicorp.com/agent-inject-template-myenv: |
      {{- with secret "secret/data/extsecret/config" -}}
        export MY_SECRET_PASSWORD="{{ index .Data "some-password" }}"
      {{- end -}}
spec:
  containers:
    - image: ubuntu:22.04
      name: test
      resources: {}
      command:
        - bash
        - -c
        - 'echo "sleeping 5 seconds"; sleep 5;source /vault/secrets/myenv ; env | grep MY_SECRET_PASSWORD'
  dnsPolicy: ClusterFirst
  restartPolicy: Never
  serviceAccountName: ext-secret-vault
  serviceAccount: ext-secret-vault
```

This pod definition looks almost identical to the volume-based Vault integration we created before. The main difference is that we are running a command in our container to generate the environment variables from the template we created in our annotations. While this approach will work, it means updating your manifests so that they are looking for a specific file. This approach breaks accepted patterns for manifest reusability and will make it difficult to use externally generated images.

The issues with trying to generate environment variables with an external secrets manager reinforces that this is an antipattern that should be avoided. Next, we'll look at using the Kubernetes API directly for retrieving Secrets.

## Using the Kubernetes Secrets API

So far, we have focussed on abstractions for interacting with APIs so that our workloads do not need to know about the Kubernetes API. For most workloads, this makes sense as secret metadata is often static. For instance, your application may need to talk to a database whose credentials may change but the fact that you need credentials won't. What if you need something more dynamic? If you're building a system that services other systems, this might be the case. In this section, we'll walk through why you would call the Secrets API directly from your pod.

The first question you might be asking is “why?” With the multiple options for abstraction on top of the Kubernetes Secrets API, why would you want to call the Kubernetes API directly? There have been some interesting trends that will likely make this more of a reality:

- **More monoliths:** In recent months there's been a trend of re-examining if microservices are the right architecture for most systems. One of the most visible moves to monoliths was Amazon's Prime Video Quality of Service (QoS) going from Lambda to a monolith. We discuss the trade-offs between these approaches when we talk about Istio in *Chapter 17, Building and Deploying Applications on Istio*. If you were to build a monolith, you may need to provide more flexibility in how you access your Secrets. Static metadata definitions may not be adequate.
- **Kubernetes as a data center API:** It turns out that Kubernetes' API is simple enough to be easily adapted to multiple use cases, yet powerful enough to be scaled out. A great example is the KubeVirt (<https://kubevirt.io/>) project that lets your Kubernetes cluster manage and deploy virtual machines. More workloads are using **custom resource definitions (CRDs)** to store configuration information, and since you'd never keep a secret in a CRD, you may need to interact with the Secrets API to get access to your secret data.
- **Platform engineering:** More teams are moving to the idea of creating an **internal developer platform (IDP)** that provides a one-stop-shop for self-service access to services such as Kubernetes. If your IDP is built on Kubernetes, you'll likely need to interact with the Secrets API.



This book spends quite a bit of time on identity in Kubernetes and will often refer to IdPs. The case of the “d” is important because an **IdP** is an **Identity Provider**, while an **IDP** is an **Internal Developer Portal**.

With these points in mind, you may find yourself needing to interact directly with the Kubernetes API to retrieve `Secrets`. The good news is that you're not really doing anything different for the `Secrets` API than any other API. Most of the Kubernetes client SDKs even handle base64 decoding the data from `Secret` objects.

Since this isn't a programming book, and there are many ways to interact with the API, we're not going to dive into any SDK specifics. We're going to cover some specific guidance:

- **Use Your Pod's Identity:** Just like interacting with external vaults, use your Pod's own identity when interacting with the API server. Don't use a hardcoded Secret.
- **Use an SDK:** This is good general advice. Yes, you can use the Kubernetes API via a direct RESTful call, but let the SDK do the work for you. It will make life easier and lead to fewer security issues (who hasn't accidentally logged a token while testing an HTTPS call? I mean, besides me).
- **Store Metadata in CRDs:** Any situation where you're going to describe a secret should be done in a CRD. This gives you the benefit of having a schema language that you can use to generate your own SDKs.

While it is more difficult to interact with the `Secrets` API than to use Kubernetes' multiple abstractions for interacting with secrets, it does provide tremendous flexibility that is not available to the more common abstractions. Next, we'll look at whether interacting directly with our vault's API can provide the same benefits.

## Using the Vault API

Every vault service has an API. That's how the sidecars that integrate with them interact with their vaults so that they can inject secrets into your workloads. In the previous section, we walked through the advantages of directly calling the Kubernetes `Secrets` API. Does the same logic apply to the Vault API? The reasoning for calling the Vault API, or any secrets management API, directly is the same as calling the `Secrets` API.

What are the disadvantages of calling the Vault API directly? The main drawback is that you are now tightly binding your workload to a specific vendor or project. One of the benefits of the Kubernetes API is that it is relatively consistent across implementations. While this isn't always true, at least for the `Secret` API it is.

There is unfortunately no standard API for secrets, nor is there likely to ever be one. HashiCorp, AWS, Microsoft, Google, VMWare, etc. all have their own ideas as to how secrets should be managed and there isn't much incentive to create a standard. There's also no standard for integrating language bindings. For instance, the database world often has common standards for integrating into programming languages such as the JDBC standard in Java. It would be great for Kubernetes to make the `Secrets` API pluggable, but that will never happen. The complexities at both the technical and non-technical levels are just too high for Kubernetes to own such an undertaking.

With that said, the same recommendations that were made for using the Kubernetes `Secrets` API should be followed when integrating your workloads directly into the Vault API. Make sure that you're using a language SDK and rely on your workload's identity.

## Summary

This chapter walked through multiple aspects of secrets management. We began by discussing the difference between secret data and more generic configuration data. We considered why Kubernetes stores and represents Secret objects as base64-encoded text, and why you shouldn't store secret data in git. There was also a discussion on threat modelling secret data in Kubernetes clusters. Next, we then walked through various ways to store and manage secret data including Secret objects, external vaults, Sealed Secrets, and hybrid approaches. Finally, we walked through integrating your secrets into your workloads via volume mounts, environment variables, and directly with APIs.

Having finished this chapter, you should now have enough information and examples to build your own secrets management strategy for your clusters.

In the next chapter, we are going to begin focusing on multi-tenancy with virtual clusters.

## Questions

1. Compliance and security are the same thing.
  - a. True
  - b. False
2. Base64 is a type of encryption.
  - a. True
  - b. False
3. The best way to authenticate to an external vault is:
  - a. Using a password
  - b. Using your Pod's identity
  - c. Using a password that changes every 3 months
4. Which integration type is automatically updated in a pod when a secret is updated?
  - a. A Secret with an environment variable
  - b. An external vault with an environment variable
  - c. An external vault with a volume
  - d. A Secret with a volume
5. When discussing IT security, what does CIA stand for?
  - a. Confidentiality, Integrity, Availability
  - b. Central Intelligence Agency
  - c. Confidentiality, Interesting, Availability
  - d. Culinary Institute of America

## Answers

1. b - False
2. b - False
3. c - Using your Pod's identity
4. d - A Secret with a volume
5. a - Confidentiality, Integrity, Availability



# 9

# Building Multitenant Clusters with vClusters

We've alluded to multitenancy in previous chapters, but this is the first chapter where we will focus on the challenges of multitenancy in Kubernetes and how to approach them with a relatively new technology called "virtual clusters." In this chapter, we'll explore the use cases for virtual clusters, how they're implemented, how to deploy them in an automated way, and how to interact with external services with your Pod's identity. We'll finish the chapter by building and deploying a self-service multitenant portal for Kubernetes.

In this chapter, we'll cover:

- The benefits and challenges of multitenancy
- Using vClusters for tenants
- Building a multitenant cluster with self service

## Technical requirements

This chapter will involve a larger workload than previous chapters, so a more powerful cluster will be needed. This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 8 GB of RAM, though 16 GB is suggested
- Scripts from the `chapter9` folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Getting Help

We do our best to test everything, but there are sometimes half a dozen systems or more in our integration labs. Given the fluid nature of technology, sometimes things that work in our environment don't work in yours. Don't worry, we're here to help! Open an issue on our GitHub repo at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/issues> and we'll be happy to help you out!

## The Benefits and Challenges of Multitenancy

Before we dive into virtual clusters and the vCluster project, let's first explore what makes multitenant Kubernetes valuable and so difficult to implement. So far, we've alluded to challenges with multitenancy, but our focus has been on configuring and building a single cluster. This is the first chapter where we're going to directly address multitenancy and how to implement it. The first topic we will explore is what multitenant Kubernetes is, and why you should consider using it.

### Exploring the Benefits of Multitenancy

Kubernetes orchestrates the allocation of resources for workloads through an API and a database. These workloads are typically comprised of Linux processes that require specific computing resources and memory. In the initial five to six years of Kubernetes' evolution, a prevailing trend was to have a dedicated cluster for each "application." It's important to note that when we say "application," it could refer to a single monolithic application, a collection of microservices, or several interrelated monolithic applications. This approach is notably inefficient and results in the proliferation of management complexities and potential wasted resources. To understand this better, let's examine all of the elements within a single cluster:

- **etcd Database:** You should always have an odd number of etcd instances; at least three instances of etcd are needed to maintain high availability.
- **Control Plane Nodes:** Similar to etcd, you'll want at least two control plane nodes, but more likely three for high availability.
- **Worker Nodes:** You'll want a minimum of two nodes, regardless of the load you're putting on your infrastructure.

Your control plane requires resources, so even though most Kubernetes distributions don't require dedicated control plane nodes anymore, it's still additional components to manage. Looking at your worker nodes, how much utilization are the nodes using? If you have a heavily used system, you will get the most out of the hardware, but are all your applications always that heavily utilized? Allowing multiple "applications" to use a single infrastructure can vastly reduce your hardware utilization, requiring fewer clusters, whether you're running on pre-paid infrastructure or pay-as-you-go infrastructure. Over-provisioning resources will increase power, cooling, rack space, etc., all of which will add additional costs. Fewer servers also means reducing the maintenance requirements for hardware, further reducing costs.

In addition to hardware costs, there's a human cost to the cluster-per-application approach. Kubernetes skills are very expensive and difficult to find in the market. It's probably why you're reading this book! If each application group needs to maintain its own Kubernetes expertise, that means duplicating skills that are difficult to obtain, also increasing costs. By pooling infrastructure resources, it becomes possible to establish a centralized team responsible for overseeing Kubernetes deployments, eliminating the necessity for other teams, whose primary focus lies outside infrastructure development, to duplicate these skills.

There are major security benefits to multitenancy as well. With common infrastructure, it's easier to centralize enforcement of security requirements. Taking authentication as an example, if every application gets its own cluster, how will you enforce common authentication requirements? How will you onboard new clusters in an automated way? If you're using OIDC, are you going to integrate a provider for each cluster?

Another example of the benefits of centralized infrastructure is secret management. If you have a centralized Vault deployment, do you want to integrate a new cluster for each application? In *Chapter 8*, we integrated a cluster with Vault; if you have massive cluster sprawl, the same integration needs to be done to each individual cluster – how will that be automated and managed?

Moving to a multitenant architecture reduces your long-term costs by reducing the amount of infrastructure needed to run your workload. It also cuts down on the number of administrators needed to manage infrastructure and makes it easier to centralize security and policy enforcement.

While multitenancy provides a significant advantage, it does come with some challenges. Next, we'll explore the challenges of implementing multitenant Kubernetes clusters.

## The Challenges of Multitenant Kubernetes

We explored the benefits of multitenant Kubernetes, and while the adoption is growing for multitenancy, why isn't it as common as the cluster-per-application approach?

Creating a multitenant Kubernetes cluster requires several considerations for security, usability, and management. Implementing solutions for these challenges is often very implementation-specific and requires integration with third-party tools, which are outside the scope of most distributions.

Most enterprises add an additional layer of complexity based on their management silos. Application owners are judged based on their own criteria and by their own management. Anyone whose pay is dependent on certain criteria will want to make sure they have as much control of those criteria as possible, even if it's for the wrong reasons. This silo effect can have an adversely negative impact on any centralization effort that doesn't afford appropriate control to application owners. Since these silos are unique to each enterprise, it's impossible for a single distribution to account for them in a way that is easily marketable. Rather than deal with the additional complexities, it's much easier for a vendor to market a cluster-per-application approach.

With the fact that there are few multitenant Kubernetes distributions on the market, the next question becomes "What are the challenges of making a generic Kubernetes cluster multitenant?"

We're going to break down the answers to this question by impact:

- **Security:** Most containers are just Linux processes. We'll cover this in more detail in *Chapter 12, Node Security with Gatekeeper*. What's important to understand for now is that there is very little security that separates processes on a host. If you're running processes from multiple applications, you want to make sure that a breakout doesn't impact other processes.
- **Container breakouts:** While this is important for any cluster, it's a necessity in multitenant clusters. We will cover securing our container runtimes in *Chapter 13, KubeArmor Securing Your Runtime*.
- **Impact:** Any issues with centralized infrastructure will have adverse impacts on multiple applications. This is often referred to as "blast radius." If there's an upgrade that goes wrong or fails, who's impacted? If there's a container breakout, who's impacted?
- **Resource Bottlenecks:** While it's true that a centralized infrastructure gets better utilization of resources, it can also create bottlenecks. How quickly can you onboard a new tenant? How much control do application owners have in their own tenants? How difficult is it to grant or revoke access? If your multitenant solution can't keep up with application owners, application owners will take on the infrastructure themselves. This will lead to wasted resources, configuration drift, and difficulty reporting and auditing all of the clusters.
- **Restrictions:** A centralized platform that is overly restrictive will result in application owners looking to either maintain their own infrastructure or outsource their infrastructure to third-party solutions. This is one of the most common issues with any centralized service that can be best illustrated by the continuous rise and fall of **Platform as a Service (PaaS)** implementations that fail to provide the flexibility needed for application workloads.

While these issues can be applied to any centralized service, they do have some unique impacts on Kubernetes. For instance, if each application gets its own namespace, how can an application properly subdivide that namespace for different logical functions? Should you grant multiple namespace per application?

Another major impact on Kubernetes cluster designs is the deployment and management of **Custom Resource Definitions (CRDs)**. CRDs are cluster-level objects, and running multiple versions is nearly impossible in the same cluster; as we have pointed out in previous chapters, CRDs are growing in popularity as a way of storing configuration data. Multitenant clusters may have version conflicts that need to be managed.

Many of these challenges will be addressed in later chapters, but in this chapter, we're going to focus on two aspects:

- **Tenant Boundaries:** What is the scope of a tenant? How much control within the boundary does the tenant have?
- **Self-Service:** How does a centralized Kubernetes service interact with users?

Both aspects will be addressed by adding two components to our clusters. OpenUnison has already been introduced to handle authentication and will be extended for its self-service capabilities with namespace as a Service. The other external system will be vCluster, from Loft Labs.

Since we have already used OpenUnison to demonstrate namespace as a Service, we can move on to the additional challenges, like CRD versioning issues in the next section using the vCluster project.

## Using vClusters for Tenants

In the *KinD* chapter, we explained that KinD is nested in Docker to provide us with a full Kubernetes cluster. We compared this to nesting dolls, where components are embedded in other components, which can cause confusion to users who are newer to containers and Kubernetes. vCluster is a similar concept – it creates a virtual cluster in the main host clusters, and while it does appear to be a standard Kubernetes cluster, it is nested within the host clusters. Keep this in mind as you are reading the rest of the chapter.

In the previous section, we walked through the benefits and challenges of multitenancy and how those challenges impact Kubernetes. In this section, we’re going to introduce the vCluster project from Loft Labs, which allows you to run a Kubernetes control plane inside of an unprivileged namespace. This allows each tenant to get their own “virtual” Kubernetes infrastructure that they can have complete control over without impacting other tenants’ own implementation or other workloads in the “main cluster.”

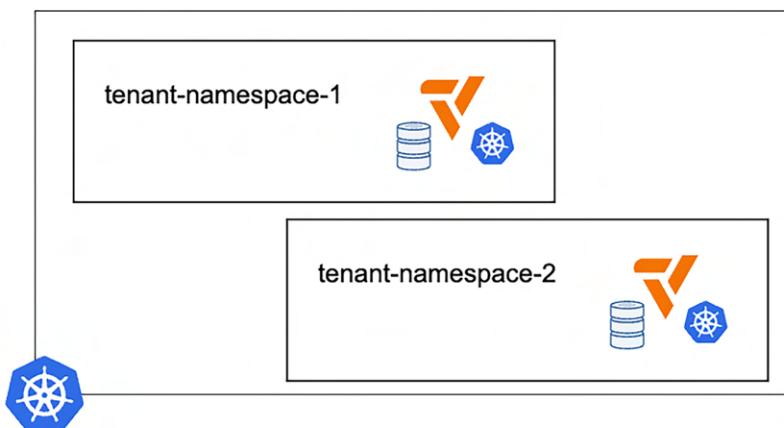


Figure 9.1: Logical layout of a vCluster

In the above diagram, each tenant gets their own namespace, which runs a vCluster. The vCluster is a combination of three components:

- **Database:** Somewhere that can store the vCluster’s internal information. This may be etcd or a relational database, depending on which cluster type you deploy vCluster with.
- **API Server:** The vCluster includes its own API server for its pods to interact with. This API server is backed by the database managed by the vCluster.
- **Synchronization Engine:** While a vCluster has its own API server, the pods are all run in the host cluster. To achieve this, the vCluster synchronizes certain objects between the host and vCluster. We’ll cover this in greater detail next.

The benefit of the vCluster's approach is that from the Pod's perspective, it's working within a private cluster while it's really running in a main host cluster. The tenant can divide its own cluster into whatever namespace suit it and deploy CRDs, or operators, as needed.

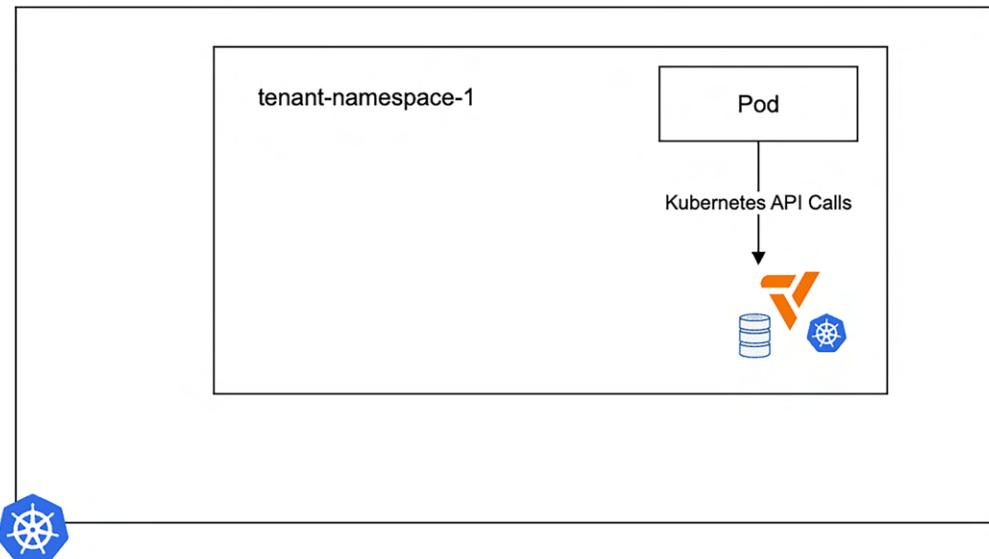


Figure 9.2: Pod's perspective of a vCluster

In the above diagram, we see that the pod is deployed into our tenant's namespace, but instead of communicating with the host cluster's API server, it's communicating with the vCluster's API server. This is possible because the pod definition that gets synchronized from the vCluster into the host cluster has its environment variables and DNS overwritten to tell the pod that the host `kubernetes.default.svc` points to the vCluster, not the host cluster's API server.

Since the pod runs in the host cluster, and the vCluster runs in the host cluster, all of the pods are subject to the ResourceQuotas put in place on the namespace. This means that any pod deployed to a vCluster is bound by the same rules as a pod deployed directly into a namespace including any restrictions created by quotas, policies, or other admission controllers. In *Chapter 11, Extending Security Using Open Policy Agent*, you'll learn about using admission controllers to enforce policies across your cluster. Since the pod is running in the host cluster, you only need to apply those policies to your host. This vastly simplifies our security implementation because, now, tenants can be given `cluster-admin` access to their virtual clusters without compromising the security of the host cluster.

Another important note is that because the host cluster is responsible for running your pod, it's also responsible for all of the vClusters Ingress traffic. You do not have to redeploy your Ingress controller on each vCluster – they share the host Ingress controller, reducing the need for maintaining additional Ingress deployments or creating multiple wildcard domains for each vCluster.

Now that we have a basic understanding of what a vCluster is and how it helps to address some of the challenges of multitenancy in Kubernetes, the next step is to deploy a vCluster and see how the internals work.



As shown in the output above, there are two pods running in the tenant1 namespace: CoreDNS and our vCluster. If we look at the services in our namespace, you will see a list of services similar to the following:

\$ kubectl get svc -n tenant1				
NAME		TYPE	CLUSTER-IP	EXTERNAL-IP
PORT(S)	AGE			
kube-dns-x-kube-system-x-myvcluster	7m18s	ClusterIP	10.96.142.24	<none>
53/UDP,53/TCP,9153/TCP				
myvcluster		NodePort	10.96.237.247	<none>
443:32489/TCP,10250:31188/TCP	7m45s			
myvcluster-headless		ClusterIP	None	<none>
443/TCP	7m45s			
myvcluster-node-cluster01-worker		ClusterIP	10.96.209.122	<none>
10250/TCP	7m18s			

There are several services set up to point to our vCluster's API server and DNS server that provide access to the vCluster, making it logically appear as a "full" standard cluster.

Now let's connect to our vCluster and deploy a pod. In the chapter9/simple directory, we have a pod manifest that we will use for our example. First, we will connect to the cluster and deploy the example pod using kubectl in the chapter9/simple directory:

```
$ vcluster connect myvcluster -n tenant1
$ cd chapter9/simple
$ kubectl create -f ./virtual-pod.yaml
$ kubectl logs -f virtual-pod
Wed Sep 20 17:50:03 UTC 2023
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=virtual-pod
PWD=/
HOME=/root
KUBERNETES_PORT_443_TCP=tcp://10.96.237.247:443
SHLVL=1
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.96.237.247
KUBERNETES_SERVICE_HOST=10.96.237.247
KUBERNETES_PORT=tcp://10.96.237.247:443
KUBERNETES_PORT_443_TCP_PORT=443
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
_=/usr/bin/env
search default.svc.cluster.local svc.cluster.local cluster.local
nameserver 10.96.142.24
```

Notice that the Pod's environment variables use 10.96.237.247 as the IP address for the API server; this is the ClusterIP from the mycluster Service that's running on the host. Also, the nameserver is 10.96.142.24, which is the ClusterIP for our vCluster's kube-dns Service in the host. As far as the pod is concerned, it thinks it is running inside of the vCluster. It doesn't know anything about the host cluster. Next, disconnect from the vCluster and take a look at the pods in our tenant1 namespace on the host cluster:

```
$ vcluster disconnect
$ kubectl get pods -n tenant1
NAME                                         READY   STATUS    RE-
STARTS   AGE
coredns-864d4658cb-mdcx5-x-kube-system-x-myvcluster   1/1     Running   0
22m
myvcluster-0                                     2/2     Running   0
22m
virtual-pod-x-default-x-myvcluster               1/1     Running   0
7m11s
```

Notice that your vCluster's pod is running in your host cluster. The Pod's name on the host includes both the name of the pod and the namespace from the vCluster. Let's take a look at the pod definition. We're not going to put all of the output here because it would take up multiple pages. What we want to point out is that in addition to our original definition, the pod includes a hard-coded env section:

```
env:
  - name: KUBERNETES_PORT
    value: tcp://10.96.237.247:443
  - name: KUBERNETES_PORT_443_TCP
    value: tcp://10.96.237.247:443
  - name: KUBERNETES_PORT_443_TCP_ADDR
    value: 10.96.237.247
  - name: KUBERNETES_PORT_443_TCP_PORT
    value: "443"
  - name: KUBERNETES_PORT_443_TCP_PROTO
    value: tcp
  - name: KUBERNETES_SERVICE_HOST
    value: 10.96.237.247
  - name: KUBERNETES_SERVICE_PORT
    value: "443"
  - name: KUBERNETES_SERVICE_PORT_HTTPS
    value: "443"
```

It also includes its own hostAliases:

```
hostAliases:
  - hostnames:
```

```
- kubernetes
- kubernetes.default
- kubernetes.default.svc
ip: 10.96.237.247
```

So, while the Pod is running in our host cluster, all of the things that tell the pod where it's running are pointing to our vCluster.

In this section, we launched our first vCluster and a pod in that vCluster to see how it gets mutated to run in our host cluster. In the next section, we're going to look at how we can access our vCluster with an eye on the same enterprise security we're required to use in our host cluster.

## Securely Accessing vClusters

In the previous section, we deployed a simple vCluster and accessed the vCluster using the `vcluster connect` command. This command first creates a port-forward to the vCluster's API server Service and then adds a context with a master certificate to our `kubectl` configuration file, which is similar to our KinD cluster.

We spent much of *Chapter 6, Integrating Authentication into Your Cluster*, walking through why this is an anti-pattern, and those reasons still apply to vClusters. You're still going to need to integrate enterprise authentication into your vCluster. Let's look at two approaches:

- **Decentralized:** You can leave authentication as an exercise to the cluster owner. This negates many of the advantages of multitenancy and will require that each cluster is treated as a new integration into your enterprise's identity system.
- **Centralized:** If you host an **identity provider (IdP)** in your host cluster, you can tie each vCluster to that IdP instead of directly to the centralized identity store. In addition to providing centralized authentication, this approach makes it easier to automate the onboarding of new vClusters and limits the amount of secret information, such as credentials, that needs to be stored in the vCluster.

The choice is clear when working in a multitenant environment; your host cluster should also host central authentication.

The next issue to understand regarding vCluster access is the network path to your vCluster. The `vcluster` command creates a local port-forward to your API server. This means that every time a user wants to use their API server, they'll need to set up a port-forward to their API server. This isn't a great **user experience (UX)** and can be error-prone. It would be better to set up a direct connection to our vCluster's API server, as we would for any standard Kubernetes cluster. The challenge with setting up direct network access to our vCluster's API server is that while it's a `NodePort`, nodes are rarely exposed directly to the network. They usually sit behind a load balancer and rely on `Ingress` controllers to provide access to cluster resources.

The answer is to use the application infrastructure our host cluster already provides for our vClusters.

In *Chapter 6, Integrating Authentication into Your Cluster*, we talked about using impersonating proxies with cloud-managed clusters. The same scenario can be applied to vClusters. While you can configure k3s to use OIDC for authentication, using an impersonating proxy vastly simplifies the network management because we're not creating new load balancers or infrastructure to support our vClusters.

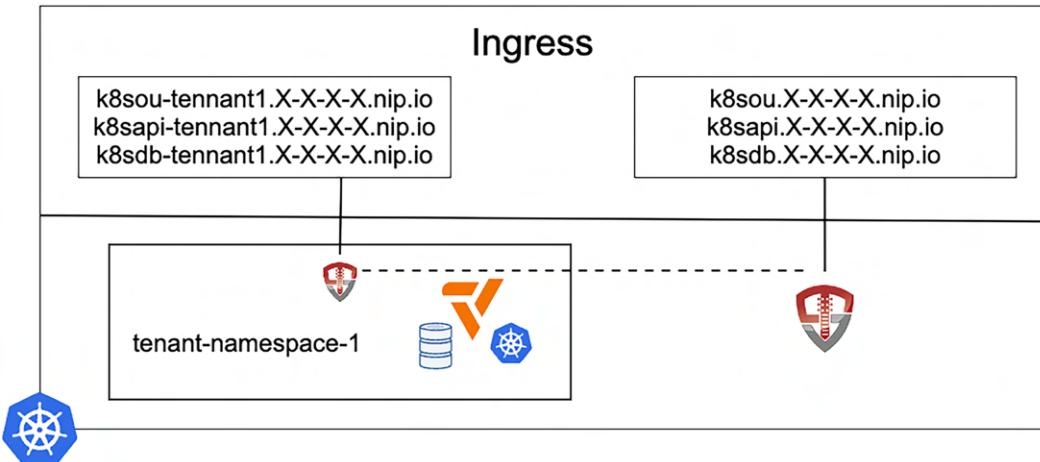


Figure 9.3: vCluster with authentication

In the above diagram, we can see how the networking and authentication come together in our host cluster. The host cluster will have an OpenUnison that authenticates users to our Active Directory. Our vCluster will have its own OpenUnison with a trust established with our host cluster's OpenUnison. The vCluster will use kube-oidc-proxy to translate the authentication tokens from OpenUnison into impersonation headers to our vCluster's API server. This approach gives us a central authentication and networking system, while also making it easier for vCluster owners to incorporate their own management applications without having to get the host cluster team involved. Local cluster management applications such as ArgoCD and Grafana can all be integrated into the vCluster's OpenUnison instead of the host cluster's OpenUnison.

To show our setup in action, the first thing we need to do is update our vCluster so that it will synchronize Ingress objects from our vCluster into our host cluster, using the vcluster tool. In the chapter/host directory, we have an updated values file called `vcluster-values.yaml`; we will use this values file to upgrade the vCluster in the `tenant1` namespace:

```
$ cd chapter9/host
$ vcluster create myvcluster --upgrade -f ./vcluster-values.yaml -n tenant1
```

This command will update our vCluster to synchronize the Ingress objects we create in our vCluster into our host cluster. Next, we'll need OpenUnison running in our host cluster:

```
$ vcluster disconnect
$ ./deploy_openunison_imp_implementation.sh
```

Before deploying anything, we want to make sure that we're running against the host, not our vCluster. The script we just ran is similar to what we ran in *Chapter 6, Integrating Authentication into Your Cluster*; it will deploy our “Active Directory” and OpenUnison to the vCluster. Once OpenUnison has been deployed, the last step is to run the satellite deployment process for OpenUnison:

```
$ vcluster disconnect
$ ./deploy_openunison_vcluster.sh
```

This script is like the host's deployment script with some key differences:

- The values for our OpenUnison do not contain any authentication information.
- The values for our OpenUnison have a different cluster name from our host cluster.
- Instead of running `ouctl install-auth-portal`, the script runs `ouctl install-satellite`, which sets up OpenUnison to use OIDC between the satellite cluster and the host cluster. This command creates the `oidc` section of the `values.yaml` file for us.

Once the script has completed executing, you can log in to OpenUnison just as you did in *Chapter 6*. In your browser, go to `https://k8sou.apps.X-X-X-X.nip.io`, where X-X-X-X is the IP address of your cluster, but with dashes instead of dots. Since our cluster is at 192.168.2.82, we use `https://k8sou.apps.192-168-2-82.nip.io/`. To log in, use the user `mmosley` with the password `start123`.

Once you're logged in, you'll see that there is now a tree with options for **Host Cluster** and **tenant1**. You can click on **tenant1**, then click on the **tenant1 Tokens** badge.

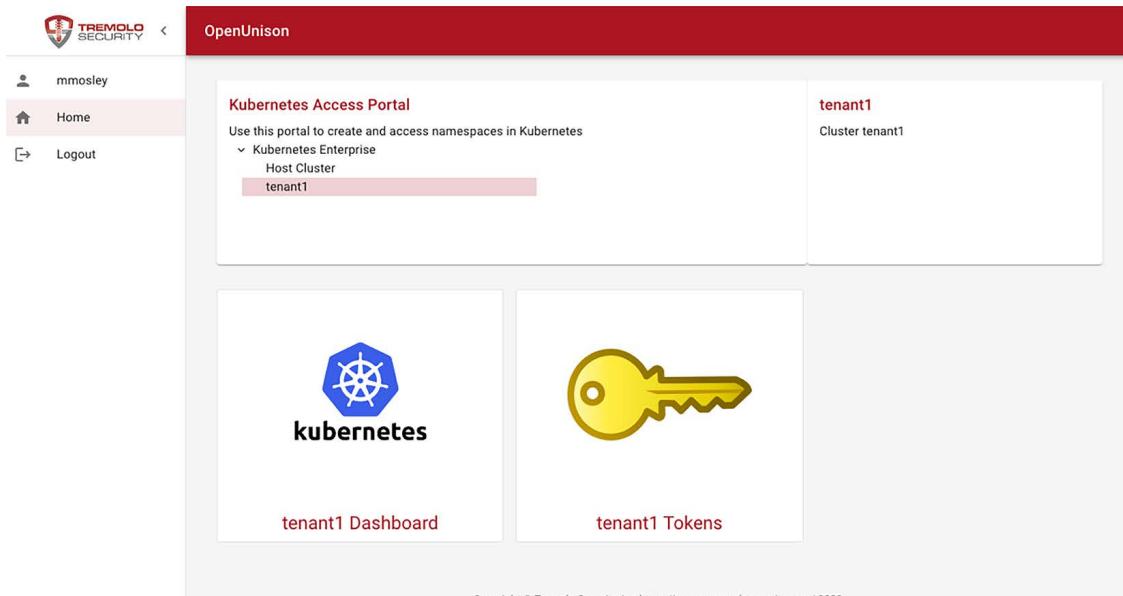


Figure 9.4: OpenUnison portal page

Once the new page loads, you can grab your `kubectl` configuration and paste it into your terminal:

```
$ TMP_CERT=New-TemporaryFile ; -----BEGIN CERTIFICATE-----`nMIERzCCq1bKLFOyUnitHTWpyvWpTDVOS44w8/AqT3YMCiyPKTUwME6BkEo6Sql5yUWVBSWpKUGP
-----BEGIN CERTIFICATE-----
MIERzCCAY+gAwIBAgIGAYq9dPK0MA0GCSqGSIb3DQEBCwUAMIGKMS8wLQYDVQJD
DCzrOHNvds10ZWhbnQxLmFwcHMuMTkyLTE2OC0yLTgyLm5pcC5pbzETMBEGA1UE
CwvKS3V1ZXJuZXrlc2E0MAGA1UECwgtX1PcmcsKEARBgNVAcNCk15IEnsdXNO
ZXIxCTAHBgNVBAgMADESMBAGA1UEBhJTX1b3VudHJ5MB4XDITzMDkyMjElMDQz
MloXDTI0MDkyMTE1NTQzMl0wgYoxLzAtBgNVBAMMJms4c291LXRlbmFuDEuYXbw
cy4xOTIMTY4lTItODtubmlwlmvMRMwEQYDVQQLDApLdWJlcm5lGVzMQ4wDAYD
VQQKDAvNeU9y2zTMBEGA1UEBwwkTxkgQ2x1c3RlcjEJNACGA1UECwAUHRIwEAyD
VQQGEw1NeUNvdW50cnwgEiMA0GCSqGSIb3DQEBAQUAA4IBDwAwggEKAoIBAQDa
jWdIZkbp7f6GtZ1spbe53I+H3DYKv9uQ7l1sQWjFU8DC17qwbSGSl2lZkXSp
YUHeVuX9vDogFLu+r5ptCjtQmoXanv+g2hyWgtkCnB5WT5UzLbxrzuMnUh/p1
SjNk48cpn5TLLhH2awpLntj7s52gZCObqfhtNxH/DxUoHkg/sSJaWDtMA35gIzX
BrSwD1/korO+EnRVW08mLtoqY+Cyi397/M9U3J2WXQ0Ydkrbal2q73GicRhfS6
p+nbelKunINnjjvD11MionW21scalNebD16CeNC8BruCwISKjUwO5gg9004yJ3rA
-----BEGIN CERTIFICATE-----
```

Figure 9.5: OpenUnison `kubectl` configuration generator

Depending on which client you’re running on, you can paste this command into a Windows or Linux/macOS terminal and start using your vCluster without having to distribute the `vcluster` CLI tool and while using your enterprise’s authentication requirements.

In this section, we looked at how to integrate enterprise authentication into our vClusters and how to provide consistent network access to our vClusters as well. In the next section, we’ll explore how to integrate our vClusters with external services, such as HashiCorp’s Vault.

## Accessing External Services from a vCluster

In the previous chapter, we integrated a HashiCorp Vault instance into our cluster. Our pods communicated with Vault using the tokens projected into our pods, allowing us to authenticate to Vault without a pre-shared key or token and using short-lived tokens. Relying on short-lived tokens reduces the risk that a compromised token can be used against your cluster.

The pod-based identity used with Vault gets more complex with vClusters because the keys used to create the Pod’s tokens are unique to the vCluster. Also, Vault needs to know about each vCluster in order to verify the projected tokens used in the vCluster.

If we are running our own Vault in our host cluster, we could automate the onboarding so that each new vCluster is registered with Vault as its own cluster. The challenge with this approach is that Vault is a complex system that’s often run by its own team with its own onboarding process. Adding a new vCluster in a way that works for the team that owns Vault may not be as simple as calling some APIs. Therefore, before we can implement a strategy for integrating our vClusters into Vault, we need to examine how vClusters handle identity.

A pod running in a vCluster has two distinct identities:

- **vCluster Identity:** The token that is projected into our pod from a vCluster is scoped to the API server for the vCluster. It was signed by a unique key that the host cluster has no knowledge of. It is associated with the ServiceAccount the pod runs as inside of the vCluster's API server.
- **Host Cluster Identity:** While the pod is defined on the vCluster, it's executed in the host cluster. This means that the security context of the pod will run and it requires a distinct identity from the vCluster. It will have its own name and its own signing keys.

If we inspect a pod from our vCluster as synchronized into our host cluster, we'll see that there's an annotation that contains a token in it:

```
vcluster.loft.sh/token-ejijuegk: >-
eyJhbGciOiJSUzI1NiIsImtpZCI6IkVOZDVhZnEzUzdtLXBSR2JUM3RJUkRHMOFqWkhzQV9K-
SkNZcm8yMHdNVUUifQ...
```

This token is injected into our pod via the `fieldPath` configuration later in the pod. This could be a security issue since anything that logs the Pod's creation, such as an audit log, can now leak a token. The vCluster project has a configuration to generate `Secret` objects in the host cluster for project tokens so that they're not in the pod manifest. Adding the following to our `values.yaml` file will fix this issue:

```
syncer:
  extraArgs:
    - --service-account-token-secrets=true
```

With that done, let's update our cluster and redeploy all the Pods:

```
$ vcluster disconnect
$ vcluster create myvcluster --upgrade -f ./vcluster-values-secrets.yaml -n
tenant1
$ kubectl delete pods --all --all-namespaces --force
```

In a moment, the pods inside of the vCluster will come back. Inspecting the Pod, we see that the token is no longer in the Pod's manifest but is now mounted to a `Secret` in the host. This is certainly an improvement, as audit systems are generally more discreet about logging the contents of `Secrets`. Next, let's inspect our token's claims.

If we inspect this token, we'll see some issues:

```
{
  "aud": [
    "https://kubernetes.default.svc.cluster.local"
  ],
  "exp": 2010881742,
  "iat": 1695521742,
  "iss": "https://kubernetes.default.svc.cluster.local",
```

```
"kubernetes.io": {
    "namespace": "openunison",
    "pod": {
        "name": "openunison-orchestra-d7bc468bc-qhpts",
        "uid": "fb6874f7-c01c-4ede-9062-ce5409509200"
    },
    "serviceaccount": {
        "name": "openunison-orchestra",
        "uid": "3d9c9147-d0e0-43f6-9e4d-a0d7db0c6b8c"
    }
},
"nbf": 1695521742,
"sub": "system:serviceaccount:openunison:openunison-orchestra"
}
```

The `exp` and `iat` claims are in bold because, when you translate this from Unix Epoch time to something a human can understand, this token is good from `Sunday, September 24, 2023 2:15:42 AM` until `Wednesday, September 21, 2033 2:15:42 AM`. That's a ten-year token! This ignores the fact that the token was configured in the pod to only be good for ten minutes. This is a known issue in vCluster. The good news is that the tokens themselves are projected, so when the pod they're projected into dies, the API server will no longer accept these tokens.

The issue of vCluster token length comes into play when accessing external services because this will be true of any token we generate, not just tokens for the vCluster API server. When we integrated our clusters into Vault in the previous chapter, we did so using our Pod's identity so that we could leverage shorter-lived tokens that aren't static and have well-defined expirations. A ten-year token is effectively a token with no expiration. The main mitigation is that we configured Vault to verify the status of the token before accepting it, so a token bound to a destroyed pod will be rejected by Vault.

The alternative to using the vCluster's injected identity is to leverage the host cluster's injected identity. This identity will be governed by the same rules as any other identity generated by the `TokenRequest` API in the host cluster. There are two issues with this approach:

- **vCluster disables host tokens:** In the host synced pod, `automountServiceAccountToken` is `false`. This is to prevent a collision between the vCluster and the host cluster because our pod shouldn't know the host cluster exists! We can get around this by creating a mutating webhook that will add a `TokenRequest` API projection in the host cluster that can be accessed by our pod.
- **Host tokens don't have vCluster namespaces:** When we do generate a host token for our synced pod, the namespace will be embedded in the name of the `ServiceAccount`, not as a claim in the token. This means that most external services' policy languages will not be able to accept policies based on the host token, but configured via a namespace without creating a new policy for each vCluster namespace.

These approaches both have benefits and drawbacks. The biggest benefit to using vCluster tokens is that you can easily create a policy that allows you to limit access to secrets based on namespaces inside of your vCluster without creating new policies for each namespace. The downside is the issues with vCluster tokens and the fact that you now need to onboard each individual vCluster into your Vault. Using host tokens better mitigates the issues with vCluster tokens, but you're not able to easily create generic policies for each vCluster in Vault.

In this section, we spent time understanding how vClusters manage pod identities and how those identities can be used to interact with external services, such as Vault. In the next section, we will spend time on what's needed to create a highly available vCluster and manage operations.

## **Creating and Operating High-Availability vClusters**

So far in this chapter, we've focused on the theory of how vClusters work, how to access a vCluster securely, and how vClusters handle pod identity to interact with external systems. In this section, we'll focus on how to deploy and manage vClusters for high availability. Much of the documentation and examples for vCluster focuses on vCluster as a development or testing tool. For the use cases discussed earlier in this chapter, we want to focus on creating vClusters that can run production workloads. The first part of building production-ready vClusters is to understand how to run a vCluster in a way that allows for failures or downtime of individual components without hampering the vCluster's ability to run.

### **Understanding vCluster High Availability**

Let's define the goal of a highly available vCluster and the gap between our current deployments and that goal. When you have a highly available vCluster, you want to make sure that:

- You can continue to interact with the API server during an upgrade or migration to another physical node in either the host cluster or the vCluster.
- If there's a catastrophic issue, you can restore from a backup.

When running a vCluster, the first point about being able to interact with an API server becomes clear while upgrading the host cluster's nodes. During that upgrade process, you want your API server to be able to continue to run. You want to be able to continue syncing objects from your virtual API server into your host; you also want pods that interact with the API server to still be able to do so during a physical host upgrade. For instance, if you're using OpenUnison, then you want it to be able to create session objects so users can interact with their vClusters while host cluster operations are happening.

The second point about disaster recovery is also important. We hope to never need it, but what happens if we've irrevocably broken our vCluster? Can we restore back to a point that we know was functional?

The first aspect of understanding how to run a highly available vCluster is that it will need multiple instances of pods that run the API server, syncer, and CoreDNS. If we look at our `tenant1` namespace, we'll see that our vCluster has one pod that is associated with a `StatefulSet` that hosts the vCluster's API server and syncer. There is also a pod that is synced from inside the vCluster for CoreDNS. We'd want there to be at least two (better if three) instances of each of these pods so that we can tell our API server to use `PodDisruptionBudget` to make sure we have a minimum number of instances running so that one can be brought down for whatever event is occurring.

The second aspect to understand is how vCluster manages data. Our current deployment uses k3s, which uses a local SQLite database with data stored on a `PersistentVolume`. This works well for development, but for a production cluster, we want each component of our vCluster to be working off the same data. For k3s-based vClusters, this means using one of the supported relational databases or etcd. We could deploy etcd, but a relational database is generally easier to manage. We're going to deploy our database in-cluster, but it wouldn't be unusual to use an external database as well. In our exercises, we'll use MySQL. We won't worry about building a highly available database for our examples, since each database has its own mechanisms for high availability. If this were a production deployment though, you'd want to make sure that your database is built using the project's recommended HA deployment and that you have a regular backup and recovery plan in place. With that said, let's start by tearing down our current cluster and creating a new one:

```
$ kind delete cluster -n cluster01  
$ cd chapter2/HAdemo  
$ ./create-multinode.sh
```

Wait for the new multi-node cluster to finish launching. Once it's running, deploy MySQL:

```
$ cd ../../chapter9/ha  
$ ./deploy_mysql.sh
```

If the `deploy_mysql.sh` script fails with “Can't connect to local MySQL server through socket,” wait a moment and rerun it. It's safe to rerun. This script:

1. Deploys the cert-manager project with self-signed `ClusterIssuers`.
2. Creates TLS keypairs for MySQL.
3. Installs MySQL as a `StatefulSet` and configures it to accept TLS authentication.
4. Creates a database for our cluster and a user that's configured to authenticate via TLS.

With MySQL deployed and configured for TLS authentication, we'll next create the `tenant` namespace and a certificate that will map to our database user:

```
$ kubectl create -f ./vcluster-tenant1.yaml
```

Finally, we can deploy our vCluster:

```
$ vcluster create tenant1 --distro k3s --upgrade -f ./vcluster-ha-tenant1-  
values.yaml -n tenant1 --connect=false
```

It will take a few minutes, but you'll have four pods in the `tenant1` namespace:

NAME	READY	STATUS	RESTARTS
AGE			
coredns-6ccdd78696-r5kmd-x-kube-system-x-tenant1	1/1	Running	0
88s			
coredns-6ccdd78696-rw9lv-x-kube-system-x-tenant1	1/1	Running	0
88s			
tenant1-0	2/2	Running	0
tenant1-1	0	2m16s	

We can now leverage `PodDisruptionBudget` to tell Kubernetes to keep one of the vCluster pods running during upgrades.

Speaking of upgrades, the next question is how to upgrade our vCluster. Now that we have a highly available vCluster, we can look to upgrade our vCluster to a new version.

## Upgrading vClusters

It's important that you know how to upgrade your vClusters. You'll want to make sure that your vCluster and host cluster don't drift too far apart. While the pods that are synced into your host cluster will communicate with your vCluster's API server, any impact on the synchronized pods (and other synchronized objects) could impact your workloads.

Given the importance of staying up to date, it is great to report that upgrading a vCluster is incredibly easy. It's important to remember that vCluster orchestrates the clusters and synchronizes objects, but the clusters themselves are managed by their own implementation. In our deployments, we're using k3s, which will upgrade its data storage in the database when the new pods are deployed. Since the `vcluster create` command is a wrapper for Helm, all we need to do is update our values with the new image and redeploy:

```
$ kubectl get pod tenant1-0 -n tenant1 -o json | jq -r '.spec.initContain-  
ers[0].image'  
rancher/k3s:v1.29.5-k3s1  
$ vcluster create tenant1 --upgrade -f ./vcluster-ha-tenant1-values-upgrade.  
yaml -n tenant1 --connect=false
```

This command upgrades our vCluster to use a k3s 1.30 image, which is just running a Helm upgrade on our installed chart. You're leveraging the power of Kubernetes to simplify upgrades! Once it's done running, you can check that the pods are now running k3s 1.30:

```
$ kubectl get pod tenant1-0 -n tenant1 -o json | jq -r '.spec.initContain-  
ers[0].image'  
rancher/k3s:v1.30.1-k3s1
```

We've covered creating highly available clusters and how to upgrade our vClusters. This is enough to embark on building a multitenant cluster. In the next section, we'll integrate what we have learned to build out a multitenant cluster where each tenant gets their own vCluster.

## Building a Multitenant Cluster with Self Service

In the previous sections, we explored how multitenancy works, how the vCluster project helps to address multitenancy challenges, and how to configure a vCluster with secure access and high availability. Each of these individual components was addressed as a separate component. The next question is how to integrate all these components into a single service. In this section, we'll walk through creating a self-service platform for a multitenant cluster.

One of the most important aspects of multitenancy is repeatability. Can you create each tenant the same way consistently? In addition to making sure that your approach is repeatable, what's the amount of work that a customer needs to go through to get a new tenant? Remember that this book has a focus on enterprise, and enterprises almost always have compliance requirements. You also need to consider how to integrate your compliance requirements into the onboarding process.

The combination of needing repeatability and compliance often leads to the need for a self-service portal for onboarding new tenants. Creating a self-service portal has become the focus of many projects, often as part of a “Platform Engineering” initiative. We’re going to build our self-service platform from OpenUnison’s namespace as a Service portal. Using OpenUnison as our starting point, let’s us focus on how the components will integrate, rather than diving into the specifics of writing the code for the integrations. This multitenant self-service onboarding portal will serve as a starting point that we’ll add to as we explore more aspects of multitenancy through this book.

We’re going to approach our multitenant cluster first by defining our requirements, then analyze how each of those requirements will be fulfilled, and finally, we’ll roll out our cluster and portal. Once we’re done with this section, you’ll have the start of a multitenant platform you can build from.

## Analyzing Requirements

Our requirements for each individual tenant will be like requirements for a physical cluster. We’re going to want to:

- **Isolate tenants by authorization:** Who should have access to make updates to each tenant? What drives access? So far, we’ve been mainly concerned with cluster administrators, but now we need to worry about tenant administrators.
- **Enforce enterprise authentication:** When a developer or admin accesses a tenant, we’ll need to ensure that we’re doing so using our enterprise authentication.
- **Externalized Secrets:** We want to make sure our source of truth for secret data is outside of our cluster. This will make it easier for our security team to audit usage.
- **High Availability and Disaster Recovery:** There are going to be times that we need to impact if a tenant’s API server is running. We’ll need to rely on Kubernetes to make sure that, even during those times, there’s a way for tenants to do their work.
- **Encryption in Transit:** All connections between components need to be encrypted.
- **No Secrets in Helm Charts:** Keeping secret data in a chart would mean that it’s stored as a Secret in our namespace, violating the requirement to externalize secret data.

We’ve worked with most of these requirements already in this chapter. The key question is, “How do we pull everything together and automate it?” Having read through this chapter and looked through the scripts, you can probably see where this implementation is going. Just like any enterprise project, we need to understand how silos are going to impact our implementation. For our platform, we’re going to assume that:

- **Active Directory cannot be automatically updated:** It’s not unusual that you won’t be given the ability to create your own groups via an API to Active Directory (AD). While interacting with AD only requires LDAP capabilities, compliance requirements often dictate that a formal process is followed for creating groups and adding members to those groups.

- Vault can be automated:** Since Vault is API enabled and we have a good relationship with the Vault team, they'll let us automate the onboarding of new tenants directly.
- Intra-Cluster Communication does not require the Enterprise CA:** Enterprises often have their own **certificate authorities (CAs)** for generating TLS certificates. These CAs are generally not exposed to an external facing API and are not able to issue intermediate CAs that can be used by a local cert-manager instance. We'll use a CA specific to our cluster for issuing all certificates for use within the cluster.
- Host Cluster-Managed MySQL:** We're going to host our MySQL instance on the cluster, but we won't dive into operations around MySQL. We'll assume that it's been deployed as highly available. Database administration is its own discipline, and we won't pretend to be able to cover it in this section.

With these requirements and assumptions in hand, the next step is to plan out how to implement our multitenant platform.

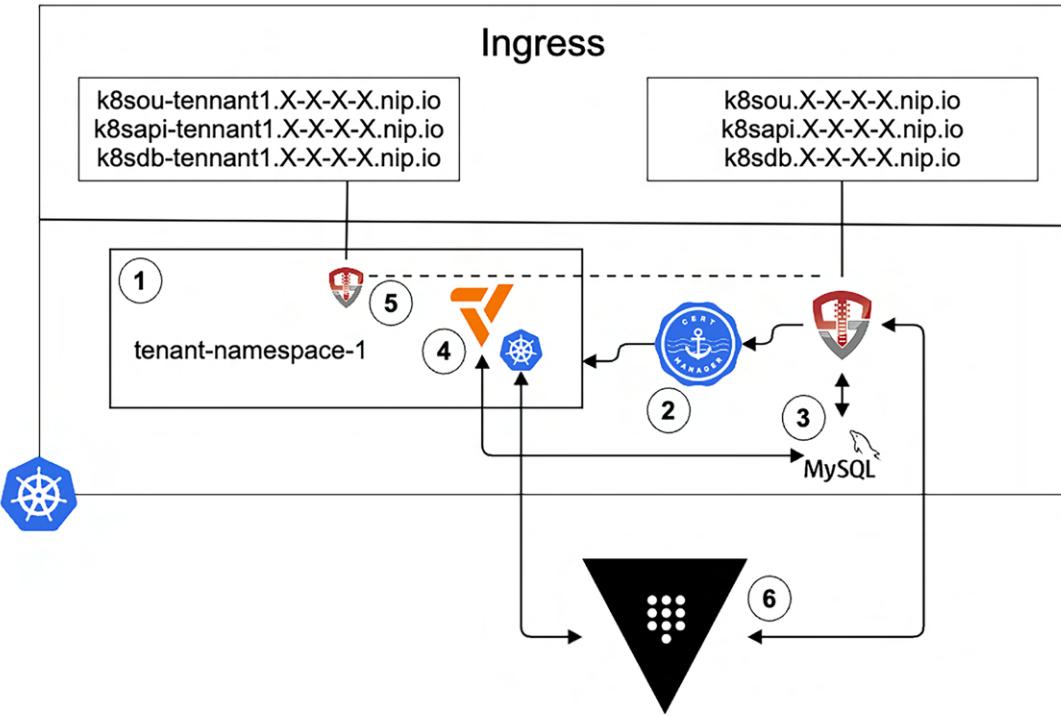
## Designing the Multitenant Platform

In the previous section, we defined our requirements. Now, let's construct a matrix of tools that will tell us what each component will be responsible for:

Requirement	Component	Notes
Portal Authentication	OpenUnison + Active Directory	OpenUnison will capture credentials; Active Directory will verify them.
Tenant	Kubernetes namespace + vCluster	Each tenant will receive their own namespace in the host cluster, with a vCluster deployed to it.
Tenant Authentication	OpenUnison	Each tenant will receive its own OpenUnison instance.
Authorization	OpenUnison with Active Directory Groups	Each tenant will have a unique Active Directory group that will provide administrative capabilities.
Certificate Generation	cert-manager Project	cert-manager will generate the keys needed to communicate between vCluster and MySQL.
Secrets Management	Centralized Vault	Each tenant will receive its own Vault database that will be enabled with Kubernetes authentication.
Orchestration	OpenUnison	We'll use OpenUnison's workflow engine for onboarding new tenants.

Table 9.1: Implementation matrix

Given our requirements and implementation matrix, our multitenant platform will look like this:



*Figure 9.6: Multitenant platform design*

Using the above diagram, let's walk through the steps that will need to occur to implement our platform:

1. OpenUnison will create a namespace and RoleBinding to our Active Directory group to the `admin ClusterRole`.
2. OpenUnison will generate a `Certificate` object in our tenant's namespace, which will be used by our vCluster to communicate with MySQL.
3. OpenUnison will create a database in MySQL for the vCluster and a user tied to the certificate generated in Step 2.
4. OpenUnison will deploy a Job that will run the `vcluster` command and deploy the tenant's vCluster.
5. OpenUnison will deploy a Job that will deploy the Kubernetes Dashboard, deploy OpenUnison, and integrate the vCluster OpenUnison into the host cluster's OpenUnison.
6. OpenUnison will create an authentication policy in Vault that will allow tokens from our tenant's vCluster to authenticate to Vault using local pod identities. It will also run a Job that will install the `vault` sidecar into our cluster.

We're giving our vCluster capabilities in the centralized Vault for retrieving secrets. In an enterprise deployment, you'd also want to control who can log in to Vault using the CLI and web interface using the same authentication and authorization as our clusters to tailor access, but that's beyond the scope of this chapter (and book).

Finally, you could use any automation engine you'd like to perform these tasks, such as **Terraform** or **Pulumi**. If you want to use one of these tools instead, the same concepts can be used and translated into the implementation-specific details. Now that we've designed our onboarding process, let's deploy it.

## Deploying Our Multitenant Platform

The previous section focused on the requirements and design of our multitenant platform. In this section, we're going to deploy the platform and walk through deploying a tenant. The first step is to start with a fresh cluster:

```
$ kind delete cluster -n cluster01  
$ kind delete cluster -n multinode  
$ cd chapter2  
$ ./create-cluster.sh
```

Once your cluster is running, the next step is to deploy the portal. We scripted everything:

```
$ cd chapter9/multitenant/setup/  
$ ./deploy_openunison.sh
```

This script does quite a bit:

1. Deploys cert-manager with internal CAs for our cluster
2. Deploys MySQL configured with our internal CA
3. Deploys OpenUnison, using impersonation, and deploys our customizations for vCluster
4. Deploys Vault
5. Integrates Vault and our control plane cluster
6. Enables OpenUnison to create new authentication mechanisms and policies in Vault

Depending on how much horsepower your infrastructure has, this script can take ten to fifteen minutes to run. Once deployed, the first step will be to log in to the portal at <https://k8sou.apps.IP.nip.io/>, where IP is your IP address with the dots changed to dashes. My cluster's IP is 192.168.2.82, so the URL is <https://k8sou.apps.192-168-2-102.nip.io/>. Use the user `mmosley` with the password `start123`. You'll notice a new badge called **New Kubernetes Namespace**. Click on that badge.

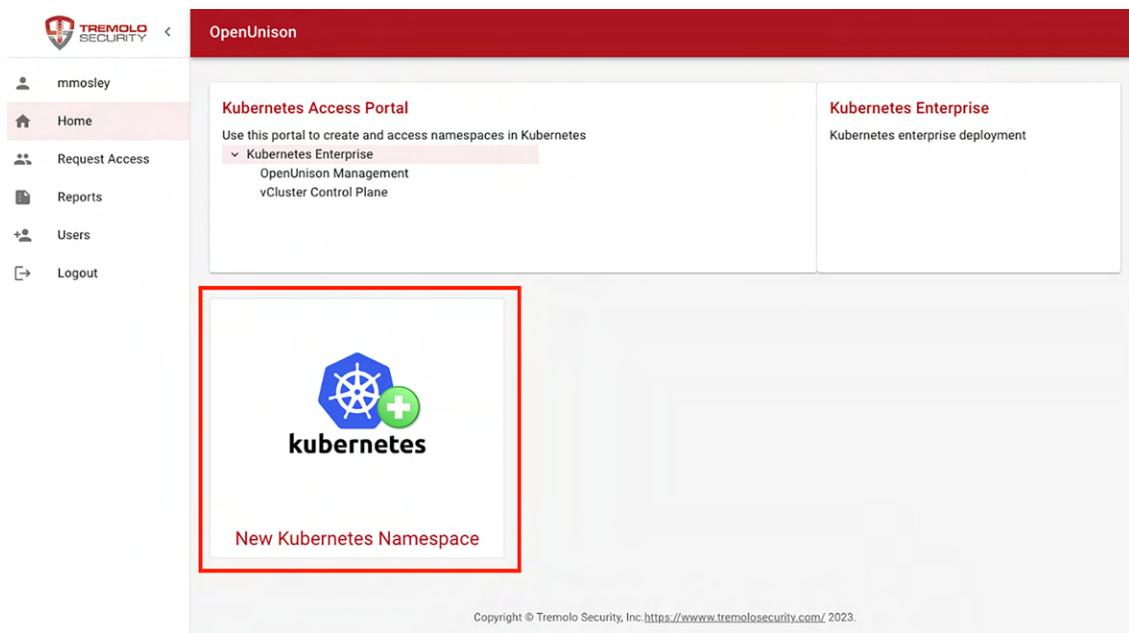


Figure 9.7: OpenUnison front page with the New Kubernetes Namespace badge

On the next screen, you'll be asked to provide some information for the new namespace (and tenant). We created two groups in our “Active Directory” for managing access to our tenant. While, out of the box, OpenUnison supports both the admin and view ClusterRole for mappings, we're going to focus on the admin ClusterRole mapping. The admin group for our namespace will also be the cluster-admin for our tenant vCluster. This means any user that is added to this group in Active Directory will gain cluster-admin access to our vCluster for this tenant. Fill out the form as you see in *Figure 9.8* and click **SAVE**.

The screenshot shows the 'Create New Namespace' form. The sidebar on the left includes icons for Home, Request Access, Reports, Users, and Logout. The main area has a red header bar with the text 'OpenUnison'. Below it, there's a section titled 'Create New Namespace' with the sub-section 'Create New Namespace' highlighted. The form fields are as follows:

- Cluster:** vCluster Control Plane
- Namespace Name:** tenant1
- Administrator Group:** cn=vcluster-test-admin,ou=Groups,DC=domain,DC=com
- Viewer Group:** cn=vcluster-test-view,ou=Groups,DC=domain,DC=com
- Reason:** Chapter 9 Demo

At the bottom, there are 'SAVE' and 'CANCEL REQUEST' buttons, and a copyright notice: Copyright © Tremolo Security, Inc. <https://www.tremolosecurity.com/> 2024.

Figure 9.8: New Namespace

Once saved, close this tab to return to the main portal and hit Refresh. You'll see that there is a new menu option on the left-hand side called **Open Approvals**. OpenUnison is designed around self-service, so the assumption is that the tenant owners will request that a new tenant be deployed. In this case, mmosley will be both the tenant owner and the approver. Click on **Open Approvals** and click **Act on Request**

The screenshot shows the OpenUnison interface with a sidebar on the left containing links for Home, Request Access, Open Approvals (which has a red notification badge), Reports, Users, and Logout. The main content area is titled 'Request Details' and shows the following information:

Subject Information	
Login ID	mmosley
Email Address	mmosley@tremolo.dev
Name	Create New Namespace
Description	Create New Namespace
Open Since	8/16/2024, 6:31:40 PM
Subject's Request	Create New Namespace - k8s - tenant1
Subject's Request Reason	Chapter 9 Demo

Below this is the 'Additional Request Details' section, which lists the Cluster Namespace as 'vCluster Control Plane tenant1'.

The 'Requester Details' section shows the requester's attributes and current roles:

Attributes		Current Roles		
Login ID	mmosley	Cluster	Namespace	Name
Email Address	mmosley@tremolo.dev	N/A	N/A	administrators-external
		N/A	N/A	users
		vCluster Control Plane	N/A	Administrators

The 'Act on Request' section at the bottom contains a justification field with the text 'For the chapter' and two buttons: 'APPROVE REQUEST' (in red) and 'DENY REQUEST'.

Figure 9.9: Approval screen

Fill in the **Justification** field and click on **APPROVE REQUEST** and **CONFIRM APPROVAL**. This will approve the user's request and launch a workflow that implements the steps we designed in *Figure 9.6*. This workflow will take five to ten minutes, depending on the horsepower of your cluster. Usually, OpenUnison will send the requestor an email once a workflow is complete, but we're using an SMTP blackhole here to pull in all emails that are generated to make the lab implementation easier. You'll have to wait until the `tenant1` namespace is created and the OpenUnison instance is running. If you look in the `tenant1` namespace on the host cluster, you'll see that the `vault-agent-injector` pod is running. This lets you know the rollout is complete.



To track your vCluster's deployment, there are three pods to look at:

- `onboard-vcluster-openunison-tenant1` – The pod from this Job contains the logs for creating and deploying vCluster into your tenant's namespace.
- `deploy-helm-vcluster-tenant1` – The pod from this Job integrates Vault.
- `openunison-orchestra` – The pod from this deployment runs OpenUnison's onboarding workflows.

If there are any errors in the process, you'll find them here.

Now that our tenant has been deployed, we can log in and deploy a pod. Log out of OpenUnison, and log back in using the user name `jjackson` and the password `start123`. The `jjackson` user is a member of our admin group in Active Directory, so they'll immediately be able to access and administer the vCluster in the `tenant1` namespace.

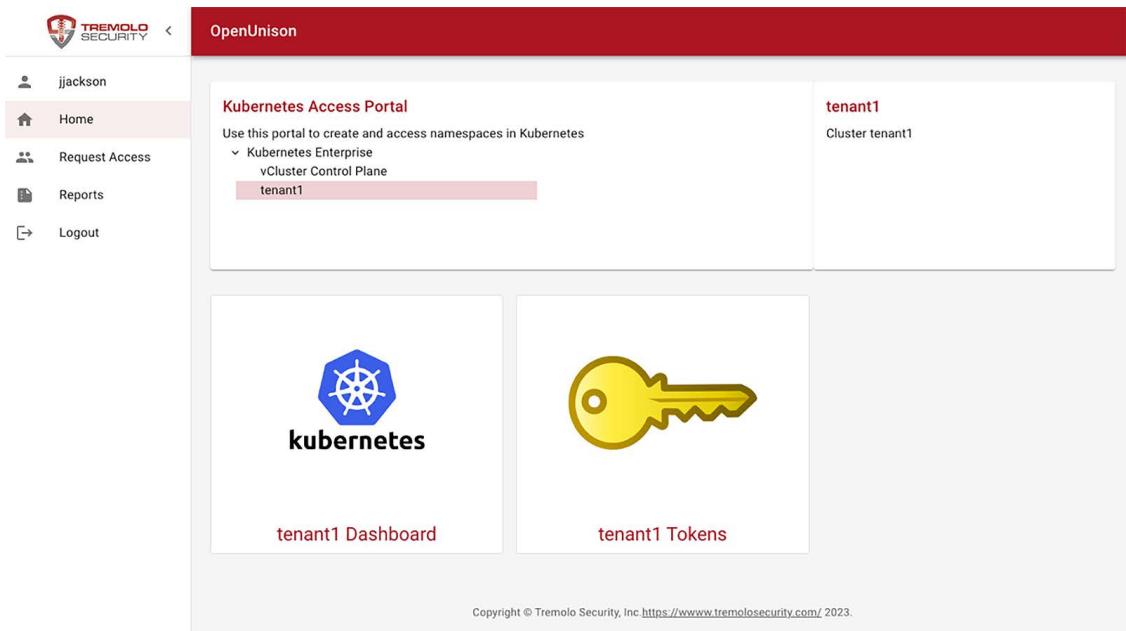


Figure 9.10: Access to tenant1 vCluster

The `jjackson` user is able to interact with our vCluster the same way they would with the host cluster, using either the dashboard or directly via the CLI. We're going to use `jjackson`'s session to log in to our tenant vCluster and deploy a pod that uses secret data from our Vault. First, `ssh` into your cluster's host on a new session and create a secret in our Vault for our pod to consume:

```
$ ssh ubuntu@192.168.2.82
$ cd chapter9/multitenant/setup/vault
$ ./vault_cli.sh
$ vault kv put secret/data/vclusters/tenant1/ns/default/config some-password=mysupersecretp@ssw0rd
```

The last command creates our secret data in the Vault. Note that the path that we created specifies that we're working with the `tenant1` vCluster in the `default` namespace. The way our cluster is deployed, only pods with `ServiceAccounts` in the `default` namespace for our `tenant1` vCluster will be able to access the `some-password` value.

Next, let's log in to our vCluster using `jjackson`. First, set your `KUBECONFIG` variable to a temporary file and set `jjackson`'s session up using the `tenant1` token:

```
$ export KUBECONFIG=$(mktemp)
$ export TMP_CERT=$(mktemp)...
Cluster "tenant1" set.
Context "tenant1" created.
User "jjackson@tenant1" set.
Switched to context "tenant1".
$ cd ../../examples
$ ./create-vault.sh
$ kubectl logs test-vault-vault-watch -n default
Defaulted container "test" out of: test, vault-agent, vault-agent-init (init)
Wed Oct 11 16:30:37 UTC 2023
MY_SECRET_PASSWORD="mysupersecretp@ssw0rd"
```

The pod was able to authenticate to our Vault using its own `ServiceAccount`'s identity to retrieve the secret! We did have to make two updates to our pod for our vCluster to connect to Vault:

```
annotations:
  vault.hashicorp.com/service: "https://vault.apps.192-168-2-82.nip.io"
  vault.hashicorp.com/auth-path: "auth/vcluster-tenant1"
  vault.hashicorp.com/agent-inject: "true"
  vault.hashicorp.com/log-level: trace
  vault.hashicorp.com/role: cluster-read
  vault.hashicorp.com/tls-skip-verify: "true"
  vault.hashicorp.com/agent-inject-secret-myenv: 'secret/data/vclusters/tenant1/ns/default/config'
```

```
vault.hashicorp.com/secret-volume-path-myenv: '/etc/secrets'  
vault.hashicorp.com/agent-inject-template-myenv: |  
  {{- with secret "secret/data/vclusters/tenant1/ns/default/config" -}}  
    MY_SECRET_PASSWORD="{{ index .Data "some-password" }}"  
  {{- end -}}
```

We had to add a new annotation to tell the Vault sidecar where to authenticate. The `auth/vcluster-tenant1` authentication path was created by our onboarding workflow. We also needed to set the requested role to `cluster-read`, which was also created by the onboarding workflow. Finally, we needed to tell the sidecar where to look up our secret data.

With that, we've now built the start of a self-service multitenant portal! We're going to expand on this portal as we dive into more topics that are important to multitenancy. If you want to dive into the code for how we automated the vCluster onboarding, `chapter9/multitenant/vcluster-multitenant` is the Helm chart that holds the custom workflows and `templates/workflows/onboard-vcluster.yaml` is the starting point for all the work that gets done. We broke up each major step in its own workflow to make it easier to read.

## Summary

Multitenancy is an important topic in modern Kubernetes deployments. Providing a shared infrastructure for multiple tenants cuts down on resource utilization and can provide more flexibility while creating the isolation needed to maintain both security and compliance. In this chapter, we worked through the benefits and challenges of multitenancy in Kubernetes, introduced the vCluster project, and learned how to deploy vClusters to support multiple tenants. Finally, we walked through implementing a self-service multitenant portal and integrated our Vault deployment so tenants could have their own secrets management.

In the next chapter, we'll dive into the security of the Kubernetes Dashboard. We've used it and deployed it in the last few chapters, and now we're going to understand how its security works and how those lessons learned apply to other cluster management systems too.

## Questions

1. Kubernetes Custom Resource Definitions can support multiple versions.
  - a. True
  - b. False
2. What is the security boundary in Kubernetes?
  - a. pods
  - b. Containers
  - c. NetworkPolicies
  - d. Namespaces

3. Where do the pods in a vCluster run?
  - a. In the vCluster
  - b. In the host cluster
  - c. There are no pods
4. vClusters have their own Ingress controllers.
  - a. True
  - b. False
5. vClusters share keys with host clusters.
  - a. True
  - b. False

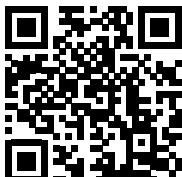
## Answers

1. b – False – There's some version management, but generally you can only have one version of a CRD.
2. d – Namespaces are the security boundary in Kubernetes.
3. b – When a pod is created in a vCluster, the syncer creates a matching pod in the host cluster for scheduling.
4. b – False – Generally, the Ingress object is synced into the host cluster.
5. b – False – Each vCluster gets its own unique keys to identify it.

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 10

## Deploying a Secured Kubernetes Dashboard

The Kubernetes Dashboard is a very helpful tool for understanding how your cluster is running. It's often the first thing someone will install when learning Kubernetes because it shows you something. Even after the beginning stage, dashboards provide a tremendous amount of information very quickly in a way that isn't possible using `kubectl`. On one screen, you can quickly see what workloads are running, where, how many resources they're using, and if you need to update them, you can do so quickly. Too often, the dashboard is called "insecure" or difficult to access. In this chapter, we're going to show you how the dashboard is in fact quite secure and how to make it easy to access.

Beyond the Kubernetes Dashboard, Kubernetes clusters are made up of more than the API server and the `kubelet`. Clusters are generally made up of additional applications that need to be secured, such as container registries, source control systems, pipeline services, GitOps applications, and monitoring systems. The users of your cluster will often need to interact with these applications directly.

While many clusters are focused on authenticating access to user-facing applications and services, cluster solutions are not given the same first-class status. Users are often asked to use `kubectl`'s port-forward or proxy capability to access these systems. This method of access is an anti-pattern from a security and user experience standpoint. The first exposure users and administrators will have to this anti-pattern is the Kubernetes Dashboard. This chapter will detail why this method of access is an anti-pattern and how to properly access the dashboard. We'll walk you through how not to deploy a secure web application and point out the issues and risks so that you'll know what to look for when being advised on how to access management applications.

We'll use the Kubernetes Dashboard as a way to learn about web application security and how to apply those patterns in your own cluster. These lessons will work with not just the dashboard but also other cluster-focused applications such as the **Kiali dashboard** for Istio, Grafana, Prometheus, ArgoCD, and other cluster management applications.

Finally, we'll spend some time talking about local dashboards and how to evaluate their security. This is a popular trend, but not universal. It's important to understand the security of both approaches, and we'll explore them in this chapter.

In this chapter, we will cover the following topics:

- How does the dashboard know who you are?
- Understanding dashboard security risks
- Deploying the dashboard with a reverse proxy
- Integrating the dashboard with OpenUnison
- What's changed in the Kubernetes Dashboard 7.0

Having covered what we'll work through in this chapter, next, let's work through the technical requirements for this chapter.

## Technical requirements

To follow the exercises in this chapter, you will require a fresh KinD cluster from *Chapter 2, Deploying Kubernetes Using KinD*.

You can access the code for this chapter at the following GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter10>.

## Getting help

We do our best to test everything, but there are sometimes half a dozen systems or more in our integration labs. Given the fluid nature of technology, sometimes things that work in our environment don't work in yours. Don't worry, we're here to help! Open an issue on our GitHub repo at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/issues> and we'll be happy to help you out!

## How does the dashboard know who you are?

The Kubernetes Dashboard is a powerful web application for quickly accessing your cluster from inside a browser. It lets you browse your namespaces and view the status of nodes and even provides a shell you can use to access pods directly. There is a fundamental difference between using the dashboard and kubectl. The dashboard, being a web application, needs to manage your session, whereas kubectl does not. This means there's a different set of security issues during deployment that are often not accounted for, leading to severe consequences. In this section, we'll explore how the dashboard identifies users and interacts with the API server.

## Dashboard architecture

Before diving into the specifics of how the dashboard authenticates a user, it's important to understand the basics of how the dashboard works. The dashboard, at a high level, has three logical layers:

- **User interface:** This is the Angular + HTML frontend that is displayed in your browser and that you interact with
- **Middle tier:** The frontend interacts with a set of APIs hosted in the dashboard's container to translate calls from the frontend into Kubernetes API calls
- **API server:** The middle-tier API interacts directly with the Kubernetes API server

This three-layered architecture of the Kubernetes Dashboard can be seen in the following diagram:

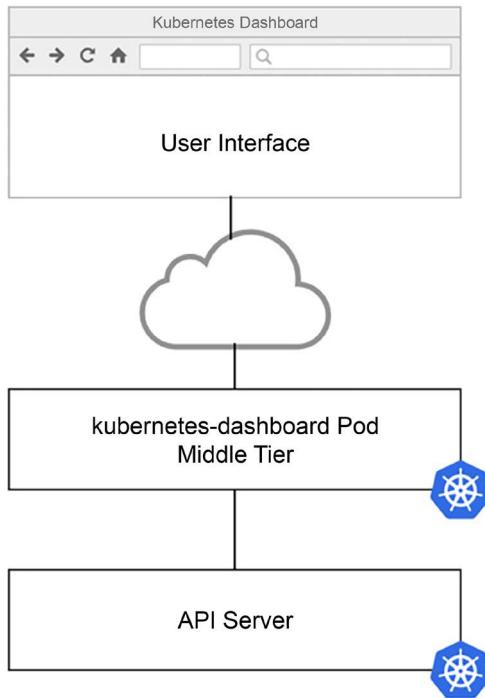


Figure 10.1: Kubernetes Dashboard logical architecture

When a user interacts with the dashboard, the user interface makes calls to the middle tier, which in turn makes calls to the API server. The dashboard doesn't know how to collect credentials; there's no place to supply a username or password to log in to the Dashboard. It has a very simple session mechanism system based on cookies, but for the most part, the dashboard doesn't really know, or care, who the currently logged-in user is. The only thing the dashboard cares about is what token to use when communicating with the API server.

While this is the logical architecture, the physical architecture divides these components across different containers:

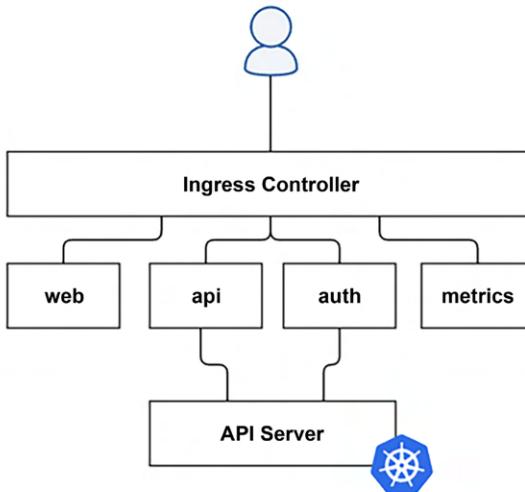


Figure 10.2: Kubernetes Dashboard container architecture

Starting with version 7.0, the dashboard is now broken into five components in their own containers:

- **Web:** The user interface for the dashboard, serving the HTML and JavaScript rendered by your browser. This component has no authentication and doesn't need it.
- **Api:** This container hosts the workhorse of the dashboard. It's the component that interacts with the API server on your behalf. This container needs to know who you are.
- **Auth:** The auth container is used to tell the frontend if your token is valid. If the token isn't valid, the UI will redirect you to login by providing the token to the API server for verification.
- **Metrics:** This container provides a metrics endpoint for **Prometheus** of the Kubernetes Dashboard.
- **Ingress Controller:** Since each of these containers provide their own path from the same host, something needs to combine them into a single URL. The default deployment includes Kong's Ingress controller.

If you looked at the pods running in *Chapter 6*, you'll notice that neither Kong nor the auth container are running. We'll cover that a bit later. Now that we understand how the dashboard is architected, how does the dashboard know who you are? Let's walk through the options.

## Authentication methods

There are two ways that the dashboard can determine who a user is:

- **Token from login/uploaded kubectl configuration:** The dashboard can prompt the user for their kubectl configuration file or for a bearer token to use. Once a token is provided, the UI uses it as a header to the API container. There is no session management. When the token is no longer valid, the user is redirected back to the login screen to upload a new token.

- **Token from a reverse proxy:** If there's an authorization header containing a bearer token in requests from the user interface to the middle tier, the middle tier will use that bearer token in requests to the API server. This is the most secure option and the implementation that will be detailed in this chapter.

If you've read our previous editions, or have used previous versions of the dashboard, you may be wondering what happened to using the dashboard's own identity and skipping login. The architectural changes in version 7.x meant the removal of this option. No matter what, there *MUST* be an Authorization header with a token in each request. This is a very positive development as it makes it very hard to deploy a dashboard that can be taken over by an anonymous request. In fact, we removed the section of this chapter that talked about how to compromise an improperly deployed dashboard because that attack vector is no longer valid.

Throughout the rest of this chapter, the first option will be explored as an anti-pattern for accessing the dashboard, and we will explain why the reverse proxy pattern is the best option for accessing a cluster's dashboard implementation from a security standpoint and a user experience standpoint.

Let's now try to understand dashboard security risks.

## Understanding dashboard security risks

The question of the dashboard's security often comes up when setting up a new cluster. Securing the dashboard boils down to how the dashboard is deployed, rather than if the dashboard itself is secure. Going back to the architecture of the dashboard application, there is no sense of "security" being built in. The middle tier simply passes a token to the API server.

When talking about any kind of IT security, it's important to look at it through the lens of *defense in depth*. This is the idea that any system should have multiple layers of security. If one fails, there are other layers to fill the gap until the failed layers can be addressed. A single failure doesn't give an attacker direct access.

The most often cited incident related to the dashboard's security was the breach of Tesla in 2018 by crypto miners. Attackers were able to access pods running in Tesla's clusters because the dashboard wasn't secured.

The cluster's pods had access to tokens that provided the attackers with access to Tesla's cloud providers where the attackers ran their crypto-mining systems. It's important to note that this attack would not have worked in version 7.x and above because the api container will not accept requests that don't have Authorization headers.

Dashboards in general are often an attack vector because they make it easy to find what attackers are looking for and can easily be deployed insecurely. Illustrating this point, at KubeCon NA 2019, a **Capture the Flag (CTF)** challenge was presented where one of the scenarios was a developer "accidentally" exposing the cluster's dashboard.

The CTF challenge is available as a home lab at <https://securekubernetes.com/>. It's a highly recommended resource for anyone learning the Kubernetes security. In addition to being educational (and terrifying), it's also really fun!

Since we can no longer deploy a dashboard without some kind of authentication, we're going to focus on the security issues of using ServiceAccount tokens and the default. Additionally, there is no encryption between Kong and the downstream services.

## Exploring Dashboard Security Issues

The version 7.x dashboard eliminated the ability to deploy the dashboard without a login, but there are still some security issues with the default deployment that should be addressed.

First, deploy the dashboard to your cluster:

```
helm repo add kubernetes-dashboard https://kubernetes.github.io/dashboard/  
helm upgrade --install kubernetes-dashboard kubernetes-dashboard/kubernetes-  
dashboard --create-namespace --namespace kubernetes-dashboard
```

When you inspect the pods, you'll notice that there are now the five containers we described earlier. There's also a NodePort service for the Kong ingress gateway. While you can choose to deploy a different Ingress, we're going to focus on the defaults. You'll also see there's also no encryption from Kong to the pods.

## Using a token to log in

A user may upload a token to the dashboard on the login screen. As discussed earlier, the dashboard will take the user's bearer token and use it with all requests to the API server. While this may appear to provide a secure solution, it brings its own issues. The dashboard isn't kubectl and doesn't know how to refresh tokens as they expire. This means that a token would need to be fairly long-lived to be useful. It would require either creating service accounts that can be used or making your OpenID Connect id\_tokens longer-lived. Both options would negate much of the security put in place by leveraging OpenID Connect for authentication.

As has been repeated throughout the book, ServiceAccount tokens were never meant to be used outside of the cluster. You'll need to distribute the token, and of course since it's a bearer token it's easy to lose, maybe it gets checked in to a git repo, or can be leaked by some buggy code. There's also no difference between a token that is used by the dashboard vs Kubernetes, so a leaked token can be used directly against the Kubernetes API. While this solution exists to make it relatively easy to use the dashboard, it shouldn't be used in production.

Having looked at the issues with a token login into the dashboard, next we'll look at the issues with the default installation and lack of encryption.

## Unencrypted Connections

The default dashboard Helm chart doesn't encrypt connections from the Ingress controller, Kong by default, to the individual containers. Regardless of how you authenticate to the dashboard, this can be a serious security antipattern. As we've discussed, a bearer token can be used by anyone with network access, which means a lost token from an unencrypted network connection can lead to a serious breach. Even if using short-lived tokens, this is a worrying design choice. Whenever you build in security, it's important to use a defense-in-depth approach, where you never have a single point of failure. In this case, the lack of encryption means that you could have that single point of failure by not having any fall back.

When you deploy your dashboard, you should enable encryption between the reverse proxy and the api container. We'll walk through that in the next section. The web and metrics containers aren't as important. The auth container doesn't support any encryption, which is an issue, but with the right configuration can be bypassed.

An alternative approach is to rely on a service mesh like Istio. If you enable a mesh, you could use that to rely on for encryption, but that's an additional component to add.

Given how easy it is to create an internal certificate authority, there really isn't any reason to not have these connections encrypted.

While we've focussed on the security concerns of the default Kubernetes Dashboard installation, we'll next move on to how to correctly deploy the dashboard.

## Deploying the dashboard with a reverse proxy

Proxies are a common pattern in Kubernetes; there are proxies at every layer in a Kubernetes cluster. The proxy pattern is also used by most service mesh implementations on Kubernetes, creating sidecars that will intercept requests. The difference between the reverse proxy described here and these proxies is in their intent. Microservice proxies often do not carry a session, whereas web applications need a session to manage the state.

The following diagram shows the architecture of a Kubernetes Dashboard with a reverse proxy:

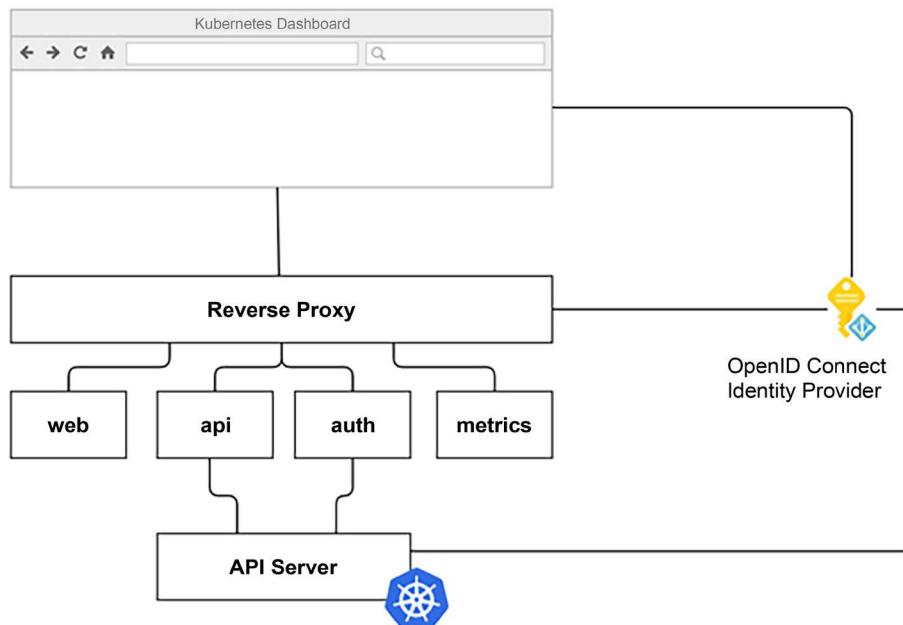


Figure 10.3: Kubernetes Dashboard with a reverse proxy

The reverse proxy shown in *Figure 10.3* performs four roles:

- **Routing:** Each of the containers used by the dashboard has its own path off of the host URL. The reverse proxy is responsible for routing requests to the correct container.
- **Authentication:** The reverse proxy intercepts unauthenticated requests (or stale sessions) and triggers the authentication process with an OpenID Connect identity provider to authenticate the user.
- **Session management:** The Kubernetes Dashboard is a user-facing application. It should have the typical controls put in place to support session timeouts and revocation. Be wary of a reverse proxy that stores all session data in a cookie. These methods are difficult to revoke.
- **Identity injection:** Once the proxy has authenticated a user, it needs to be able to inject an HTTP authorization header on each request that is a JWT identifying the logged-in user, is signed by the same OpenID Connect identity provider, and has the same issuer and recipient as the API server. The exception to this is using impersonation, which, as discussed in *Chapter 6, Integrating Authentication into Your Cluster*, injects specific headers into the requests.

What's important is that when you configure your reverse proxy, it should:

1. **Encrypt traffic to the api and the auth containers:** These two containers are the ones that need the user's token, so encryption is important. Since the auth container doesn't support any encryption, you may want to just bypass this container entirely. We'll explain this more in the next section when we talk about how OpenUnison integrates with the dashboard.
2. **Manage and renew tokens:** There's no reason to use long-lived tokens with your reverse proxy. It should be able to renew them based on how long the token is good for.

Combining these means eliminating the Kong Ingress controller. It's not needed anymore because your authenticating reverse proxy is doing the work.

The reverse proxy does not need to run on the cluster. Depending on your setup, it may be advantageous to do so, especially when utilizing impersonation with your cluster. When using impersonation, the reverse proxy uses a service account's token, so it's best for that token to never leave the cluster.

The focus of this chapter has been on the Kubernetes project's dashboard. There are multiple options for dashboard functionality. Next, we'll explore how these dashboards interact with the API server and how to evaluate their security.

## Local dashboards

A common theme among third-party dashboards is to run locally on your workstation and use a Kubernetes SDK to interact with the API server the same way `kubectl` would. These tools offer the benefit of not having to deploy additional infrastructure to secure them.

Visual Studio Code's Kubernetes plugin is an example of a local application leveraging direct API server connections. When launching the plugin, Visual Studio Code accesses your current `kubectl` configuration and interacts with the API server using that configuration. It will even refresh an OpenID Connect token when it expires:

```

! pod-openunison-operator-858d496-lpm52.yaml - Visual Studio Code - Insiders
File Edit Selection View Go Debug Terminal Help pod-openunison-operator-858d496-lpm52.yaml

KUBERNETES
  CLUSTERS
    kubernetes
      Namespaces
        default
        ingress-nginx
        kube-node-lease
        kube-public
        kube-system
        kubernetes-dashboard
        local-path-storage
        not-going-to-work
          openunison
          still-not-going-to-work
        Nodes
        Workloads
          Deployments
            openunison-operator
              openunison-operator-858d496-lpm52
                Running (1/1)
                10.240.189.139
                openunison-orchestra
                StatefulSets
                DaemonSets
                Jobs
                CronJobs
                Pods
                Network
                Storage
                Configuration
      HELM REPOS
      CLOUDS

! pod-openunison-operator-858d496-lpm52.yaml > abc apiVersion
1   apiVersion: v1
2   kind: Pod
3   metadata:
4     annotations:
5       cni.projectcalico.org/podIP: 10.240.189.139/32
6       cni.projectcalico.org/podIPs: 10.240.189.139/32
7       creationTimestamp: "2020-05-20T00:53:43Z"
8       generateName: openunison-operator-858d496-
9       labels:
10      app: openunison-operator
11      pod-template-hash: 858d496
12      name: openunison-operator-858d496-lpm52
13      namespace: openunison
14      ownerReferences:
15        - apiVersion: apps/v1
16          blockOwnerDeletion: true
17          controller: true
18          kind: ReplicaSet
19          name: openunison-operator-858d496
20          uid: f0858ee2-bd6f-4c35-ba93-c5bf4729a28a
21        resourceVersion: "11683"
22        selfLink: /api/v1/namespaces/openunison/pods/openunison-operator-858d496-lp
23        uid: 33e5f72a-43af-41ad-afa8-461ff212ffb0
24      spec:
25        containers:
26          - command:
27            - java
28            - -jar
29            - ./usr/local/openunison/javascript-operator.jar
30            - -tokenPath
31            - ./var/run/secrets/kubernetes.io/serviceaccount/token
32            - -rootCaPath
33            - ./var/run/secrets/kubernetes.io/serviceaccount/ca.crt
34            - -kubernetesURL

```

In 1, Col 1   Spaces: 2   UTF-8   LF   YAML   Q 2

Figure 10.4: Visual Studio Code with the Kubernetes plugin

The Kubernetes plugin for Visual Studio Code is able to refresh its OpenID Connect token because it's built with the client-go SDK, the same client libraries used by `kubectl`. When evaluating a client dashboard, make sure it works with your authentication type, even if it isn't OpenID Connect. Many of the SDKs for Kubernetes don't support OpenID Connect token refreshes. The Java and Python SDKs only recently (as of the published date of this book) began supporting the refresh of OpenID Connect tokens the way the client-go SDK does. When evaluating a local dashboard, make sure it's able to leverage your short-lived tokens and can refresh them as needed, just like `kubectl` can.

There is no shortage of different dashboards in the Kubernetes ecosystem, all with their own spins on management. I don't want to simply provide a list of these dashboards without giving you an in-depth review of their benefits and security impacts. Instead, let's focus on what's important when evaluating which dashboard you want to use:

- If the dashboard is web-based:
  - Does it support OpenID Connect directly?
  - Can it run behind a reverse proxy and accept both tokens and impersonation headers?
  - Does it require any permissions for its own service account? Do these permissions adhere to a least-privilege approach?

- If the dashboard is local:
  - Does the client SDK support OpenID Connect, with the ability to automatically refresh tokens as `kubectl` does, using the client-go SDK?

These are important evaluation questions not just for the Kubernetes Dashboard, but for dashboards that you may use for other cluster management applications. As an example, the **TektonCD dashboard**, which is a web application for managing your pipelines, requires deleting several RBAC bindings to make sure the dashboard has to use the user's identity and can't be co-opted to use its `ServiceAccount` identity.

## Other cluster-level applications

The introduction of this chapter discussed how a cluster is made up of several applications besides Kubernetes. Other applications will likely follow the same model as the dashboard for security, and the reverse proxy method is a better method for exposing those applications than `kubectl` port-forwarding, even when the application has no built-in security. Take the common Prometheus stack as an example. Grafana has support for user authentication, but Prometheus and Alert Manager do not.

How would you track who had access to these systems or when they were accessed using port-forwarding?

Using a reverse proxy, logs of each URL and the user that was authenticated to access the URL can be forwarded to a central log management system and analyzed by a **Security Information and Event Manager (SIEM)** providing an additional layer of visibility into a cluster's usage.

Just as with the dashboard, using a reverse proxy with these applications provides a layered security approach. It offloads session management from the application in question and provides the capability to have enhanced authentication measures in place such as multi-factor authentication and session revocation. These benefits will lead to a more secure and easier-to-use cluster.

Let's now discuss how to integrate the dashboard with OpenUnison.

## Integrating the dashboard with OpenUnison

The topic of how OpenUnison injects identity headers using impersonation was covered in *Chapter 6, Integrating Authentication into Your Cluster*, but not how OpenUnison injects a user's identity into the dashboard with an OpenID Connect integrated cluster. It worked, but it wasn't explained. This section will use the OpenUnison implementation as an example of how to build a reverse proxy for the dashboard. Use the information in this section to get a better understanding of API security or to build your own solution for dashboard authentication.

The OpenUnison deployment comprises two integrated applications:

- **The OpenID Connect Identity Provider & Login Portal:** This application hosts the login process and the discovery URLs used by the API server to get the keys needed to validate an `id_token`. It also hosts the screens where you can obtain your token for `kubectl`.
- **The dashboard:** A reverse proxy application that authenticates to the integrated OpenID Connect identity provider and injects the user's `id_token` into each request.

This diagram shows how the dashboard's user interface interacts with its server-side component with a reverse proxy injecting the user's `id_token`:

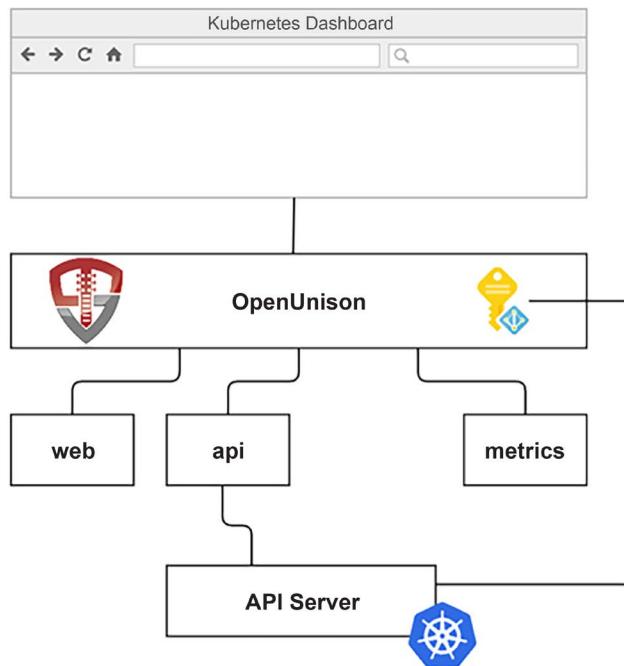


Figure 10.5: OpenUnison integration with the dashboard

The dashboard uses the same OpenID Connect identity provider as the API server but doesn't use the `id_token` provided by it. Instead, OpenUnison has a plugin that will generate a new `id_token` independent of the identity provider with the user's identity data in it. OpenUnison can do this because the key used to generate an `id_token` for the OpenID Connect identity provider, used by `kubectl` and the API server, is stored in OpenUnison. This is different from how you would integrate the dashboard with Keycloak or Dex because you would need an additional component to authenticate users and maintain the `id_token` that is injected into the requests. This is often done with the OAuth2 proxy, which would need to be integrated with both your identity provider (i.e., Dex or Keycloak), the dashboard, and your ingress controller. OpenUnison did all these steps for you.

A new, short-lived token is generated separately from the OpenID Connect session used with `kubectl`. This way, the token can be refreshed independently of a `kubectl` session. This process provides the benefits of a 1- to 2-minute token life with the convenience of a direct login process.

You'll also notice that there is no auth container. The auth container's only role is to return some JSON to tell the UI that the user is still authenticated. Since this container doesn't support any encryption, we don't bother calling it and instead generate the JSON directly in OpenUnison. This cuts out the need for the auth container and any issues that might arise from not having a TLS network connection with a bearer token.

If you have an eye for security, you may point out that this method has a glaring single point of failure in the security model: a user's credentials! An attacker generally just needs to ask for credentials in order to get them. This is often done via email in an attack called phishing, where an attacker sends a victim a link to a page that looks like their login page but really just collects credentials. This is why multi-factor authentication is so important for infrastructure systems.

In a 2019 study, Google showed that multi-factor authentication stopped 99% of automated and phishing attacks (<https://security.googleblog.com/2019/05/new-research-how-effective-is-basic.html>). Adding multi-factor authentication to the identity provider OpenUnison authenticates against, or integrating it directly into OpenUnison, is one of the most effective ways to secure the dashboard and your cluster.

Next, we'll look at what's changed with the new release of the dashboard.

## What's changed in the Kubernetes Dashboard 7.0

We've spent this chapter talking about the 7.0 dashboard, but as is often true in enterprises the old 2.7 dashboard is still in use and probably will be for a while. The major difference between the 2.7 version and the 7.0 version that is coming is that the API layer and the frontend layer are broken up into multiple containers in 7.0. This was done by the maintainers to make it easier to support more complex use cases, so keep an eye on this project!

## Summary

In this chapter, we explored the security of the Kubernetes Dashboard in detail. First, we walked through the architecture and how the dashboard passes your identity information on to the API server. We then explored how the dashboard gets compromised, and finally, we detailed how to correctly deploy the dashboard securely.

With this knowledge, you can now provide a secure tool to your users. Many users prefer the simplicity of accessing the dashboard via a web browser. Adding multi-factor authentication adds an additional layer of security and peace of mind. When your security team questions the security of the dashboard, you'll have the answers needed to satisfy their concerns.

The previous three chapters focused on the security of the Kubernetes APIs. Next, in *Chapter 11, Extending Security Using Open Policy Agent*, we'll explore securing the soft underbelly of every Kubernetes deployment: nodes!

## Questions

1. The dashboard is insecure.
  - a. True
  - b. False
2. How can the dashboard identify a user?
  - a. A token injected from a reverse proxy or provided by the login form

- b. Username and password
  - c. service account
  - d. Multi-factor authentication
3. How does the dashboard track the session state?
- a. Sessions are stored in etcd
  - b. Sessions are stored in custom resource objects called `DashboardSession`
  - c. There are no sessions
  - d. If a token is uploaded, it's encrypted and stored in the browser as a cookie
4. When using a token, how often can the dashboard refresh it?
- a. Once a minute
  - b. Every thirty seconds
  - c. When the token expires
  - d. None of the above
5. What's the best way to deploy the dashboard?
- a. Using `kubectl port-forward`
  - b. Using `kubectl proxy`
  - c. With a secret Ingress host
  - d. Behind a reverse proxy
6. The dashboard doesn't support impersonation.
- a. True
  - b. False
7. OpenUnison is the only reverse proxy that supports the dashboard.
- a. True
  - b. False

## Answers

1. b
2. a – There must be a token
3. c – When your token expires, you'll be asked for a new one
4. d – The dashboard can't refresh tokens
5. d – Better security and usability
6. b
7. b



# 11

## Extending Security Using Open Policy Agent

So far, we have covered Kubernetes' built-in authentication and authorization capabilities, which help to secure a cluster. While this will cover most use cases, it doesn't cover all of them. Some security best practices that Kubernetes can't handle are pre-authorizing container registries and ensuring that Ingress objects don't overlap (though most Ingress controllers do check, such as NGINX).

These tasks are left to outside systems and are called dynamic admission controllers. **Open Policy Agent (OPA)** and its Kubernetes native sub-project, **Gatekeeper**, is one of the most popular ways to handle these use cases. This chapter will detail the deployment of OPA and Gatekeeper, how OPA is architected, and how to develop policies.

In this chapter, we will cover the following topics:

- Introduction to dynamic admission controllers
- What is OPA and how does it work?
- Using Rego to write policies
- Enforcing Ingress policies
- Mutating objects and default values
- Creating policies without Rego

Once you've completed this chapter, you'll be on your way to developing and implementing important policies for your cluster and workloads.

### Technical requirements

To complete the hands-on exercises in this chapter, you will require an Ubuntu 22.04 server.

You can access the code for this chapter at the following GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter11>.

## Introduction to dynamic admission controllers

An admission controller is a specialized webhook in Kubernetes that runs when an object is created, updated, or deleted. When one of these three events happens, the API server sends information about the object and operation to the webhook. Admission controllers can be used to either determine if an operation should happen or give the cluster operator a chance to change the object definition before it's processed by the API server. We're going to look at using this mechanism to both enforce security and extend the functionality of Kubernetes.

There are two ways to extend Kubernetes:

- Build a custom resource definition so that you can define your own objects and APIs.
- Implement a webhook that listens for requests from the API server and responds with the necessary information. You may recall that in *Chapter 6, Integrating Authentication into Your Cluster*, we explained that a custom webhook could be used to validate tokens.

Starting in Kubernetes 1.9, a webhook can be defined as a dynamic admission controller, and in 1.16, the dynamic admission controller API became **Generally Available (GA)**.

There are two types of dynamic admission controllers, validating and mutating. Validating admission controllers verify that a new object, update, or deletion can move forward. Mutation allows a webhook to change the payload of an object's creation, deletion, or update. This section will focus on the details of admission controllers. We'll talk more about mutation controllers in the next chapter, *Chapter 12, Node Security with GateKeeper*.

The protocol is very straightforward. Once a dynamic admission controller is registered for a specific object type, the webhook is called with an HTTP POST every time an object of that type is created or edited. The webhook is then expected to return JSON, which represents whether it is allowed or not.

As of 1.16, `admission.k8s.io/v1` is at GA. All examples will use the GA version of the API.

The request submitted to the webhook is made up of several sections. We're not including an example here because of how large an `Admission` object can get, but we'll use [https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/blob/main/chapter11/example\\_admission\\_request.json](https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/blob/main/chapter11/example_admission_request.json) as an example:

- **Object identifiers:** The `resource` and `subResource` attributes identify the object, API, and group. If the version of the object is being upgraded, then `requestKind`, `requestResource`, and `requestSubResource` are specified. Additionally, `namespace` and `operation` are provided to provide the location of the object and whether it is a `CREATE`, `UPDATE`, `DELETE`, or `CONNECT` operation. In our example, a `Deployment` resource with a `subResource` of `Scale` is being created to scale our `Deployment` up in the `my-namespace` namespace.
- **Submitter identifiers:** The `userInfo` object identifies the user and groups of the submitter. The submitter and the user who created the original request are not always the same. For instance, if a user creates a `Deployment`, then the `userInfo` object won't be for the user who created the original `Deployment`; it will be for the `ReplicaSet` controller's service account because the `Deployment` creates a `ReplicaSet` that creates the pod. In our example, a user with the `uid` of `admin` submitted the scaling request.

- **Object:** object represents the JSON of the object being submitted, whereas oldObject represents what is being replaced if this is an update. Finally, options specifies additional options for the request. In our example, the new pod with the new number of replicas after the scaling operation is submitted.

The response from the webhook will simply have two attributes, the original uid from the request and allowed, which can be true or false. For instance, to allow our scaling operation to complete:

```
{  
  "uid": "705ab4f5-6393-11e8-b7cc-42010a800002"  
  "allowed": true  
}
```

The userInfo object can create complications quickly. Since Kubernetes often uses multiple layers of controllers to create objects, it can be difficult to track usage creation based on a user who interacts with the API server.

It's much better to authorize based on objects in Kubernetes, such as namespace labels or other objects.

A common use case is to allow developers to have a **sandbox** that they are administrators in, but that has very limited capacity. Instead of trying to validate the fact that a particular user doesn't try to request too much memory, annotate a personal namespace with a limit so that the admission controller has something concrete to reference regardless of whether the user submits a pod or a Deployment. This way, the policy will check the annotation on the namespace instead of the individual user. To ensure that only the user who owns the namespace is able to create something in it, use RBAC to limit access.

One final point on generic validating webhooks: there is no way to specify a key or password. It's an anonymous request. While, in theory, a validating webhook could be used to implement updates to your cluster, it is not recommended. For instance, you could use a validating webhook to create a ClusterRoleBinding when creating a Namespace, but that would mean that your policy check is not repeatable. It's best to separate policy checking and workflow.

Now that we've covered how Kubernetes implements dynamic access controllers, we'll look at one of the most popular options in OPA.

## What is OPA and how does it work?

OPA is a lightweight authorization engine that fits well in Kubernetes. It didn't get its start in Kubernetes, but it's certainly found a home there. There's no requirement to build dynamic admission controllers in OPA, but it's very good at it and there are extensive resources and existing policies that can be used to start your policy library.

This section provides a high-level overview of OPA and its components with the rest of the chapter getting into the details of an OPA implementation in Kubernetes.

### OPA architecture

OPA comprises three components – the HTTP listener, the policy engine, and the database:

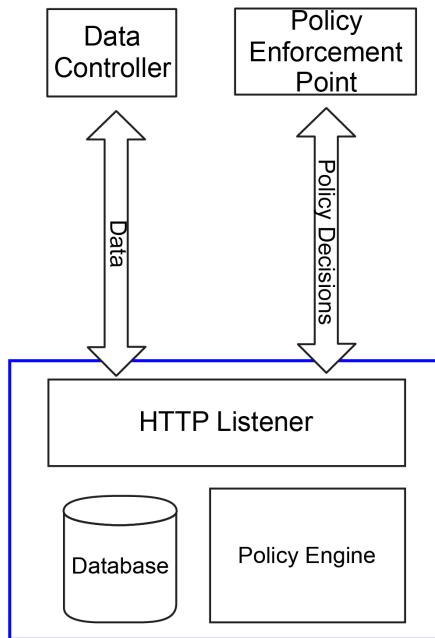


Figure 11.1: OPA architecture

The database used by OPA is in memory and ephemeral. It doesn't persist information used to make policy decisions. On the one hand, this makes OPA very scalable since it is essentially an authorization microservice. On the other hand, this means that every instance of OPA must be maintained on its own and must be kept in sync with authoritative data:

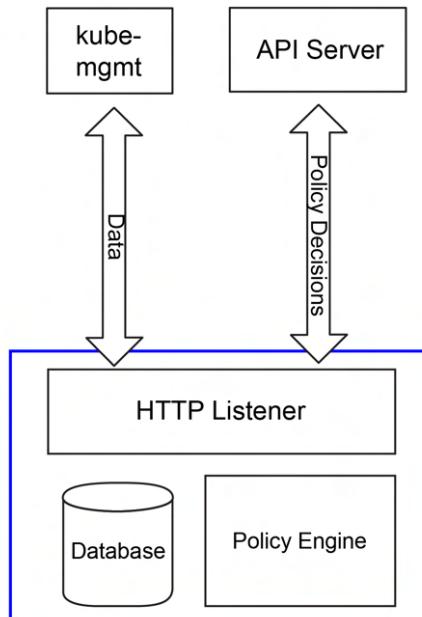


Figure 11.2: OPA in Kubernetes

When used in Kubernetes, OPA populates its database using a sidecar, called `kube-mgmt`, which sets up watches on the objects you want to import into OPA. As objects are created, deleted, or changed, `kube-mgmt` updates the data in its OPA instance. This means that OPA is “eventually consistent” with the API server, but it won’t necessarily be a real-time representation of the objects in the API server. Since the entire etcd database is essentially being replicated over and over again, great care needs to be taken in order to refrain from replicating sensitive data, such as `Secrets`, in the OPA database.

Now, let’s get introduced to the OPA policy language, Rego.

## Rego, the OPA policy language

We’ll cover the details of Rego in the next section in detail. The main point to mention here is that **Rego is a policy evaluation language**, not a generic programming language. Rego can be difficult for developers who are used to languages such as Golang, Java, or JavaScript, which support complex logic such as iterators and loops. Rego is designed to evaluate policy and is streamlined as such. For instance, if you wanted to write code in Java to check that all the container images in a pod started with one of a list of registries, it would look something like the following:

```
public boolean validRegistries(List<Container> containers, List<String>
allowedRegistries) {
    for (Container c : containers) {
        boolean imagesFromApprovedRegistries = false;
        for (String allowedRegistry : allowedRegistries) {
            imagesFromApprovedRegistries =
                imagesFromApprovedRegistries ||
                c.getImage().
startsWith(allowedRegistry);
        }
        if (! imagesFromApprovedRegistries) {
            return false;
        }
    }
    return true;
}
```

This code iterates over every container and every allowed registry to make sure that all of the images conform to the correct policy. The same code in Rego is much smaller:

```
invalidRegistry {
    ok_images = [image | startswith(input_images[j], input.parameters.
registries[_]) ; image = input_images[j] ]
    count(ok_images) != count(input_images)
}
```

The preceding rule will evaluate to true if any of the images on the containers come from unauthorized registries. We'll cover the details of how this code works later in the chapter. The key to understanding why this code is so much more compact is that much of the boilerplate of loops and tests is inferred in Rego. The first line generates a list of conforming images, and the second line makes sure that the number of conforming images matches the number of total images. If they don't match, then one or more of the images must come from invalid registries. The ability to write compact policy code is what makes Rego so well suited to admission controllers.

So far, we've focused on generic OPA and Rego. In the early days, you would integrate Kubernetes directly into OPA using ConfigMaps to store policies; however, this proved to be really unwieldy. Microsoft developed a tool called Gatekeeper, which is Kubernetes native and makes it easier to get the most out of OPA in Kubernetes. So, now, let's get introduced to Gatekeeper.

## Gatekeeper

Thus far, everything discussed has been generic to OPA. It was mentioned at the beginning of the chapter that OPA didn't get its start in Kubernetes. Early implementations had a sidecar that kept the OPA database in sync with the API server, but you had to manually create policies as ConfigMap objects and manually generate responses for webhooks. In 2018, Microsoft debuted Gatekeeper (<https://github.com/open-policy-agent/gatekeeper>) to provide a Kubernetes-native experience.

In addition to moving from ConfigMap objects to proper custom resources, Gatekeeper adds an audit function that lets you test policies against existing objects. If an object violates a policy, then a violation entry is created to track it. This way, you can get a snapshot of the existing policy violations in your cluster or know whether something was missed during Gatekeeper downtime due to an upgrade.

A major difference between Gatekeeper and generic OPA is that in Gatekeeper, OPA's functionality is not exposed via an API anyone can call. OPA is embedded, with Gatekeeper calling OPA directly to execute policies and keep the database up to date. Decisions can only be made based on data in Kubernetes or by pulling data at evaluation time.

## Deploying Gatekeeper

The examples that will be used will assume the use of Gatekeeper instead of a generic OPA deployment.

First, create a new cluster to deploy Gatekeeper into:

```
$ cd chapter2  
$ ./create-cluster.sh
```

Once the new cluster is running, based on the directions from the Gatekeeper project, use the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/  
gatekeeper/master/deploy/gatekeeper.yaml
```

This launches the Gatekeeper namespace pods and creates the validating webhook. Once deployed, move on to the next section. We'll cover the details of using Gatekeeper throughout the rest of this chapter.

## Automated testing framework

OPA has a built-in automated testing framework for your policies. This is one of the most valuable aspects of OPA. Being able to test policies consistently before deployment can save you hours of debugging time. When writing policies, have a file with the same name as your policies file, but with `_test` in the name. For instance, to have test cases associated with `mypolicies.rego`, have the test cases in `mypolicies_test.rego` in the same directory. Running the `opa test` will then run your test cases. We'll show how to use this to debug your code in the next section.

Having covered the basics of OPA and how it is constructed, the next step is to learn how to use Rego to write policies.

## Using Rego to write policies

Rego is a language specifically designed for policy writing. It is different from most languages you have likely written code in. Typical authorization code will look something like the following:

```
//assume failure
boolean allowed = false;
//on certain conditions allow access
if (someCondition) {
    allowed = true;
}
//are we authorized?
if (allowed) {
    doSomething();
}
```

Authorization code will generally default to unauthorized, with a specific condition having to happen in order to allow the final action to be authorized. Rego takes a different approach. Rego is generally written to authorize everything unless a specific set of conditions happens.

Another major difference between Rego and more general programming languages is that there are no explicit `if/then/else` control statements. When a line of Rego is going to make a decision, the code is interpreted as “If this line is false, stop execution.” For instance, the following code in Rego says “If the image starts with `myregistry.lan/`, then stop the execution of the policy and pass this check; otherwise, generate an error message”:

```
not startsWith(image, "myregistry.lan/")
msg := sprintf("image '%v' comes from untrusted registry", [image])
```

The same code in Java might look as follows:

```
if (! image.startsWith("myregistry.lan/")) {
    throw new Exception("image " + image + " comes from untrusted registry");
}
```

This difference between inferred control statements and explicit control statements is often the steepest part of the learning curve when learning Rego. While this can produce a steeper learning curve than other languages, Rego more than makes up for it by making it easy to test and build policies in an automated and manageable way. Another benefit of Rego is that it can be used for application-level authorizations. We'll cover this more when we get to Istio later in the book.

OPA can be used to automate the testing of policies. This is incredibly important when writing code that the security of your cluster relies upon. Automating your testing will help speed up your development and will increase your security by catching any bugs introduced into previously working code by means of new working code. Next, let's work through the life cycle of writing an OPA policy, testing it, and deploying it to our cluster.

## Developing an OPA policy

A common example of using OPA is to limit which registries a pod can come from. This is a common security measure in clusters to help restrict which pods can run on a cluster. For instance, we've mentioned Bitcoin miners a few times. If the cluster won't accept pods except from your own internal registry, then that's one more step that needs to be taken for a bad actor to abuse your cluster. First, let's write our policy, taken from the OPA documentation website (<https://www.openpolicyagent.org/docs/latest/kubernetes-introduction/>):

```
package k8sallowedregistries
invalidRegistry {
    input_images[image]
    not startswith(image, "quay.io/")
}
input_images[image] {
    image := input.review.object.spec.containers[_.].image
}
input_images[image] {
    image := input.review.object.spec.template.spec.containers[_.].image
}
```

The first line in this code declares the package our policy is in. Everything is stored in OPA in a package, both data and policies.

Packages in OPA are like directories on a filesystem. When you place a policy in a package, everything is relative to that package. In this case, our policy is in the `k8sallowedregistries` package.

The next section defines a rule. This rule ultimately will be undefined if our pod has an image that comes from `quay.io`. If the pod doesn't have an image from `quay.io`, the rule will return `true`, signifying that the registry is invalid. Gatekeeper will interpret this as a failure and return `false` to the API server when the pod is evaluated during a dynamic admission review.

The next two rules look very similar. The first of the `input_images` rules says “Evaluate the calling rule against every `container` in the object’s `spec.container`,” matching pod objects directly submitted to the API server and extracting all the `image` values for each container. The second `input_images` rule states “Evaluate the calling rule against every `container` in the object’s `spec.template.spec.containers`” to short circuit Deployment objects and StatefulSets.

Finally, we add the rule that Gatekeeper requires to notify the API server of a failed evaluation:

```
violation[{"msg": msg, "details": {}}] {
    invalidRegistry
    msg := "Invalid registry"
}
```

This rule will return an empty `msg` if the registry is valid. It’s a good idea to break up your code into code that makes policy decisions and code that responds with feedback. This makes it easier to test, which we’ll do next.

## Testing an OPA policy

Once we have written our policy, we want to set up an automated test. Just as with testing any other code, it’s important that your test cases cover both expected and unexpected input. It’s also important to test both positive and negative outcomes. It’s not enough to corroborate that our policy allowed a correct registry; we also need to make sure it stops an invalid one. Here are eight test cases for our code:

```
package k8sallowedregistries
test_deployment_registry_allowed {
    not invalidRegistry with input as {"apiVersion": ...
}
test_deployment_registry_not_allowed {
    invalidRegistry with input as {"apiVersion": ...
}
test_pod_registry_allowed {
    not invalidRegistry with input as {"apiVersion": ...
}
test_pod_registry_not_allowed {
    invalidRegistry with input as {"apiVersion": ...
}
test_cronjob_registry_allowed {
    not invalidRegistry with input as {"apiVersion": ...
}
test_cronjob_registry_not_allowed {
    invalidRegistry with input as {"apiVersion": ...
}
test_error_message_not_allowed {
```

```
control := {"msg": "Invalid registry", "details": {}}
result = violation with input as {"apiVersion": "admissi...
result[_] == control
}
test_error_message_allowed {
    result = violation with input as {"apiVersion": "admissi...
    result == set()
}
```

There are eight tests in total: two tests to make sure that the proper error message is returned when there's an issue, and six tests covering two use cases for three input types. We're testing simple pod definitions, Deployment, and CronJob. To validate success or failure as expected, we have included definitions that have `image` attributes that include `docker.io` and `quay.io` for each input type. The code is abbreviated for print but can be downloaded from <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter11/simple-opa-policy/rego/>.

To run the tests, first, install the OPA command-line executable as per the OPA website: <https://www.openpolicyagent.org/docs/latest/#running-opa>. Once it has been downloaded, go to the `simple-opa-policy/rego` directory and run the tests:

```
$ opa test .
data.kubernetes.admission.test_cronjob_registry_not_allowed: FAIL (248ns)
-----
PASS: 7/8
FAIL: 1/8
```

Seven of the tests passed, but `test_cronjob_registry_not_allowed` failed. The CronJob submitted as `input` should not be allowed because its `image` uses `docker.io`. The reason it snuck through was that CronJob objects follow a different pattern to Pods and Deployments, so our two `input_image` rules won't load any of the container objects from the CronJob. The good news is that when the CronJob ultimately submits the pod, Gatekeeper will not validate it, thereby preventing it from running. The bad news is that no one will know this until the pod is supposed to be run. Making sure we pick up CronJob objects in addition to our other objects with containers in them will make it much easier to debug because the CronJob won't be accepted.

To get all tests passing, add a new `input_container` rule to the `limitregistries.rego` file in the GitHub repo that will match the container used by a CronJob:

```
input_images[image] {
    image := input.review.object.spec.jobTemplate.spec.template.spec.
    containers[_].image
}
```

Now, running the tests will show that everything passes:

```
$ opa test .
PASS: 8/8
```

With a policy that has been tested, the next step is to integrate the policy into Gatekeeper.

## Deploying policies to Gatekeeper

The policies we've created need to be deployed to Gatekeeper, which provides Kubernetes custom resources that policies need to be loaded into. The first custom resource is `ConstraintTemplate`, which is where the Rego code for our policy is stored. This object lets us specify parameters in relation to our policy enforcement, and we'll cover this next. To keep things simple, create a template with no parameters:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sallowedregistries
spec:
  crd:
    spec:
      names:
        kind: K8sAllowedRegistries
      validation: {}
  targets:
    - target: admission.k8s.gatekeeper.sh
      rego: |
        package k8sallowedregistries
        .
        .
        .
```

The entire source code for this template is available at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/blob/main/chapter11/simple-opa-policy.yaml/gatekeeper-policy-template.yaml>.

Once created, the next step is to apply the policy by creating a constraint based on the template. Constraints are objects in Kubernetes based on the configuration of `ConstraintTemplate`. Notice that our template defines a custom resource definition. This gets added to the `constraints.gatekeeper.sh` API group. If you look at the list of CRDs on your cluster, you'll see `k8sallowedregistries` listed:

<code>constrainttemplatepodstatuses.status.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>constrainttemplates.templates.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>expansiontemplate.expansion.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>expansiontemplatepodstatuses.status.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>felixconfigurations.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>globalnetworkpolicies.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>globalnetworksets.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>hostendpoints.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>imagesets.operator.tigera.io</code>	<code>2024-08-19T12:32:41Z</code>
<code>installations.operator.tigera.io</code>	<code>2024-08-19T12:32:41Z</code>
<code>ipamblocks.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>ipamconfigs.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>ipamhandles.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>ippools.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>ipreservations.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>k8sallowedregistries.constraints.gatekeeper.sh</code>	<code>2024-08-19T12:35:29Z</code>
<code>kubecontrollersconfigurations.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>modifyset.mutations.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>mutatorpodstatuses.status.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>networkpolicies.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>networksets.crd.projectcalico.org</code>	<code>2024-08-19T12:32:40Z</code>
<code>providers.externaldata.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>
<code>syncsets.syncset.gatekeeper.sh</code>	<code>2024-08-19T12:34:14Z</code>

Figure 11.3: CRD created by `ConstraintTemplate`

Creating the constraint means creating an instance of the object defined in the template.

To keep from causing too much havoc in our cluster, we're going to restrict this policy to the `testpolicy` namespace:

```
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRegistries
metadata:
  name: restrict-openunison-registries
spec:
  match:
    kinds:
      - apiGroups: [""]
```

```
    kinds: ["Pod"]
- apiGroups: ["apps"]
  kinds:
  - StatefulSet
  - Deployment
- apiGroups: ["batch"]
  kinds:
  - CronJob
namespaces: ["testpolicy"]
```

The constraint limits the policy we wrote to just Deployment, CronJob, and Pod objects in the `testpolicy` namespace. Once our policy is created, if we try to create a pod in the `testpolicy` namespace that comes from `docker.io`, it will fail because the image comes from `docker.io`, not `quay.io`, not `quay.io`. First, let's create our `testpolicy` namespace and an example Deployment that will violate this policy:

```
$ chapter11/simple-opa-policy.yaml
$ kubectl create ns testpolicy
$ kubectl create deployment nginx-prepolicy --image=docker.io/nginx/nginx-
ingress -n testpolicy
$ kubectl create -f ./gatekeeper-policy.yaml
$ kubectl create deployment nginx-withpolicy --image=docker.io/nginx/nginx-
ingress -n testpolicy
error: failed to create deployment: admission webhook "validation.gatekeeper.
sh" denied the request: [restrict-openunison-registries] Invalid registry
```

The last line tried to create a new Deployment that references `docker.io` instead of `quay.io`, which failed because our policy blocked it. But we also created a Deployment that violates this rule before deploying our policy, which means that our admission controller never received a create command. This is one feature of Gatekeeper over generic OPA that is very powerful: Gatekeeper audits your existing infrastructure against new policies. This way, you can find offending deployments quickly.

Next, look at the policy object. You will see that there are several violations in the `status` section of the object:

```
$ kubectl get k8sallowedregistries.constraints.gatekeeper.sh restrict-
openunison-registries -o json | jq -r '.status.violations'
[
  {
    "enforcementAction": "deny",
    "group": "",
    "kind": "Pod",
    "message": "Invalid registry",
    "name": "nginx-prepolicy-8bd5cbfc9-szzs4",
    "namespace": "testpolicy",
    "version": "v1"
```

```

},
{
  "enforcementAction": "deny",
  "group": "apps",
  "kind": "Deployment",
  "message": "Invalid registry",
  "name": "nginx-prepolicy",
  "namespace": "testpolicy",
  "version": "v1"
}
]

```

Having deployed your first Gatekeeper policy, you may quickly notice it has a few issues. The first is that the registry is hardcoded. This means that we'd need to replicate our code for every change of registry. It's also not flexible for the namespace. As an example, Tremolo Security's images are across multiple `github.io` registries, so instead of limiting a specific registry server, we may want flexibility for each namespace and to allow multiple registries. Next, we'll update our policies to provide this flexibility.

## Building dynamic policies

Our current registry policy is limiting. It is static and only supports a single registry. Both Rego and Gatekeeper provide functionality to build a dynamic policy that can be reused in our cluster and configured based on individual namespace requirements. This gives us one code base to work from and debug instead of having to maintain repetitive code. The code we're going to use is at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter11/parameter-opa-policy>.

When inspecting `rego/limitregistries.rego`, the main difference between the code in `parameter-opa-policy` and `simple-opa-policy` comes down to the `invalidRegistry` rule:

```

invalidRegistry {
  ok_images = [image | startswith(input_images[i], input.parameters.
registries[_]) ; image = input_images[i] ]
  count(ok_images) != count(input_images)
}

```

The goal of the first line of the rule is to determine which images come from approved registries using a comprehension. Comprehensions provide a way to build out sets, arrays, and objects based on some logic. In this case, we want to only add images to the `ok_images` array that start with any of the allowed registries from `input.parameters.registries`.

To read a comprehension, start with the type of brace. Ours starts with a square bracket, so the result will be an array. Objects and sets can also be generated. The word between the open bracket and the pipe character (`|`) is called the head and this is the variable that will be added to our array if the right conditions are met. Everything to the right of the pipe character (`|`) is a set of rules used to determine what `image` should be and if it should have a value at all. If any of the statements in the rule resolve to `undefined` or `false`, the execution exits for that iteration.

The first rule of our comprehension is where most of the work is done. The `startswith` function is used to determine whether each of our images starts with the correct registry name. Instead of passing two strings to the function, we instead pass arrays. The first array has a variable we haven't declared yet, `i`, and the other uses an underscore (`_`) where the index would usually be. The `i` is interpreted by Rego as “*Do this for each value in the array, incrementing by 1, and let it be referenced throughout the comprehension.*” The underscore is shorthand in Rego for “*Do this for all values.*” Since we specified two arrays, every combination of the two arrays will be used as input to the `startswith` function.

That means that if there are two containers and three potential pre-approved registries, then `startswith` will be called six times. When any of the combinations return true from `startswith`, the next rule is executed. That sets the `image` variable to `input_image` with index `i`, which then means that the image is added to `ok_images`. The same code in Java would look something like the following:

```
ArrayList<String> okImages = new ArrayList<String>();
for (int i=0;i<inputImages.length;i++) {
    for (int j=0;j<registries.length;j++) {
        if (inputImages[i].startsWith(registries[j])) {
            okImages.add(inputImages[i]);
        }
    }
}
```

One line of Rego eliminated seven lines of mostly boilerplate code.

The second line of the rule compares the number of entries in the `ok_images` array with the number of known container images. If they are equal, we know that every container contains a valid image.

With our updated Rego rules for supporting multiple registries, the next step is to deploy a new policy template (if you haven't done so already, delete the old `k8sallowedregistries ConstraintTemplate` and `restrict-openunison-registries K8sAllowedRegistries`):

```
$ kubectl delete -f ./gatekeeper-policy.yaml k8sallowedregistries.constraints.
gatekeeper.sh "restrict-openunison-registries" deleted
$ kubectl delete -f ./gatekeeper-policy-template.yaml constrainttemplate.
templates.gatekeeper.sh "k8sallowedregistries" deleted
```

Here's our updated `ConstraintTemplate`:

```
apiVersion: templates.gatekeeper.sh/v1beta1
kind: ConstraintTemplate
metadata:
  name: k8sallowedregistries
spec:
  crd:
    spec:
      names:
```

```

kind: K8sAllowedRegistries
validation:
  openAPIV3Schema:
    properties:
      registries:
        type: array
        items: string
targets:
  - target: admission.k8s.gatekeeper.sh
rego: |
  package k8sallowedregistries
  .
  .
  .

```

Beyond including our new rules, the highlighted section shows that we added a schema to our template. This will allow for the template to be reused with specific parameters. This schema goes into the `CustomResourceDefinition` that will be created and is used to validate input for the `K8sAllowedRegistries` objects we'll create in order to enforce our pre-authorized registry lists. Create this new policy definition:

```
$ cd chapter11/parameter-opa-policy/yaml/
$ kubectl create -f gatekeeper-policy-template.yaml
```

Finally, let's create our policy for the `testpolicy` namespace. Since the only containers that are running in this namespace should come from NGINX's `docker.io` registry, we'll limit all pods to `docker.io/nginx/` using the following policy:

```

apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sAllowedRegistries
metadata:
  name: restrict-openunison-registries
spec:
  match:
    kinds:
      - apiGroups: []
        kinds: ["Pod"]
      - apiGroups: ["apps"]
        kinds:
          - StatefulSet
          - Deployment
      - apiGroups: ["batch"]
        kinds:

```

```
- CronJob
namespaces: ["testpolicy"]
parameters:
registries: ["docker.io/nginx/"]
```

Unlike our previous version, this policy specifies which registries are valid instead of embedding the policy data directly into our Rego. With our policies in place, let's try to run the `BusyBox` container in the `testpolicy` namespace to get a shell:

```
$ kubectl create -f ./gatekeeper-policy.yaml
$ kubectl run tmp-shell --rm -i --tty --image docker.io/busybox -n testpolicy
-- /bin/bash
Error from server (Forbidden): admission webhook "validation.gatekeeper.sh"
denied the request: [restrict-openunison-registries] Invalid registry
$ kubectl create deployment nginx-withpolicy --image=docker.io/nginx/nginx-
ingress -n testpolicy
deployment.apps/nginx-withpolicy created
```

In the above example, we were able to stop the `BusyBox` container from being deployed, but the NGINX Deployment was created because we were able to restrict the specific container registry on `docker.io`.

Using this generic policy template, we can restrict which registries the namespaces are able to pull from. As an example, in a multi-tenant environment, you may want to restrict all pods to the owner's own registry. If a namespace is being used for a commercial product, you can stipulate that only that vendor's containers can run in it. Before moving on to other use cases, it's important to understand how to debug your code and handle Rego's quirks.

## Debugging Rego

Debugging Rego can be challenging. Unlike more generic programming languages such as Java or Go, there's no way to step through code in a debugger. Take the example of the generic policy we just wrote for checking registries. All the work was done in a single line of code. Stepping through it wouldn't do much good.

To make Rego easier to debug, the OPA project provides a trace of all failed tests when verbose output is set on the command line. This is another great reason to use OPA's built-in testing tools.

To make better use of this trace, Rego has a function called `trace` that accepts a string. Combining this function with `sprintf` lets you more easily track where your code is not working as expected. In the `chapter11/parameter-opa-policy-fail/rego` directory, there's a test that will fail. There is also an `invalidRegistry` rule with multiple trace options added:

```
invalidRegistry {
  trace(sprintf("input_images : %v",[input_images]))
  ok_images = [image |
    trace(sprintf("image %v",[input_images[j]]))
    startswith(input_images[j],input.parameters.registries[_]) ;
```

```

    image = input_images[j]
]
trace(sprintf("ok_images %v", [ok_images]))
trace(sprintf("ok_images size %v / input_images size %v", [count(ok_
images), count(input_images)]))
count(ok_images) != count(input_images)
}

```

When the test is run, OPA will output a detailed trace of every comparison and code path. Wherever it encounters the `trace` function, a “note” is added to the trace. This is the equivalent of adding `print` statements in your code to debug. The output of the OPA trace is very verbose, and far too much text to include in print. Running `opa test . -v` in this directory will give you the full trace you can use to debug your code.

## Using existing policies

Before moving into more advanced use cases for OPA and Gatekeeper, it’s important to understand the implications of how OPA is built and used. If you inspect the code we worked through in the previous section, you might notice that we aren’t checking for an `initContainer`. We’re only looking for the primary containers. An `initContainer` is a special container that is run before the containers listed in a pod are expected to end. They’re often used to prepare the filesystem of a volume mount and for other “initial” tasks that should be performed before the containers of a pod have run. If a bad actor tried to launch a pod with an `initContainer` that pulls in a Bitcoin miner (or worse), our policy wouldn’t stop it.

It’s important to be very detailed in the design and implementation of policies. One of the ways to make sure you’re not missing something when building policies is to use policies that already exist and have been tested. The Gatekeeper project maintains several libraries of pre-tested policies and how to use them in its GitHub repo at <https://github.com/open-policy-agent/gatekeeper-library>. Before attempting to build one of your own policies, see whether one already exists there first.

This section provided an overview of Rego and how it works in policy evaluation. It didn’t cover everything, but should give you a good point of reference for working with Rego’s documentation. Next, we’ll learn how to build policies that rely on data from outside our request, such as other objects in our cluster.

## Enforcing Ingress policies

So far in this chapter, we’ve built policies that are self-contained. When checking whether an image is coming from a pre-authorized registry, the only data we needed was from the policy and the containers. This is often not enough information to make a policy decision. In this section, we’ll work on building a policy that relies on other objects in your cluster to make policy decisions.

Before diving into the implementation, let’s talk about the use case. It’s common to limit which namespaces can have Ingress objects. If a namespace hosts a workload that doesn’t require any inbound access, why allow an Ingress object at all? You may think you can enforce this using RBAC by limiting what tenants are allowed to deploy using a Role and RoleBinding, but this has some limitations:

- The `admin` and `edit` `ClusterRoles` are default aggregate `ClusterRoles`, so you would need to create a new `ClusterRole` that enumerates all objects except `Ingress` that you want your namespace admin to be able to create.
- If your new `ClusterRole` included `RoleBindings`, your namespace owner could just grant themselves `Ingress` creation.

Using an admission controller with an annotation or a label is a good approach to enforcing if the namespace can have an `Ingress` in it. The `Namespace` object is cluster-scoped, so an `admin` won't be able to elevate their privileges in the namespace and add the label.

In our next example, we'll write a policy that only allows `Ingress` objects in namespaces that have the correct label. The pseudo-code would look something like this:

```
if (! hasIngressAllowedLabel(input.review.object.metadata.namespace)) {  
    generate error;  
}
```

The hard part here is understanding if the namespace has a label. Kubernetes has an API, which you could query, but that would mean either embedding a secret into the policy so it can talk to the API server or allowing anonymous access. Neither of those options is a good idea. Another issue with querying the API server is that it's difficult to automate testing since you are now reliant on an API server being available wherever you run your tests.

We discussed earlier that OPA can replicate data from the API server in its own database. Gatekeeper uses this functionality to create a `cache` of objects that can be tested against. Once this cache is populated, we can replicate it locally to provide test data for our policy testing.

## Enabling the Gatekeeper cache

The Gatekeeper cache is enabled by creating a `Config` object in the `gatekeeper-system` namespace. Add this configuration to your cluster:

```
apiVersion: config.gatekeeper.sh/v1alpha1  
kind: Config  
metadata:  
  name: config  
  namespace: "gatekeeper-system"  
spec:  
  sync:  
    syncOnly:  
      - group: ""  
        version: "v1"  
        kind: "Namespace"
```

```
$ cd chapter11/enforce-ingress/yaml/  
$ kubectl create -f ./config.yaml
```

This will begin replicating Namespace objects in Gatekeeper's internal OPA database. Let's create a Namespace with a label allowing Ingress objects and without a label allowing Ingress objects:

```
apiVersion: v1
kind: Namespace
metadata:
  name: ns-with-ingress
  labels:
    allowing ingress: "true"
spec: {}
---
apiVersion: v1
kind: Namespace
metadata:
  name: ns-without-ingress
spec: {}
```

After a moment, the data should be in the OPA database and ready to query.

The Gatekeeper service account has read access to everything in your cluster with its default installation. This includes secret objects. Be careful what you replicate in Gatekeeper's cache as there are no security controls from inside a Rego policy. Your policy could very easily log secret object data if you are not careful. Also, make sure to control who has access to the `gatekeeper-system` namespace. Anyone who gets hold of the service account's token can use it to read any data in your cluster.

Now that we have Gatekeeper set up and ready to start enforcing policies, how do we test policies? We could test them directly against a cluster, but that will slow down our development cycle. Next, we'll see how to mock test data so that we can automate our testing outside of a Kubernetes cluster.

## Mocking up test data

In order to automate the testing of our policy, we need to create test data. In the previous examples, we used data injected into the `input` variable. Cache data is stored in the `data` variable. Specifically, in order to access our namespace labels, we need to access `data.inventory.cluster["v1"].Namespace["ns-with-ingress"].metadata.labels`. This is the standard way for you to query cluster data from Rego in Gatekeeper. If you want to query objects inside of a namespace, it would instead look like `data.inventory.namespace["myns"]["v1"]["ConfigMaps"]["myconfigmap"]`. Just as we did with the input, we can inject a mocked-up version of this data by creating a `data` object. Here's what our JSON will look like:

```
{
  "cluster": {
    "v1": {
      "Namespace": {
        "ns-with-ingress": {}}
```

```
    "metadata": {
        "labels": {
            "allowingress": "true"
        }
    },
    "ns-without-ingress": {
        "metadata": {
    }}}}}}}
```

When you look at `chapter11/enforce-ingress/rego/enforceingress_test.rego`, you'll see the tests have `with input as {...}` with `data as {...}` with the preceding document as our control data. This lets us test our policies with data that would exist in GateKeeper without having to deploy our code in a cluster.

## Building and deploying our policy

Just as before, we've written test cases prior to writing our policy. Next, we'll examine our policy:

```
package k8senforceingress

violation[{"msg":msg, "details":{}}] {
    missingIngressLabel
    msg := "Missing label allowingress: \"true\""
}

missingIngressLabel {
    data.inventory.cluster["v1"].Namespace[input.review.object.metadata.namespace].
    metadata.labels["allowingress"] != "true"
}

missingIngressLabel {
    not data.inventory.cluster["v1"].Namespace[input.review.object.metadata.
    namespace].metadata.labels["allowingress"]
}
```

This code should look familiar. It follows a similar pattern as our earlier policies. The first rule, `violation`, is the standard reporting rule for Gatekeeper. The second and third rules have the same name but different logic. This is because Rego evaluates all of the statements in a rule as an AND, so for the rule to be true, all of the statements need to be true. If we only had the first `missingIngressLabel` rule, which checks if the `allowingress` label is true, then Ingress objects without this label at all would break the rule and so bypass our requirement. We could have a rule that requires that the label be set, but that would lead to a bad user experience. It would be better to set up our policy so that it will fail if the label isn't true OR the label isn't set at all.

To set up the logic of “if the label’s value is not true or the label is not present,” we need to have two rules with the same name. One rule checks for the label’s value, and the other validates if the label is there at all. Rego will execute both `missingIngressLabel` rules and, as long as one passes, execution will continue. In our case, if the namespace the `Ingress` object is created in doesn’t have the correct value of `allowingress`, or doesn’t have the label `allowingress` at all, the violation rule will complete, returning an error to the user.

This is a key difference between Rego and other languages. Rego isn’t executed in sequence the way Java, Go, or JavaScript is. It’s a policy language that’s evaluated, so the execution path is different. When writing Rego, it’s important to remember you’re not working with a typical programming language.

To deploy, add `chapter11/enforce-ingress/yaml/gatekeeper-policy-template.yaml` and `chapter11/enforce-ingress/yaml/gatekeeper-policy.yaml` to your cluster.

To test, we’ll try creating an `Ingress` object in the `ns-without-ingress` namespace:

```
$ kubectl create ingress test --rule="foo.com/bar=svc1:8080,tls=my-cert" -n ns-without-ingress
error: failed to create ingress: admission webhook "validation.gatekeeper.sh" denied the request: [require-ingress-label] Missing label allowingress: "true"
```

You can see that our policy blocked the `Ingress` object’s creation. Next, we’ll try to create the same `Ingress` object but in the `ns-with-ingress` namespace, which has the correct label:

```
$ kubectl create ingress test --rule="foo.com/bar=svc1:8080,tls=my-cert" -n ns-with-ingress
ingress.networking.k8s.io/test created
```

This time, our policy allowed the `Ingress` object to be created!

Most of this chapter has been spent writing policies. Next, we’ll cover how to provide sane defaults to your objects using mutating webhooks.

## Mutating objects and default values

Until this point, everything we have discussed has been about how to use Gatekeeper to enforce a policy. Kubernetes has another feature called mutating admission webhooks that allows a webhook to change, or mutate, an object before the API server processes it and runs validating admission controllers.

A common usage of a mutating webhook is to explicitly set security context information on pods that don’t have it set. For instance, if you create a pod with no `spec.securityContext.runAsUser`, then the pod will run as the user the Docker container was built to run using the `USER` directive (or root by default) when it was built. This is insecure since it means you could be running as root, especially if the container in question is from Docker Hub. While you can have a policy that blocks running as root, you could also have a mutating webhook that will set a default user ID if it’s not specified to make it a default. This makes for a better developer experience because, now, as a developer, I don’t have to worry about which user my container was built to run as so long as it was designed to work with any user.

This brings up a common question of defaults versus explicit configuration. There are two schools of thought. The first is that you should provide sane defaults wherever possible to minimize what developers have to know to get a typical workload running. This creates consistency and makes it easier to spot outliers. The other school of thought requires explicit configuration of security contexts so that it's known looking at a glance what the workload expects. This can make auditing easier, especially if paired with GitOps to manage your manifests, but creates quite a bit of repetitive YAML.

I'm personally a fan of sane defaults. The vast majority of workloads will not require any privilege and should be treated as such. It doesn't mean you don't still need enforcement, just that it's a better experience for your developers. It also makes it easier to make global changes. Want to change the default user ID or security context? You make the change in your mutating webhook instead of across tens, hundreds, or even thousands of manifests. Most of Kubernetes is built this way. You don't create pod objects directly; you create Deployments and StatefulSets with controllers that create pods. Going back to our discussions on RBAC, aggregate roles work this way too. Instead of creating a massive ClusterRole for namespace administrators, Kubernetes uses a controller to generate the ClusterRole dynamically based on label selectors, making it easier to maintain. In my experience, this example should be applied to security defaults as well.

Gatekeeper's mutation isn't built on Rego the way its validation policies are. While you can write mutating webhooks in Rego, and I can say this from experience, it's not well suited to it. What makes Rego a great policy definition language also makes it very hard to build mutations.

Now that we know what mutations are useful and that we can use Gatekeeper, let's build a mutation that will configure all containers to run as a default user if none is specified.

First, let's deploy something that we can test our mutations:

```
$ kubectl create ns test-mutations
$ kubectl create deployment test-nginx --image=ghcr.io/openunison/openunison-
k8s-html:latest -n test-mutations
```

Now, we can deploy the policy in `chapter11/defaultUser/addDefaultUser.yaml`:

```
apiVersion: mutations.gatekeeper.sh/v1
kind: Assign
metadata:
  name: default-user
spec:
  applyTo:
    - groups: [""]
      kinds: ["Pod"]
      versions: ["v1"]
  match:
    scope: Namespaced
    excludedNamespaces:
      - kube-system
```

```

location: "spec.securityContext.runAsUser"
parameters:
  assign:
    value: 70391
  pathTests:
    - subPath: "spec.securityContext.runAsUser"
      condition: MustNotExist

```

Let's walk through this mutation. The first part of the `spec`, `applyTo`, tells Gatekeeper what objects you want this mutation to act on. For us, we want it to work on all pods.

The next section, `match`, gives you the chance to specify conditions on which pods we want the mutation to apply to. In our case, we're applying to all of them except in the `kube-system` namespace. In general, I tend to avoid making changes to anything in the `kube-system` namespace because it's the domain of whoever is managing your clusters.

Making changes there can have permanent impacts on your cluster. In addition to specifying which namespaces you don't want to apply your mutation to, you can also specify additional conditions:

- `kind` – What kind of object to match on
- `labelSelectors` – Labels on the object that must match
- `namespaces` – List of namespaces to apply the mutation policy to
- `namespaceSelector` – Labels on the container namespaces

We'll talk more about label matching in *Chapter 12, Node Security with GateKeeper*.

After defining how to match objects to mutate, we specify what mutation to perform. For us, we want to set `spec.securityContext.runAsUser` to a randomly chosen user ID if one isn't specified. The last part, `pathTests`, is what lets us set this value if the `spec.securityContext.runAsUser` isn't already set.

Once you've applied your mutation policy, verify that the test pod isn't running as a specific user:

```
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
  securityContext}' -n test-mutations
{}
```

Now, delete the pod and check again:

```
$ kubectl delete pods -l app=test-nginx -n test-mutations
pod "test-nginx-f6c8578fc-qkd5h" deleted
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
  securityContext}' -n test-mutations
{"runAsUser":70391}
```

Our pod is now running as user 70391! Now, let's edit our deployment so that the user is set the user identity:

```
$ kubectl patch deployment test-nginx --patch
'{"spec":{"template":{"spec":{"securityContext":{"runAsUser":19307}}}}}' -n
test-mutations
deployment.apps/test-nginx patched
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
securityContext}' -n test-mutations
{"runAsUser":19307}
```

Our mutation didn't apply because we already had a user specified in our Deployment object.

One last note on setting values: you'll often find that you want to set a value for an object in a list. For instance, you may want to create a policy that will set any container as unprivileged unless specifically set to be privileged. In `chapter11/defaultUser/yaml/setUnprivileged.yaml`, our location (and subPath) have changed:

```
location: "spec.containers[image:*].securityContext.privileged"
```

This reads as “Match all objects in the list `spec.containers` that have an attribute called `image`.” Since every container must have an image, this will match all containers. Apply this object and test it out again on the test pod:

```
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
containers[0].securityContext}' -n test-mutations
$ kubectl delete pods -l app=test-nginx -n test-mutations
pod "test-nginx-ccf9bfcd-wt97v" deleted
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
containers[0].securityContext}' -n test-mutations
{"privileged":false}
```

Now our pod is marked as unprivileged!

In this section, we looked at how you can set defaults using Gatekeeper's built-in mutation support. We discussed the benefits of mutating webhooks that set defaults, enabled Gatekeeper's support for mutations, and built policies that set a default user identity and disable privileged containers. Using what you've learned in this section, you can use Gatekeeper not only to enforce your policies but also to set sane defaults to make compliance easier for your developers. Using Gatekeeper for policy management is great, but it does require additional skills and the management of an additional system. Next, we'll learn how to create policies with alternatives to Rego or using Kubernetes' new built-in policy engine.

## Creating policies without Rego

Rego is a very powerful way to build complex policies that are then implemented by the Gatekeeper project. That power comes with a steep learning curve and complexity. It may not be the right choice for you or your clusters. It isn't the only way to implement an admission controller. We're not going to go into too many details, as these other projects all have their own capabilities that are worth exploring and I won't be able to do them justice in one section.

The two most common alternatives to GateKeeper are:

- **Kyverno**: Kyverno is a specialized policy engine for Kubernetes. It's not designed as a generic authorization engine the way OPA is so it can make assumptions that provide a simpler experience for building Kubernetes policies (<https://kyverno.io/>).
- **jsPolicy**: The jsPolicy project allows you to build your policies in JavaScript or TypeScript instead of a **domain-specific language (DSL)** like Rego. The idea is that many of the quirks that come from Rego being a policy language, not a programming language, are eliminated by using a common language like JavaScript (<https://github.com/loft-sh/jspolicy>).

These projects both have their own strengths and I encourage you to evaluate them for your use cases. If your policies are straightforward and don't require the power of one of these engines, you can also look at Kubernetes' new built-in capabilities, which is what we'll cover next.

## Using Kubernetes' validating admission policies

In 1.28 Kubernetes, validating admission policies went into beta, which allows you to create simpler policies without an external admission controller. For simpler policies, this eliminates a component that needs to be deployed. We're not going to dive too deeply into building admission policies, but we wanted to give you an overview so that you have it as an option.

From a policy development perspective, the biggest difference between using Gatekeeper and validating admission policies is that while Gatekeeper uses Rego, validating admission policies use the **Common Expression Language (CEL)**. CEL is not a Turing Complete language, which means that it isn't as expressive and capable as JavaScript but is easier to secure. CEL is being integrated into multiple layers of Kubernetes. It's being used to provide more expressive validation for custom resource definitions and is also being integrated into the new authentication configuration options that are being developed. You can learn more about CEL at <https://github.com/google/cel-spec>.

From a capability perspective, you can use CEL to validate any of the data inside of the object to be created. There are two components to build a validating admission policy:

- **ValidatingAdmissionPolicy**: This is the object that describes the policy and the expressions to run as part of that policy. This is similar to **ConstraintTemplate** from Gatekeeper.
- **ValidatingAdmissionPolicyBinding**: This is how Kubernetes knows when to apply our **ValidatingAdmissionPolicy**.

To implement our example from above where we want to limit Ingress objects to namespaces with a specific label, first, we'd create the **ValidatingAdmissionPolicy** (`chapter11/enforce-ingress-vap/vap-ingress.yaml`):

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingAdmissionPolicy
metadata:
  name: "vap-ingress"
spec:
  failurePolicy: Fail
```

```
matchConstraints:
  resourceRules:
    - apiGroups: ["networking.k8s.io"]
      apiVersions: ["v1"]
      operations: ["CREATE", "UPDATE"]
      resources: ["ingresses"]
  validations:
    - expression: |-  
      namespaceObject.metadata.labels.allowingress == "true"
```

The above policy will fail if the namespace that the Ingress is added to doesn't have the label `allowingress` with a value of `true`. Next, we need to tell Kubernetes to bind our policy. We want this to apply to all namespaces, but similar to a policy implementation for Gatekeeper, we can specify specific namespaces or namespace labels. We do this using a `ValidatingAdmissionPolicy` (`chapter11/enforce-ingress-vap/vap-binding-ingress.yaml`):

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingAdmissionPolicyBinding
metadata:
  name: "vap-binding-ingress"
spec:
  policyName: "vap-ingress"
  validationActions: [Deny]
```

This binding will bind our policy to all namespaces, and on failure, deny the request. We could also warn the user or just audit the event. In our case, we want the request to fail. With these two objects created, we can try to create Ingress objects again:

```
$ kubectl create ingress test --rule="foo.com/bar=svc1:8080,tls=my-cert" -n ns-without-ingress
error: failed to create ingress: ingresses.networking.k8s.io "test" is forbidden: ValidatingAdmissionPolicy 'vap-ingress' with binding 'vap-binding-ingress' denied request: expression 'namespaceObject.metadata.labels.allowingress == "true"' resulted in error: no such key: allowingress
$ kubectl create ingress test --rule="foo.com/bar=svc1:8080,tls=my-cert" -n ns-with-ingress
ingress.networking.k8s.io/test created
```

Just as with our Gatekeeper examples, we see that we're able to deny the creation of an Ingress rule if our namespace doesn't have the appropriate label.

The addition of validating access policies to Kubernetes adds a powerful tool, but it does have its limits. It's tempting to say we'll use validating access policies for simple use cases, and Gatekeeper for more complex ones, but there are other things to keep in mind beyond the implementation complexity. For one, how are you monitoring failures? If you use both solutions, then even though you may have some simpler rule implementations, you'll need to audit both solutions, which will create more work.

While we introduced validating access policies so you're aware of their capability, we're going to continue to focus on OPA and Gatekeeper in future chapters. In the next chapter, we're going to apply what we've learned about OPA and Gatekeeper to help secure Kubernetes nodes.

## Summary

In this chapter, we explored how to use Gatekeeper as a dynamic admission controller to provide additional authorization policies on top of Kubernetes' built-in RBAC capabilities. We looked at how Gatekeeper and OPA are architected. Then, we learned how to build, deploy, and test policies in Rego. Finally, you were shown how to use Gatekeeper's built-in mutation support to create default configuration options in pods.

Extending Kubernetes' policies leads to a stronger security profile in your clusters and provides greater confidence in the integrity of the workloads you are running.

Using Gatekeeper can also help catch previously missed policy violations through its application of continuous audits. Using these capabilities will provide a stronger foundation for your cluster.

This chapter focused on whether or not to launch a pod based on our specific policies. In the next chapter, we'll learn how to protect your nodes from the processes running in those pods.

## Questions

1. Are OPA and Gatekeeper the same thing?
  - a. Yes
  - b. No
2. How is Rego code stored in Gatekeeper?
  - a. It is stored as ConfigMap objects that are watched.
  - b. Rego has to be mounted to the pod.
  - c. Rego needs to be stored as secret objects.
  - d. Rego is saved as a ConstraintTemplate.
3. How do you test Rego policies?
  - a. In production
  - b. Using an automated framework built directly into OPA
  - c. By first compiling to WebAssembly
4. In Rego, how do you write a for loop?
  - a. You don't need to; Rego will identify iterative steps.
  - b. By using the for all syntax.
  - c. By initializing counters in a loop.
  - d. There are no loops in Rego.

5. What is the best way to debug Rego policies?
  - a. Use an IDE to attach to the Gatekeeper container in a cluster.
  - b. In production.
  - c. Add trace functions to your code and run the `opa test` command with `-v` to see execution traces.
  - d. Include `System.out` statements.
6. Constraints all need to be hardcoded.
  - a. True
  - b. False
7. Gatekeeper can replace pod security policies.
  - a. True
  - b. False

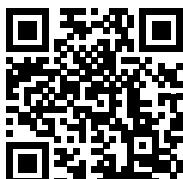
## Answers

1. b – No, Gatekeeper is a Kubernetes-native policy engine built on OPA.
2. d – Rego is saved as a `ConstraintTemplate`
3. b – Please don't test in production!
4. a – Everything is built on policy, not iterative control loops.
5. c – Add trace functions to your code and run the `opa test` command with `-v` to see execution traces
6. b – False. You can have variable constraints.
7. a – True, and we'll cover that in the next chapter!

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>





# 12

## Node Security with Gatekeeper

Most of the security discussed so far has focused on protecting Kubernetes APIs. **Authentication** has meant the authentication of API calls. **Authorization** has meant authorizing access to certain APIs. Even the discussion on the dashboard centered mostly around how to securely authenticate to the API server by way of the dashboard.

This chapter will be different, as we will now shift our focus to securing our nodes. We will learn how to use the **Gatekeeper** project to protect the nodes of a Kubernetes cluster. Our focus will be on how containers run on the nodes of your cluster and how to keep those containers from having more access than they should. We'll go into the details of impacts in this chapter, by looking at how exploits can be used to gain access to a cluster when nodes aren't protected. We'll also explore how these scenarios can be exploited even in code that doesn't need node access.

In this chapter, we will cover the following topics:

- Technical requirements
- What is node security?
- Enforcing node security with Gatekeeper
- Using pod security standards to enforce node security

By the end of this chapter, you'll have a better understanding of how Kubernetes interacts with the nodes that run your workloads and how they can be better protected.

### Technical requirements

To complete the hands-on exercises in this chapter, you will require an Ubuntu 22.04 server.

You can access the code for this chapter in the following GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter12>.

## What is node security?

Each pod that is launched in your cluster runs on a node. That node could be a VM, a “bare metal” server, or even another kind of compute service that is itself a container. Every process started by a pod runs on that node and, depending on how it is launched, can have a surprising set of capabilities on that node, such as talking to the filesystem, breaking out of the container to get a shell on the node, or even accessing the secrets used by the node to communicate with the API server. It’s important to make sure that processes that are going to request special privileges do so only when authorized and, even then, for specific purposes.

Many people have experience with physical and virtual servers, and most know how to secure the workloads running on them. Containers need to be considered differently when you talk about securing each workload. To understand why Kubernetes security tools such as the **Open Policy Agent (OPA)** exist, you need to understand how a container is different from a **virtual machine (VM)**.

## Understanding the difference between containers and VMs

“*A container is a lightweight VM*” is often how containers are described to those new to containers and Kubernetes. While this makes for a simple analogy, from a security standpoint, it’s a dangerous comparison. A container at runtime is a process that runs on a node. On a Linux system, these processes are isolated by a series of Linux technologies that limit their visibility to the underlying system.

Go to any node in a Kubernetes cluster and run the `top` command, and all of the processes from containers will be listed. As an example, even though Kubernetes runs in KinD, running `ps -A -elf | grep java` will show the OpenUnison and operator container processes:

```
4 S k8s      1193507 1193486 1 80  0 - 3446501 -    Oct07 ?          06:50:33
java -classpath /usr/local/openunison/work/webapp/
WEB-INF/lib/*:/usr/local/openunison/work/webapp/WEB-INF/classes:/tmp/
quartz -Djava.awt.headless=true -Djava.security.egd=file:/dev/.urandom
-DunisonEnvironmentFile=/etc/openunison/ou.env -Djavax.net.ssl.trustStore=/
etc/openunison/cacerts.jks com.tremolosecurity.openunison.undertow.
OpenUnisonOnUndertow /etc/openunison/openunison.yaml
0 S k8s      2734580 2730582 0 80  0 - 1608 pipe_w 13:13 pts/0   00:00:00
grep --color=auto java
```

In contrast, a VM is, as the name implies, a complete virtual system. It emulates its own hardware, has an isolated kernel, and so on. The hypervisor provides isolation for VMs down to the silicon layer, whereas, by comparison, there is very little isolation between every container on a node.

There are container technologies that will run a container on their own VM. The container is still just a process.

When containers aren’t running, they’re simply a “tarball of tarballs,” where each layer of the filesystem is stored in a file. The image is still stored on the host system, multiple host systems, or wherever the container has been run or pulled previously.

If you are new to the term “tarball,” it is a file created by the `tar` Unix command, which archives and compresses multiple files into a single file. The term “tarball” is a combination of the command used to create the file, `tar`, which is short for **Tape Archive**, and the ball, which is a bundle of files.

Conversely, a VM has its own virtual disk that stores the entire OS. While there are some very lightweight VM technologies, there’s often an order of magnitude difference between the size of a VM and that of a container.

While some people refer to containers as lightweight VMs, that couldn’t be further from the truth. They aren’t isolated in the same way and require more attention to be paid to the details of how they are run on a node.

From this section, you may think that we are implying that containers are not secure. Nothing could be further from the truth. Securing a Kubernetes cluster, and the containers running on it, requires attention to detail and an understanding of how containers differ from VMs. Since so many people do understand VMs, it’s easy to try to compare them to containers, but doing so puts you at a disadvantage, since they are very different technologies.

Once you understand the limitations of a default configuration and the potential dangers that come from it, you can remediate the “issues.”

## Container breakouts

A container breakout is when the process of your container gets access to the underlying node. Once on the node, an attacker has access to all the other pods and any capability that the node has in your environment. A breakout can also be a matter of mounting the local filesystem in your container. An example from <https://securekubernetes.com>, originally pointed out by Duffie Cooley, Field CTO at Isovalent, uses a container to mount the local filesystem. Running this on a KinD cluster opens both reads and writes to the node’s filesystem:

```
kubectl run r00t --restart=Never -ti --rm --image lol --overrides
'{"spec": {"hostPID": true, "containers": [{"name": "1", "image": "alpine", "command": ["nsenter", "--mount=/proc/1/ns/mnt", "--", "/bin/bash"], "stdin": true, "tty": true, "imagePullPolicy": "IfNotPresent", "securityContext": {"privileged": true}}]}}
If you don't see a command prompt, try pressing Enter.
```

The `run` command in the preceding code started a container that added an option that is key to this example, `hostPID: true`, which allows the container to share the host’s process namespace. You can see a few other options, such as `--mount` and a security context setting that sets `privileged` to `true`. All of the options combined will allow us to write to the host’s filesystem.

Now that you are in the container, execute the `ls` command to look at the filesystem. Notice how the prompt is `root@r00t:/#`, confirming that you are in the container and not on the host:

```
root@r00t:/# ls
bin  boot  build  dev  etc  home  kind  lib  lib32  lib64  libx32  media  mnt
opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

To prove that we have mapped the host's filesystem to our container, create a file called `this_is_from_a_container` and exit the container:

```
root@r00t:/# touch this_is_from_a_container
root@r00t:/# exit
```

Finally, let's look at the host's filesystem to see whether the container created the file. Since we are running KinD with a single worker node, we need to use Docker to exec into the worker node. If you are using the KinD cluster from the book, the worker node is called `cluster01-worker`:

```
docker exec -ti cluster01-worker ls /
bin  boot  build  dev  etc  home  kind  lib  lib32  lib64  libx32  media  mnt
opt  proc  root  run  sbin  srv  sys  this_is_from_a_container  tmp  usr  var
```

There it is! In this example, a container was run that mounted the local filesystem. From inside the pod, the `this_is_from_a_container` file was created. After exiting the pod and entering the node container, the file was there. Once an attacker has access to a node's filesystem, they also have access to the kubelet's credentials, which can open the entire cluster up.

It's not hard to envision a string of events that can lead to a Bitcoin miner (or worse) running on a cluster. A phishing attack gets the credentials that a developer uses for their cluster. Even though those credentials only have access to one namespace, a container is created to get the kubelet's credentials, and from there, containers are launched to stealthily deploy miners across the environment. There are certainly multiple mitigations that could be used to prevent this attack, including the following:

- Multi-factor authentication, which would have kept the phished credentials from being used
- Pre-authorizing only certain containers
- A Gatekeeper policy, which would have prevented this attack by stopping a container from running as privileged
- A properly secured image

It's important to note that none of these mitigations are provided by default in Kubernetes, but that's one of the main reasons you're reading this book!

We've already talked about authentication in previous chapters and the importance of multi-factor authentication. We even used port forwarding to set up a miner through our dashboard! This is another example of why authentication is such an important topic in Kubernetes.

The next two approaches listed can be done using Gatekeeper. We covered pre-authorizing containers and registries in *Chapter 11, Extending Security Using Open Policy Agent*. This chapter will focus on using Gatekeeper to enforce node-centric policies, such as whether a pod should run as privileged.

Finally, at the core of security is a properly designed image. In the case of physical machines and VMs, this is accomplished by securing the base OS. When you install an OS, you don't select every possible option during installation. It is considered poor practice to have anything running on a server that is not required for its role or function. This same practice needs to be carried over to the images that will run on your clusters, which should only contain the necessary binaries that are required for your application.

Given how important it is to properly secure images on your cluster, the next section explores container design from a security standpoint. While not directly related to Gatekeeper’s policy enforcement, it’s an important starting point for node security. It’s also important to understand how to build containers securely in order to better debug and manage your node security policies. Building a locked-down container makes managing the security of nodes much easier.

## Properly designing containers

Before exploring how to protect your nodes using Gatekeeper, it’s important to address how containers are designed. Often, the hardest part of using a policy to mitigate attacks on a node is the fact that so many containers are built and run as root. Once a restricted policy is applied, the container won’t start on reload even if it was running fine after the policy was applied. This is problematic on multiple levels. System administrators have learned over the decades of networked computing not to run processes as root, especially services such as web servers that are accessed anonymously over untrusted networks.

All networks should be considered “untrusted.” Assuming that all networks are hostile leads to a more secure approach to implementation. It also means that services that need security need to be authenticated. This concept is called zero trust. It has been used and advocated by identity experts for years, but it was popularized in the DevOps and cloud-native worlds by Google’s BeyondCorp whitepaper (<https://cloud.google.com/beyondcorp>). The concept of zero trust should apply inside your clusters too!

Bugs in code can lead to access to underlying compute resources, which can then lead to breakouts from a container. Running as root in a privileged container when not needed can lead to a breakout if exploited via a code bug.

The Equifax breach in 2017 used a bug in the Apache Struts web application framework to run code on the server, which was then used to infiltrate and extract data. Had this vulnerable web application been running on Kubernetes with a privileged container, the bug could have led to the attackers gaining access to the cluster.

When building containers, at a minimum, the following should be observed:

- **Run as a user other than root:** The vast majority of applications, especially microservices, don’t need root. Don’t run as root.
- **Only write to volumes:** If you don’t write to a container, you don’t need write access. Volumes can be controlled by Kubernetes. If you need to write temporary data, use an `emptyVolume` object instead of writing to the container’s filesystem. This makes it easier to detect something malicious that is trying to make changes to a container at runtime, such as replacing a binary or a file.
- **Minimize binaries in your container:** This can be tricky. There are those that advocate for “distroless” containers that only contain the binary for the application, statically compiled – no shells, no tools. This can be problematic when trying to debug why an application isn’t running as expected. It’s a delicate balance. In Kubernetes 1.25, ephemeral containers were introduced to make this easier. We’ll cover this later in the section.

- **Scan containers for known Common Vulnerabilities and Exposures (CVEs), and rebuild often:** One of the benefits of a container is that it can be easily scanned for known CVEs. There are several tools and registries that will do this for you, such as **Grype** from Anchor (<https://github.com/anchore/grype>) or **Trivy** (<https://github.com/aquasecurity/trivy>) from Aqua Security. Once CVEs have been patched, rebuild. A container that hasn't been rebuilt in months, or years even, is every bit as dangerous as a server that hasn't been patched.

Let's delve more into debugging "distroless" images and scanning containers.

## Using and Debugging Distroless Images

The idea of a "distroless" image is not new. Google was one of the first to really popularize their use (<https://github.com/GoogleContainerTools/distroless>), and recently, Chainguard has begun releasing and maintaining distroless images built on their **Wolfi** (<https://github.com/wolfi-dev>) distribution of Linux. The idea is that your base image isn't an Ubuntu, Red Hat, or other common Linux distribution but is, instead, a minimum set of binaries to run the system. For instance, the Java 17 image only includes OpenJDK. No tools. No utilities. Just the JDK. From a security standpoint, this is great because there are fewer "things" that can be used to compromise your environment. When an attacker doesn't need a shell to run a command, why make their lives easier?

There are two main drawbacks to this approach:

- **Debugging Running Containers:** Without dig or nslookup, how do you know the issue is DNS? We might know it's always DNS, but you still need to prove it. You may also need to debug network services, connections, etc. Without the common tools needed to debug those services, how can you determine the issue?
- **Support and Compatibility:** One of the benefits of using a common distro as your base image is that it's likely that your enterprise already has a support contract with the distro vendor. Google's Distroless is based on Debian Linux, which has no official vendor support. Wolfi is built on Alpine, which doesn't have its own support either (although Chainguard does offer commercial support for its images). If your container breaks and you suspect the issue is in your base image, you're not going to get much help from another distro's vendor.

The issues with support and compatibility aren't really technical issues; they are risk management issues that need to be addressed by your team. If you're using a commercial product, that's generally something that is covered by your support contract. If you're talking about home-grown containers, it's important to understand the risks and potential mitigations.

Debugging a distroless container is now much easier than it once was. In version 1.25, Kubernetes introduced the concept of ephemeral containers that allow you to attach a container to a running pod. This ephemeral container can include all those debugging utilities that you don't have in your distroless image. The kubectl debug command was added to make it easier to use.

First, in a new cluster, launch the Kubernetes Dashboard, and then try to attach a shell to it:

```
$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
```

```
.  
. .  
  
$ export K8S_DB_POD=$(kubectl get pods -l k8s-app=kubernetes-dashboard -n kubernetes-dashboard -o json | jq -r '.items[0].metadata.name')  
$ kubectl exec -ti $K8S_DB_POD -n kubernetes-dashboard -- sh  
error: Internal error occurred: error executing command in  
container: failed to exec in container: failed to start exec  
"24b9dba21332299828b4d8f46c360c8afe0cadfd693e6651694a63917d28b910": OCI runtime  
exec failed: exec failed: unable to start container process: exec: "sh":  
executable file not found in $PATH: unknown
```

The last line shows us that there's no shell in the kubernetes-dashboard pod. When we try to exec into the pod, it fails because the executable isn't found. Now, we can attach a debug pod that lets us use our debugging tools:

```
$ kubectl debug -it --attach=true -c debugger --image=busybox $K8S_DB_POD -n kubernetes-dashboard  
If you don't see a command prompt, try pressing enter.  
/ # ps -A  
 PID  USER      TIME  COMMAND  
   1 root      0:00  sh  
  14 root      0:00  ps -A  
 / # ping  
 BusyBox v1.36.1 (2023-07-17 18:29:09 UTC) multi-call binary.  
 Usage: ping [OPTIONS] HOST  
. . .  
 / # nslookup  
 BusyBox v1.36.1 (2023-07-17 18:29:09 UTC) multi-call binary.  
 Usage: nslookup [-type=QUERY_TYPE] [-debug] HOST [DNS_SERVER]  
 Query DNS about HOST  
 QUERY_TYPE: soa,ns,a,aaaa,cname,mx,txt,ptr,srv,any
```

We were able to attach our busybox image to the dashboard pod and use our tools!

This certainly helps with debugging, but what if we need to get into the dashboard process? Notice that when I ran the `ps` command, there was no dashboard process. That's because the containers in a pod all run their own process space with limited shared points (like `volumeMounts`). So while this might help with testing network resources, it isn't as close to our workload as possible. We can add the `shareProcessNamespace: true` option to our Deployment, but now our containers all share the same process space and lose a level of isolation. You could patch a running Deployment when needed, but now you're relaunching your pods, which may clear the issue on its own.

Distroless images are minimal images that can lower your security exposure by making it harder for attackers to leverage your pods as an attack vector. Minimizing the images does have trade-offs, and it's important to keep those trade-offs in mind. While you will get a smaller image that's harder to compromise, it can make debugging operations harder and may have impacts on your vendor support agreements.

Next, we'll look at how scanning images for known exploits should be incorporated into your build process.

## Scanning Images for Known Exploits

How do you know if your image has a vulnerability? There's a common place to report vulnerabilities in software, called the **Common Vulnerabilities and Exposures** (CVE) database from MITRE. This is, in theory, a common place where researchers and users can report vulnerabilities to vendors. In theory, this is a place where a user could go to learn if they have a known vulnerability in the software they're running.

This is a vast oversimplification of the issue of looking for vulnerabilities. In the past few years, there's been an extremely critical view of the quality of CVE data due to the way that the CVE database is managed and maintained. Unfortunately, this is really the only common database there is. With that said, it's common practice to scan containers and compare the software versions in your containers against what is contained in the CVE database. If you find a vulnerability, this creates an action for the team to remediate.

On the one hand, quickly patching known exploits is one of the best ways to cut down your security risk. On the other hand, quickly patching exploits that may not need patching can lead to broken systems. It's a difficult balancing act that requires not only understanding how scanners work but also having automation and testing that gives you confidence in updating your infrastructure.

Scanning for CVEs is a standard way to report security issues. Application and OS vendors will update CVEs with patches to their code that fix the issues. This information is then used by security scanning tools to take action when a container has a known issue that has been patched.

There are several open source scanning options. I like to use Grype from Anchore, but Trivy from AquaSecurity is another great option. What's important here is that both of these scanners will pull in your container, compare the installed packages and libraries against the CVE database, and tell you what CVEs there are and if they've been patched. For instance, if you were to scan two OpenUnison images, we'd see slightly different results. When scanning `ghcr.io/openunison/openunison-k8s:1.0.37-a207c4`, there were 43 known vulnerabilities. This image was about six days old, so there were about 15 medium CVEs that had patches. Tonight, when our process runs (which I'll explain in a moment), a new container will be generated to patch these CVEs. If we run Grype against a container that was built two weeks ago, there will be 45 known CVEs, with 2 that were patched in the latest build.

The point of this exercise is that scanners are very useful for container hygiene. At Tremolo Security, we scan our published images every night, and if there's a new OS-level CVE that's been patched, we rebuild. This keeps our images up to date.

Container scanning isn't the only scanning. We also use [snyk.io](https://snyk.io) to scan our builds and dependencies for known vulnerabilities, bumping them to fixed versions that are available. We're confident that we can do this because our automated testing includes hundreds of automated tests that will catch issues with these upgrades. Our goal is to have zero patchable CVEs at release time. Unless there's an absolutely critical vulnerability, like the Log4J fiasco from 2021, we generally release four to five releases a year.

As an open source project maintainer, I have to mention what is often seen as a sore spot in the community. The container scanners will tell you that a library is present, but they will not tell you if the library is vulnerable. There's quite a bit of nuance to what constitutes a vulnerability. Once you've found a "hit" in your scanner to a project, please do not immediately open an issue on GitHub. This causes quite a bit of work for project maintainers that is of little value.

Finally, you can be too reliant on scanners. Some very talented security gooses (Brad Geesaman, Ian Coldwater, Rory McCune, and Duffie Cooley) talked about faking out scanners at KubeCon EU 2023 in *Malicious Compliance: Reflections on Trusting Container Scanners*: <https://kccnceu2023.sched.com/event/1Hybu/malicious-compliance-reflections-on-trusting-container-scanners-ian-coldwater-independent-duffie-cooley-isovalent-brad-geesaman-ghost-security-rory-mccune-datadog>. I highly recommend taking the time to watch this video and the issues it raises on scanner reliance.

Once you've scanned your containers and restricted how the containers run, how do you know they'll work? It's important to test in a restrictive environment. At the time of writing, the most restrictive defaults for any Kubernetes distribution on the market belong to Red Hat's OpenShift. In addition to sane default policies, OpenShift runs pods with a random user ID, unless the pod definition specifies a specific ID.

It's a good idea to test your containers on OpenShift, even if it's not your distribution for production use. If a container runs on OpenShift, it's likely to work with almost any security policy that a cluster can throw at it. The easiest way to do this is with Red Hat's CodeReady Containers (<https://developers.redhat.com/products/codeready-containers>). This tool can run on your local laptop and launches a minimal OpenShift environment that can be used to test containers.

While OpenShift has very tight security controls out of the box, it doesn't use **Pod Security Policies (PSPs)**, Pod Security Standards, or Gatekeeper. It has its own policy system that pre-dates PSPs, called **Security Context Constraints (SCCs)**. SCCs are similar to PSPs but don't use RBAC to associate with pods.

Now that we've explored how to create secure container images, the next step is to make sure our clusters are built, ensuring that images that don't follow these standards are prevented from running.

## Enforcing node security with Gatekeeper

So far, we've seen what can happen when containers are allowed to run on a node without any security policies in place. We've also examined what goes into building a secure container, which will make enforcing node security much easier. The next step is to examine how to design and build policies using Gatekeeper to lock down your containers.

## What about Pod Security Policies?

Doesn't Kubernetes have a built-in mechanism to enforce node security? Yes! In 2018, the Kubernetes project decided that the **Pod Security Policies (PSP)** API would never leave beta. The configuration was too confusing, being a hybrid of Linux-focused configuration options and RBAC assignments. It was determined that the fix would likely mean an incompatible final release from the current release. Instead of marking a complex and difficult-to-manage API as generally available, the project made a difficult decision to deprecate and remove the API.

At the time, it was stated that the PSP API would not be removed until a replacement was ready for release. This changed in 2020 when the Kubernetes project adopted a new policy that no API can stay in beta for more than three releases. This forced the project to re-evaluate how to move forward with replacing PSPs. In April 2021, Tabitha Sable wrote a blog post on the future of PSPs (<https://kubernetes.io/blog/2021/04/06/podsecuritypolicy-deprecation-past-present-and-future/>). To cut a long story short, they are officially deprecated as of 1.21 and were removed in 1.25. Their replacement, called **Pod Security Standards**, became GA in 1.26. We'll cover these after we walk through using Gatekeeper to protect your nodes from your pods.

## What are the differences between PSPs, PSA, and Gatekeeper?

Before diving into the implementation of node security with Gatekeeper, let's look at how the legacy PSPs, the new **Pod Security Admission (PSA)**, and Gatekeeper are different. If you're familiar with PSPs, this will be a helpful guide for migrating. If you have never worked with PSPs, this can give you a good idea as to where to look when things don't work as expected.

The one area that all three technologies have in common is that they're implemented as admission controllers. As we learned in *Chapter 11, Extending Security Using Open Policy Agent*, an admission controller is used to provide additional checks beyond what the API server provides natively. In the case of Gatekeeper, PSPs, and PSA, the admission controller makes sure that the pod definition has the correct configuration to run with the least privileges needed. This usually means running as a non-root user, limiting access to the host, and so on. If the required security level isn't met, the admission controller fails, stopping a pod from running.

While all three technologies run as admission controllers, they implement their functionality in very different ways. PSPs are applied by first defining a `PodSecurityPolicy` object, and then defining RBAC `Role` and `RoleBinding` objects to allow a `ServiceAccount` to run with a policy. The PSP admission controller will make a decision based on whether the "user" who created the pod or the `ServiceAccount` that the pod runs on is authorized, based on the RBAC bindings. This leads to difficulties in designing and debugging the policy application. It's difficult to authorize if a user can submit a pod because users usually don't create pod objects anymore. They create `Deployments`, `StatefulSets`, or `Jobs`. Then, there are controllers that run with their own `ServiceAccounts`, which then create pods. The PSP admission controller never knows who submitted the original object. In the last chapter, we covered how Gatekeeper binds policies via namespace and label matching; this doesn't change with node security policies. Later on, we'll do a deep dive into how to assign policies.

PSA is implemented at the namespace level, not at the individual pod level. It's assumed that since the namespace is the security boundary for a cluster, then any pods that run in a namespace should share the same security context. This can often work, but there are limitations. For instance, if you need an init container that needs to change file permissions on a mount, you could run into issues with PSA.

In addition to assigning policies differently, Gatekeeper, PSPs, and PSA handle overlapping policies differently. PSPs will try to take the *best* policy based on the account and capabilities being requested. This allows you to define a high-level blanket policy that denies all privileges and then create specific policies for individual use cases, such as letting the NGINX Ingress Controller run on port 443. Gatekeeper, conversely, requires all policies to pass. There's no such thing as a *best* policy; all policies must pass. This means that you can't apply a blanket policy and then carve out exceptions. You have to explicitly define your policies for each use case. PSA is universal across the namespace, so there are no exceptions at the API level and nothing to vary. You can set up specific exemptions for users, runtime classes, or namespaces, but these are global and static.

Another difference between the three approaches is how policies are defined. The PSP specification is a Kubernetes object that is mostly based on Linux's built-in security model. The object itself has been assembled with new properties as needed, in an inconsistent way. This led to a confusing object that didn't complement the addition of Windows containers. Conversely, Gatekeeper has a series of policies that have been pre-built and are available from their GitHub repo: <https://github.com/open-policy-agent/gatekeeper-library/tree/master/library/pod-security-policy>. Instead of having one policy, each policy needs to be applied separately. The PSA defines profiles that are based on common security patterns. There really isn't much to define.

Finally, the PSP admission controller had some built-in mutations. For instance, if your policy didn't allow root and your pod didn't define what user to run as, the PSP admission controller would set a user ID of 1. Gatekeeper has a mutating capability (which we covered in *Chapter 11, Extending Security Using Open Policy Agent*), but that capability needs to be explicitly configured to set defaults. PSA has no mutation capabilities.

Having examined the differences between PSPs, PSA, and Gatekeeper, let's next dive into how to authorize node security policies in your cluster.

## Authorizing node security policies

In the previous section, we discussed the differences between authorizing policies between Gatekeeper, PSPs, and PSA. Now, we'll look at how to define your authorization model for policies. Before we get ahead of ourselves, we should discuss what we mean by "authorizing policies."

When you create a pod, usually through a Deployment or StatefulSet, you choose what node-level capabilities you want, with settings on your pod inside of the securityContext sections. You may request specific capabilities or a host mount. Gatekeeper examines your pod definition and decides, or authorizes, that your pod definition meets the policy's requirements by matching an applicable ConstraintTemplate via its constraint's match section. Gatekeeper's match section lets you match on the namespace, kind of object, and labels on the object. At a minimum, you'll want to include namespaces and object types. Labels can be more complicated.

A large part of deciding whether labels are an appropriate way to authorize a policy is based on who can set the labels and why. In a single-tenant cluster, labels are a great way to create constrained deployments. You can define specific constraints that can be applied directly via a label. For instance, you may have an operator in a namespace that you don't want to have access to a host mount but a pod that does. Creating a policy with specific labels will let you apply more stringent policies to the operator than the pod.

The risk with this approach lies in multi-tenant clusters where you, as the cluster owner, cannot limit what labels can be applied to a pod. Kubernetes' RBAC implementation doesn't provide any mechanism for authorizing specific labels. You could implement something using Gatekeeper, but that would be 100% custom. Since you can't stop a namespace owner from labeling a pod, a compromised namespace administrator's account can be used to launch a privileged pod without there being any checks in place from Gatekeeper.

You could, of course, use Gatekeeper to limit labels. The trick is that, similar to issues with PSPs, Gatekeeper won't know who created the label at the pod level because the pod is generally created by a controller. You could enforce at the Deployment or StatefulSet level, but that will mean that other controller types won't be supported. This is why PSA uses the namespace as the label point. Namespaces are the security boundary for clusters. You could also carve out specific exceptions for init containers if needed.

In *Chapter 11, Extending Security Using Open Policy Agent*, we learned how to build policies in Rego and deploy them using Gatekeeper. In this chapter, we've discussed the importance of securely building images, the differences between PSPs and Gatekeeper for node security, and finally, how to authorize policies in your clusters. Next, we'll lock down our testing cluster.

## Deploying and debugging node security policies

Having gone through much of the theory in building node security policies in Gatekeeper, let's dive into locking down our test cluster. The first step is to start with a clean cluster and deploy Gatekeeper:

```
$ kubectl apply -f https://raw.githubusercontent.com/open-policy-agent/gatekeeper/master/deploy/gatekeeper.yaml
```

Next, we'll want to deploy our node's ConstraintTemplate objects. The Gatekeeper project builds and maintains a library of templates that replicate the existing PodSecurityPolicy object at <https://github.com/open-policy-agent/gatekeeper-library/tree/master/library/pod-security-policy>. For our cluster, we're going to deploy all of the policies, except the read-only filesystem seccomp, selinux, apparmor, flexvolume, and host volume policies. I chose to not deploy the read-only filesystem because it's still really common to write to a container's filesystem, even though the data is ephemeral, and enforcing this would likely cause more harm than good. The seccomp, apparmor, and selinux policies weren't included because we're running on a KinD cluster. Finally, we ignored the volumes because it's not a feature we plan on worrying about. However, it's a good idea to look at all these policies to see whether they should be applied to your cluster. The chapter12 folder has a script that will deploy all our templates for us. Run `chapter12/deploy_gatekeeper_psp_policies.sh`. Once that's done, we have our ConstraintTemplate objects deployed, but they're not being enforced because we haven't set up any policy implementation objects. Before we do that, we should set up some sane defaults.

## Generating security context defaults

In *Chapter 11, Extending Security Using Open Policy Agent*, we discussed the trade-offs between having a mutating webhook generating sane defaults for your cluster versus explicit configuration.

I'm a fan of sane defaults, since they lead to a better developer experience and make it easier to keep things secure. The Gatekeeper project has a set of example mutations for this purpose at <https://github.com/open-policy-agent/gatekeeper-library/tree/master/mutation/pod-security-policy>. For this chapter, I took them and tweaked them a bit. Let's deploy them and then recreate all our pods so that they have our "sane defaults" in place, before rolling out our constraint implementations:

```
$ kubectl create -f chapter12/default_mutations.yaml
assign.mutations.gatekeeper.sh/k8spspdefaultallowprivilegeescalation created
assign.mutations.gatekeeper.sh/k8spspfsgroup created
assign.mutations.gatekeeper.sh/k8spsprunasnonroot created
assign.mutations.gatekeeper.sh/k8spsprunasgroup created
assign.mutations.gatekeeper.sh/k8spsprunasuser created
assign.mutations.gatekeeper.sh/k8spssupplementalgroups created
assign.mutations.gatekeeper.sh/k8spspcapabilities created
$ sh chapter12/delete_all_pods_except_gatekeeper.sh
calico-system
pod "calico-kube-controllers-7f58dbcbbd-ckshb" deleted
pod "calico-node-g5cwp" deleted
```

Now, we can deploy an NGINX pod and see how it now has a default security context:

```
$ kubectl create ns test-mutations
$ kubectl create deployment test-nginx --image=ghcr.io/openunison/openunison-
k8s-html:latest -n test-mutations
$ kubectl get pods -l app=test-nginx -o jsonpath='{.items[0].spec.
securityContext}' -n test-mutations
{"fsGroup":3000,"supplementalGroups":[3000]}
```

Our NGINX pod now has a `securityContext` that determines what user the container should run as if it's privileged, and if it needs any special capabilities. If, for some reason in the future, we want containers to run as a different process, instead of changing every manifest, we can now change our mutation configuration. Now that our defaults are in place and applied, the next step is to implement instances of our `ConstraintTemplates` to enforce our policies.

## Enforcing cluster policies

With our mutations deployed, we can now deploy our constraint implementations. Just as with the `ConstraintTemplate` objects, the Gatekeeper project provides example template implementations for each template. I put together a condensed version for this chapter in `chapter12/minimal_gatekeeper_constraints.yaml` that is designed to have a minimum set of privileges across a cluster, ignoring `kube-system` and `calico-system`. Deploy this YAML file and wait a few minutes:

```
$ kubectl apply -f chapter12/minimal_gatekeeper_constraints.yaml
k8spspallowprivilegeescalationcontainer.constraints.gatekeeper.sh/privilege-
escalation-deny-all created
k8spspcapabilities.constraints.gatekeeper.sh/capabilities-drop-all created
k8spspforbiddensysctls.constraints.gatekeeper.sh/psp-forbid-all-sysctls created
k8spsphostfilesystem.constraints.gatekeeper.sh/psp-deny-host-filesystem created
k8spsphostnamespace.constraints.gatekeeper.sh/psp-block-all-host-namespace
created
k8spsphostnetworkingports.constraints.gatekeeper.sh/psp-deny-all-host-network-
ports created
k8spspprivilegedcontainer.constraints.gatekeeper.sh/psp-deny-all-privileged-
container created
k8spspprocmount.constraints.gatekeeper.sh/psp-proc-mount-default created
k8spspallowedusers.constraints.gatekeeper.sh/psp-pods-allowed-user-ranges
created
```

Remember from *Chapter 11, Extending Security Using Open Policy Agent*, that a key feature of Gatekeeper over generic OPA is its ability to not just act as a validating webhook but also audit existing objects against policies. We're waiting so that Gatekeeper has a chance to run its audit against our cluster. Audit violations are listed in the status of each implementation of each ConstraintTemplate.

To make it easier to see how compliant our cluster is, I wrote a small script that will list the number of violations per ConstraintTemplate:

```
$ sh chapter12/show_constraint_violations.sh
k8spspallowedusers.constraints.gatekeeper.sh 16
k8spspallowprivilegeescalationcontainer.constraints.gatekeeper.sh 2
k8spspcapabilities.constraints.gatekeeper.sh 2
k8spspforbiddensysctls.constraints.gatekeeper.sh 0
k8spsphostfilesystem.constraints.gatekeeper.sh 1
k8spsphostnamespace.constraints.gatekeeper.sh 0
k8spsphostnetworkingports.constraints.gatekeeper.sh 1
k8spspprivilegedcontainer.constraints.gatekeeper.sh 0
k8spspprocmount.constraints.gatekeeper.sh 0
k8spspreadonlyfilesystem.constraints.gatekeeper.sh null
```

We have several violations. If you don't have the exact number, that's OK. The next step is debugging and correcting them.

## Debugging constraint violations

With our constraint implementations in place, we have several violations that need to be remediated. Let's take a look at the privilege escalation policy violation:

```
$ kubectl get k8spspallowprivilegeescalationconstraints.gatekeeper.sh -o jsonpath='{$.items[0].status.violations}' | jq -r
[
  {
    "enforcementAction": "deny",
    "kind": "Pod",
    "message": "Privilege escalation container is not allowed: controller",
    "name": "ingress-nginx-controller-744f97c4f-msmkz",
    "namespace": "ingress-nginx"
  }
]
```

Gatekeeper tells us that the `ingress-nginx-controller-744f97c4f-msmkz` pod in the `ingress-nginx` namespace is attempting to elevate its privileges. Looking at its `SecurityContext` reveals the following:

```
$ kubectl get pod ingress-nginx-controller-744f97c4f-msmkz -n ingress-nginx -o jsonpath='{$.spec.containers[0].securityContext}' | jq -r
{
  "allowPrivilegeEscalation": true,
  "capabilities": {
    "add": [
      "NET_BIND_SERVICE"
    ],
    "drop": [
      "all"
    ]
  },
  "runAsGroup": 2000,
  "runAsNonRoot": true,
  "runAsUser": 101
}
```

Nginx is requesting to be able to escalate its privileges and add the `NET_BIND_SERVICE` privilege so that it can run on port 443 without being a root user. Going back to our list of constraint violations, in addition to having a privilege escalation violation, there was also a capabilities violation. Let's inspect that violation:

```
$ kubectl get k8spspcapabilities.constraints.gatekeeper.sh -o jsonpath='{$.items[0].status.violations}' | jq -r
[
  {
    "enforcementAction": "deny",
    "kind": "Pod",
    "message": "container <controller> has a disallowed capability. Allowed
```

```

capabilities are []",
  "name": "ingress-nginx-controller-744f97c4f-msmkz",
  "namespace": "ingress-nginx"
}
]

```

It's the same container violating both constraints. Having determined which pods are out of compliance, we'll fix their configurations next.

Earlier in this chapter, we discussed the difference between PSPs and Gatekeeper, with one of the key differences being that while PSPs attempt to apply the “best” policy, Gatekeeper will evaluate against all applicable constraints. This means that while in PSP you can create a “blanket” policy (often referred to as a “Default Restrictive” policy) and then create more relaxed policies for specific pods, Gatekeeper will not let you do that. In order to keep these violations from stopping Nginx from running the constraint implementations, they must be updated to ignore our Nginx pods. The easiest way to do this is to add `ingress-nginx` to our list of `excludednamespaces`.

I did this for all of our constraint implementations in `chapter12/make_cluster_work_policies.yaml`. Deploy using the `apply` command:

```

kubectl apply -f chapter12/make_cluster_work_policies.yaml
k8spsphostnetworkingports.constraints.gatekeeper.sh/psp-deny-all-host-network-
ports configured
k8spsphostfilesystem.constraints.gatekeeper.sh/psp-deny-host-filesystem
configured
k8spspcapabilities.constraints.gatekeeper.sh/capabilities-drop-all configured
k8spspallowprivilegeescalationcontainer.constraints.gatekeeper.sh/privilege-
escalation-deny-all configured

```

After a few minutes, let's run our violation check:

```

sh ./chapter12/show_constraint_violations.sh
k8spspallowedusers.constraints.gatekeeper.sh 12
k8spspallowprivilegeescalationcontainer.constraints.gatekeeper.sh 0
k8spspcapabilities.constraints.gatekeeper.sh 0
k8spspforbiddensysctls.constraints.gatekeeper.sh 0
k8spsphostfilesystem.constraints.gatekeeper.sh 0
k8spsphostnamespace.constraints.gatekeeper.sh 0
k8spsphostnetworkingports.constraints.gatekeeper.sh 0
k8spspprilegedcontainer.constraints.gatekeeper.sh 0
k8spspprocmount.constraints.gatekeeper.sh 0
k8spspreadonlyrootfilesystem.constraints.gatekeeper.sh null

```

The only violations left are for our allowed users' constraints. These violations all come from `gatekeeper-system` because the Gatekeeper pods don't have users specified in their `SecurityContext`. These pods haven't received any of our sane defaults because, in the Gatekeeper Deployment, the `gatekeeper-system` namespace is ignored. Despite being ignored, it's still listed as a violation, even though it won't be enforced.

Now that we have eliminated the violations, we're done, right? Not quite. Even though Nginx isn't generating any errors, we aren't making sure it's running with the least privilege. If someone were to launch a pod in the `ingress-nginx` namespace, it could request privileges and additional capabilities without being blocked by Gatekeeper. We'll want to make sure that any pod launched in the `ingress-nginx` namespace can't escalate beyond what it needs. In addition to eliminating the `ingress-nginx` namespace from our cluster-wide policy, we need to create a new constraint implementation that limits which capabilities can be requested by pods in the `ingress-nginx` namespace.

We know that Nginx requires the ability to escalate privileges and to request `NET_BIND_SERVICE` so that we can create a constraint implementation:

```
---
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sPSPCapabilities
metadata:
  name: capabilities-ingress-nginx
spec:
  match:
    kinds:
      - apiGroups: []
        kinds: ["Pod"]
    namespaces: ["ingress-nginx"]
  parameters:
    requiredDropCapabilities: ["all"]
    allowedCapabilities: ["NET_BIND_SERVICE"]
```

We created a constraint implementation that mirrors the Deployment's required `securityContext` section. We didn't create a separate constraint implementation for privilege escalation because that `ConstraintTemplate` has no parameters. It's either enforced or it isn't. There's no additional work to be done for that constraint in the `ingress-nginx` namespace once the namespace has been removed from the blanket policy.

I repeated this debugging process for the other violations and added them to `chapter12/enforce_node_policies.yaml`. You can deploy them to finish the process.

You may be wondering why we are enforcing at the namespace level and not with specific labels to isolate individual pods. We discussed authorization strategies earlier in this chapter, and continuing the themes here, I don't see additional label-based enforcement as adding much value. Anyone who can create a pod in this namespace can set the labels. Limiting the scope more doesn't add much in the way of security.

The process for deploying and debugging policies is very detail-oriented. In a single-tenant cluster, this may be a one-time action or a rare action, but in a multi-tenant cluster, the process does not scale. Next, we'll look at strategies to apply node security in multi-tenant clusters.

## Scaling policy deployment in multi-tenant clusters

In the previous examples, we took a “small batch” approach to our node security. We created a single cluster-wide policy and then added exceptions as needed. This approach doesn’t scale in a multi-tenant environment for a few reasons:

- The `excludedNamespaces` attribute in a constraint’s `match` section is a list and is difficult to patch in an automated way. Lists need to be patched, including the original, so it’s more than a simple “apply this JSON” operation.
- You don’t want to make changes to global objects in multi-tenant systems. It’s easier to add new objects and link them to a source of truth. It’s easier to trace why a new constraint implementation was created using labels than to figure out why a global object was changed.
- You want to minimize the likelihood of a change on a global object being able to affect other tenants. Adding new objects specifically for each tenant minimizes that risk.

In an ideal world, we’d create a single global policy and then create objects that can be more specific for individual namespaces that need elevated privileges.

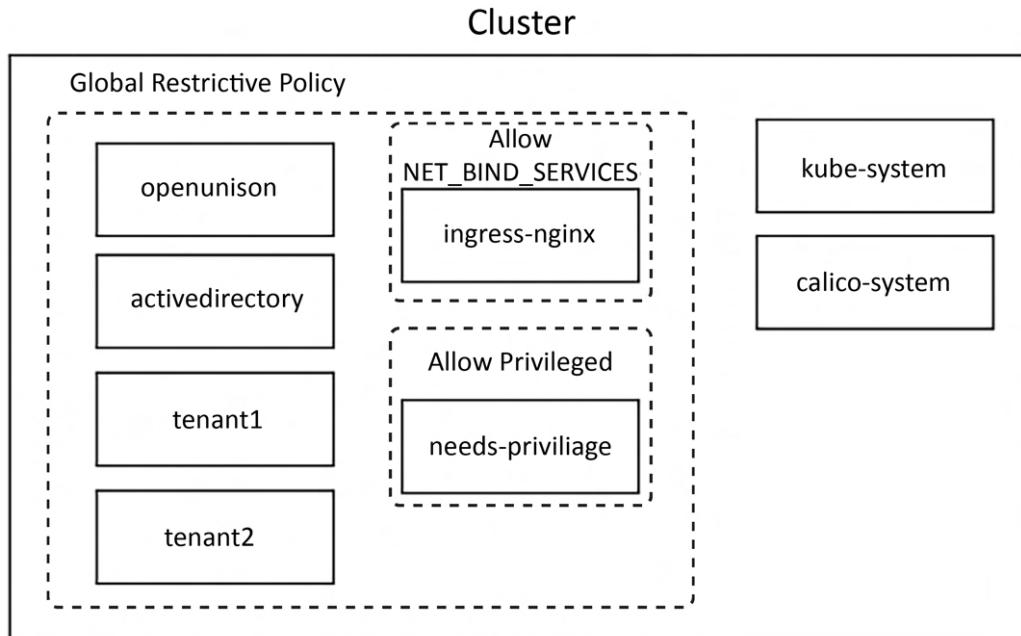


Figure 12.1: Ideal policy design for a multi-tenant cluster

The above diagram illustrates what I mean by having a blanket policy. The large, dashed box with rounded corners is a globally restrictive policy that minimizes what a pod is capable of. Then, the smaller, dashed rounded-corner boxes are carveouts for specific exceptions. As an example, the `ingress-nginx` namespace would be created with restrictive rights, and a new policy would be added that is scoped specifically to the `ingress-nginx` namespace, which would grant Nginx the ability to run with the `NET_BIND_SERVICES` capabilities. By adding an exception for a specific need to a cluster-wide restrictive policy, you're decreasing the likelihood that a new namespace will expose the entire cluster to a vulnerability if a new policy isn't added. The system is built to fail "closed."

The above scenario is not how Gatekeeper works. Every policy that matches must succeed; there's no way to have a global policy. In order to effectively manage a multi-tenant environment, we need to:

1. Have policies for system-level namespaces that cluster administrators can own
2. Create policies for each namespace that can be adjusted as needed
3. Ensure that namespaces have policies before pods can be created

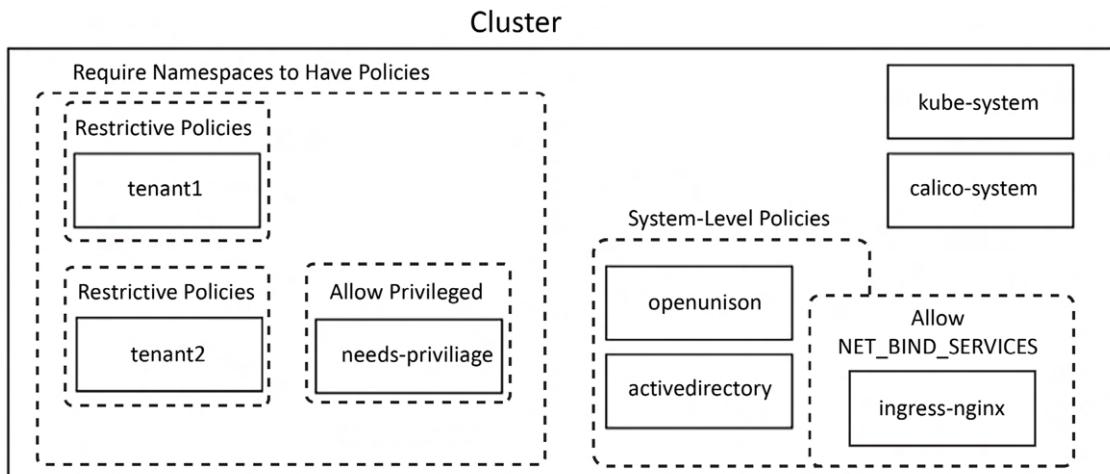


Figure 12.2: Gatekeeper policy design for a multi-tenant cluster

I visualized these goals in *Figure 12.2*. The policies we already created need to be adjusted to be "system-level" policies. Instead of saying that they need to apply globally and then make exceptions, we apply them specifically to our system-level namespaces. The policies that grant NGINX the ability to bind to port 443 are part of the system-level policies because the ingress is a system-level capability.

For individual tenants, goal #2 requires that each tenant gets its own set of constraint implementations. These individual constraint template implementation objects are represented by the rounded-corner dashed boxes circling each tenant. This seems repetitive because it is. You are likely to have very repetitive objects that grant the same capabilities to each namespace. There are multiple strategies you can put in place to make this easier to manage:

1. Define a base restrictive set of constraint template implementations, and add each new namespace to the `namespaces` list of the `match` section. This cuts down on the clutter but makes it harder to automate because of having to deal with patches on lists. It is also harder to track because you can't add any metadata to a single property like you can an object.
2. Automate the creation of constraint template implementations when namespaces are created. This is the approach we will take in *Chapter 19, Provisioning a Platform*. In that chapter, we will automate the creation of namespaces from a self-service portal. The workflow will provision namespaces, RBAC bindings, pipelines, keys, and so on. It will also provide the constraint templates needed to ensure restricted access.
3. Create a controller to replicate constraint templates based on labels. This is similar to how Fairwinds RBAC Manager (<https://github.com/FairwindsOps/rbac-manager>) generates RBAC bindings, using a custom resource definition. I've not seen a tool directly for Gatekeeper constraint implementations, but the same principle would work here.

When it comes to managing this automation, the above three options are not mutually exclusive. At KubeCon EU 2021, I presented a session called “I Can RBAC and So Can You!” ([https://www.youtube.com/watch?v=k6J9\\_P-gnro](https://www.youtube.com/watch?v=k6J9_P-gnro)), where I demoed using options #2 and #3 together to make “teams” that had multiple namespaces, cutting down on the number of RBAC bindings that needed to be created with each namespace.

Finally, we'll want to ensure that every namespace that isn't a system-level namespace has constraint implementations created. Even if we're automating the creation of namespaces, we don't want a rogue namespace to get created that doesn't have node security constraints in place. That's represented by the large, dashed, round-cornered box around all of the tenants. Now that we've explored the theory behind building node security policies for a multi-tenant cluster, let's build our policies out.

The first step is to clear out our old policies:

```
$ kubectl delete -f chapter12/enforce_node_policies.yaml
$ kubectl delete -f chapter12/make_cluster_work_policies.yaml
$ kubectl delete -f chapter12/minimal_gatekeeper_constraints.yaml
```

This will get us back to a state where there are no policies. The next step is to create our system-wide policies:

```
$ kubectl create -f chapter12/multi-tenant/yaml/minimal_gatekeeper_constraints.yaml
k8spspallowprivilegeescalationcontainer.constraints.gatekeeper.sh/system-
privilege-escalation-deny-all created
k8spspcapabilities.constraints.gatekeeper.sh/system-capabilities-drop-all
created
k8spspforbiddensysctls.constraints.gatekeeper.sh/system-psp-forbid-all-sysctls
created
k8spsphostfilesystem.constraints.gatekeeper.sh/system-psp-deny-host-filesystem
created
k8spsphostnamespace.constraints.gatekeeper.sh/system-psp-block-all-host-
```

```

namespace created
k8spsphostnetworkingports.constraints.gatekeeper.sh/system-psp-deny-all-host-
network-ports created
k8spspprilegedcontainer.constraints.gatekeeper.sh/system-psp-deny-all-
privileged-container created
k8spspprocmount.constraints.gatekeeper.sh/system-psp-proc-mount-default created
k8spspallowedusers.constraints.gatekeeper.sh/system-psp-pods-allowed-user-
ranges created
k8spsphostfilesystem.constraints.gatekeeper.sh/psp-tigera-operator-allow-host-
filesystem created
k8spspcapabilities.constraints.gatekeeper.sh/capabilities-ingress-nginx created

```

Take a look at the policies in `chapter12/multi-tenant/yaml/minimal_gatekeeper_constraints.yaml`, and you'll see that instead of excluding namespaces in the `match` section, we're explicitly naming them:

```

---
apiVersion: constraints.gatekeeper.sh/v1beta1
kind: K8sPSPAllowPrivilegeEscalationContainer
metadata:
  name: system-privilege-escalation-deny-all
spec:
  match:
    kinds:
      - apiGroups: []
        kinds: ["Pod"]
    namespaces:
      - default
      - kube-node-lease
      - kube-public
      - kubernetes-dashboard
      - local-path-storage
      - tigera-operator
      - openunison
      - activedirectory

```

With our system constraint implementations in place, we'll next want to enforce the fact that all tenant namespaces have node security policies in place before any pods can be created. There are no pre-existing `ConstraintTemplates` to implement this policy, so we will need to build our own. Our Rego for our `ConstraintTemplate` will need to make sure that all of our required `ConstraintTemplate` implementations (in other words, privilege escalation, capabilities, and so on) have at least one instance for a namespace before a pod is created in that namespace. The full code and test cases for the Rego are in `chapter12/multi-tenant/opa`. Here's a snippet:

```

# capabilities
violation[{"msg": msg, "details": {}}] {
    checkForCapabilitiesPolicy
    msg := "No applicable K8sPSPCapabilities for this namespace"
}
checkForCapabilitiesPolicy {
    policies_for_namespace = [policy_for_namespace |
        data.inventory.cluster["constraints.gatekeeper.
sh/v1beta1"].K8sPSPCapabilities[j].spec.match.namespaces[_] == input.review.
object.metadata.namespace ;
        policy_for_namespace = data.inventory.
cluster["constraints.gatekeeper.sh/v1beta1"].K8sPSPCapabilities[j] ]
    count(policies_for_namespace) == 0
}
# sysctls
violation[{"msg": msg, "details": {}}] {
    checkForSysCtlsPolicy
    msg := "No applicable K8sPSPForbiddenSysctls for this namespace"
}
checkForSysCtlsPolicy {
    policies_for_namespace = [policy_for_namespace |
        data.inventory.cluster["constraints.gatekeeper.sh/
v1beta1"].K8sPSPForbiddenSysctls[j].spec.match.namespaces[_] == input.review.
object.metadata.namespace ;
        policy_for_namespace = data.inventory.
cluster["constraints.gatekeeper.sh/v1beta1"].K8sPSPForbiddenSysctls[j]
    ]
    count(policies_for_namespace) == 0
}

```

The first thing to note is that each constraint template check is in its own rule and has its own violation. Putting all of these rules in one `ConstraintTemplate` will result in them all having to pass in order for the entire `ConstraintTemplate` to pass.

Next, let's look at `checkForCapabilitiesPolicy`. The rule creates a list of all `K8sPSPCapabilities` that list the namespace from our pod in the `match.namespaces` attribute. If this list is empty, the rule will continue to the violation and the pod will fail to create. To create this template, we first need to sync our constraint templates into Gatekeeper. Then, we create our constraint template and implementation:

```

$ kubectl apply -f chapter12/multi-tenant/yaml/gatekeeper-config.yaml
$ kubectl create -f chapter12/multi-tenant/yaml/require-psp-for-namespace-
constrainttemplate.yaml
$ kubectl create -f chapter12/multi-tenant/yaml/require-psp-for-namespace-
constraint.yaml

```

With our new policy in place, let's attempt to create a namespace and launch a pod:

```
$ kubectl create ns check-new-pods
namespace/check-new-pods created
$ kubectl run echo-test -ti -n check-new-pods --image busybox --restart=Never
--command -- echo "hello world"
Error from server ([k8srequirepspfornamespace] No applicable
K8sPSPAllowPrivilegeEscalationContainer for this namespace
[k8srequirepspfornamespace] No applicable K8sPSPCapabilities for this namespace
[k8srequirepspfornamespace] No applicable K8sPSPForbiddenSysctls for this
namespace
.
.
```

Our requirement that namespaces have node security policies in place stopped the pod from being created! Let's fix this by applying restrictive node security policies from `chapter12/multi-tenant/yaml/check-new-pods-psp.yaml`:

```
$ kubectl create -f chapter12/multi-tenant/yaml/check-new-pods-psp.yaml
$ kubectl run echo-test -ti -n check-new-pods --image busybox --restart=Never
--command -- echo "hello world"
hello world
```

Now, whenever a new namespace is created on our cluster, node security policies must be in place before we can launch any pods.

In this section, we looked at the theory behind designing node security policies using Gatekeeper, putting that theory into practice for both a single-tenant and a multi-tenant cluster. We also built out sane defaults for our `securityContexts` using Gatekeeper's built-in mutation capabilities. With this information, you have what you need to begin deploying node security policies to your clusters using Gatekeeper.

## Using Pod Security Standards to enforce Node Security

The Pod Security Standards are the “replacement” for Pod Security Policies. I put the term “replacement” in quotes because the PSA isn't a feature comparable replacement to PSPs, but it aligns with a new strategy defined in the Pod Security Standards guide (<https://kubernetes.io/docs/concepts/security/pod-security-standards/>). The basic principle of PSA is that since the namespace is the security boundary in Kubernetes, that is where it should be determined whether pods should run in a privileged or restricted mode.

At first glance, this makes a great deal of sense. When we talked about multitenancy and RBAC, everything was defined at the namespace level. Much of the difficulties of PSPs came from trying to determine how to authorize a policy, so this eliminates that problem.

The concern though is that there are scenarios where you need a privileged container, but you don't want it to be the main container. For instance, if you need a volume to have its permissions changed in an `init` container but you want your main containers to be restricted, you can't use PSA.

If these restrictions aren't an issue for you, then PSA is turned on by default, and all you need to do is enable it. For instance, to make sure a root container can't run in a namespace:

```
$ kubectl create namespace nopriv
$ kubectl label namespace nopriv pod-security.kubernetes.io/enforce=restricted
$ kubectl run echo-test -ti -n nopriv --image busybox --restart=Never --command
-- id
Error from server (Forbidden): pods "echo-test" is forbidden: violates
PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container
"echo-test" must set securityContext.allowPrivilegeEscalation=false),
unrestricted capabilities (container "echo-test" must set securityContext.
capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "echo-test"
must set securityContext.runAsNonRoot=true), seccompProfile (pod or container
"echo-test" must set securityContext.seccompProfile.type to "RuntimeDefault" or
"Localhost")
```

Since we didn't set anything to keep our container from running in a privileged way, our pod failed to launch. PSA is simple but effective. If you don't need the flexibility of a more complex admission controller like Gatekeeper or Kvrno, it's a great option!

## Summary

In this chapter, we began by exploring the importance of protecting nodes, the differences between containers and VMs from a security standpoint, and how easy it is to exploit a cluster when nodes aren't protected. We also looked at secure container design, implemented and debugged node security policies using Gatekeeper, and finally, used the new Pod Security Admission feature to restrict pod capabilities.

Locking down the nodes of your cluster provides one less vector for attackers. Encapsulating a policy makes it easier to explain to your developers how to design their containers and also makes it easier to build secure solutions.

So far, all of our security has been built to prevent workloads from being malicious. What happens when those measures fail? How do you know what's going on inside of your pods? In the next chapter, we'll find out!

## Questions

1. True or false – containers are “lightweight VMs.”
  - a. True
  - b. False

2. Can a container access resources from its host?
  - a. No, it's isolated.
  - b. If marked as privileged, yes.
  - c. Only if explicitly granted by a policy.
  - d. Sometimes.
3. How could an attacker gain access to a cluster through a container?
  - a. A bug in the container's application can lead to a remote code execution, which can be used in a breakout of a vulnerable container, and it is then used to get the kubelet's credentials.
  - b. Compromised credentials with the ability to create a container in one namespace can be used to create a container that mounts the node's filesystem to get the kubelet's credentials.
  - c. Both of the above.
4. What mechanism enforces `ConstraintTemplates`?
  - a. An admission controller that inspects all pods upon creation and updating
  - b. The `PodSecurityPolicy` API
  - c. The OPA
  - d. Gatekeeper
5. True or false – containers should generally run as root.
  - a. True
  - b. False

## Answers

1. b – false; containers are processes.
2. b – a privileged container can be granted access to host resources such as process IDs, the filesystem, and networking.
3. c - Both of the above.
4. d - Gatekeeper
5. b - False



# 13

## KubeArmor Securing Your Runtime

As the popularity of Kubernetes grows, so does the need for robust security measures to protect workloads. We learned how to secure a cluster using RBAC, which allows us to control the access that users have to resources. Using RBAC, we can control what users can execute on a cluster, controlling if someone can create or delete a pod, view logs, view Secrets, etc. We also looked at securing clusters using Gatekeeper policies that can protect nodes by denying the creation of an object that contains a value against security policies like attempting to allow privilege escalation.

While these go a long way to securing clusters, there are certain actions that are often overlooked by many organizations. One of the most important examples is securing the container runtime.

Kubernetes has limited abilities to audit or secure actions that are executed within a container. While Kubernetes can handle certain security requirements like blocking elevated privilege attempts within a container, it doesn't provide a way for operators to limit most actions that are executed in the container. It cannot allow or deny any actions that a user may be able to execute once they exec into a running container, like looking at files, deleting files, adding files, and more. Even worse, most actions executed inside a container are not audited by the Kubernetes API server, which is why they often go overlooked.

In *Chapter 8, Managing Secrets*, we learned about using Vault to store and retrieve secrets. Many people think that if they use a system like Vault, they have secured their secrets from anyone being able to view the data in the secret. It is true that the secret isn't stored in a basic K8s secret resource, where anyone with the required permissions to the namespace would be able to view and decode the secret. Since a Vault secret will be stored in your pod as an environment variable or a file, there is no way to stop someone who has access to exec into the container from viewing the container's environment variables or the files where the Vault secret is stored.

We also need a way to stop certain processes from running in containers. Your organization may have a policy that a container should never run an SSH daemon. Without an add-on tool, you have limited options to secure binaries to that level in a running container.

Sure, you can create pipelines and security checks when an image is created and deny images that don't follow documented security standards, but once the image passes and is deployed, how do you stop someone from executing an exec into the container and adding binaries like the SSH daemon, or even worse, malware or crypto mining tools?

Luckily, a company called AccuKnox has donated a project to the CNCF called **KubeArmor** that provides you with the ability to secure your container runtime. KubeArmor isn't limited to only the runtime; it has a number of other useful features that are related to securing your workloads, including restricting process execution, file access, and more.

In this chapter, we will explain how to deploy KubeArmor and how to use its many features to enhance the security of your clusters. Here are a few topics that we will cover in this chapter:

- What is runtime security?
- Introducing KubeArmor
- Deploying KubeArmor
- Enabling KubeArmor logging
- KubeArmor and LSM policies
- Creating a KubeArmorSecurityPolicy
- Using karmor to interact with KubeArmor

## Technical requirements

This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 8 GB of RAM
- A KinD cluster, preferably a new cluster, with Vault integrated
- Scripts from the chapter13 folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## What is runtime security?

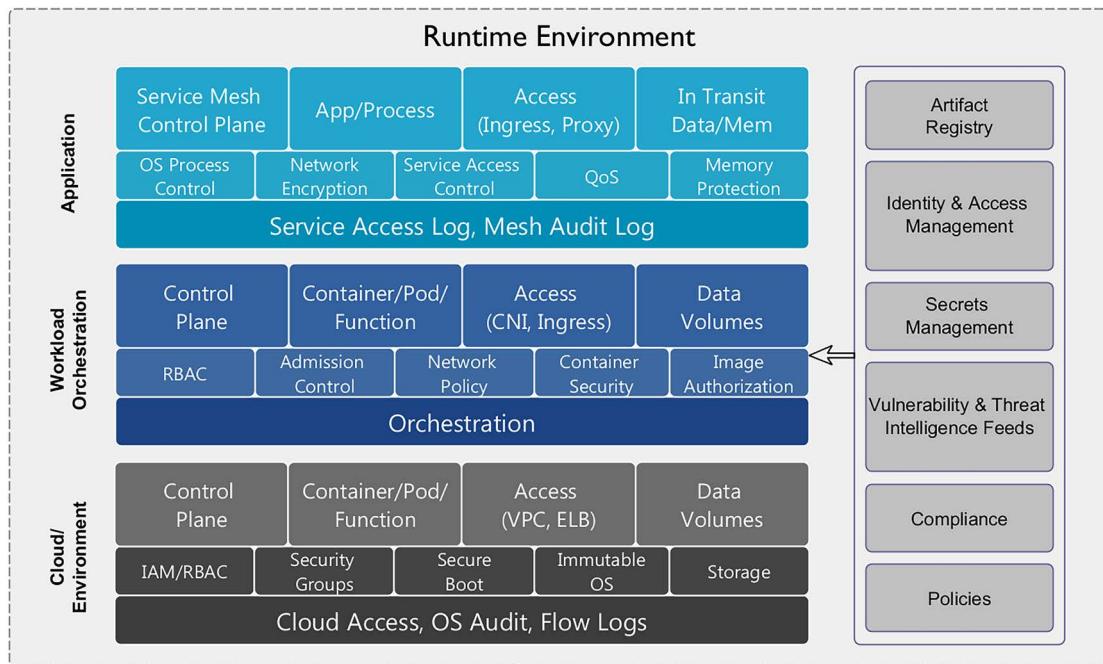
Runtime security is a vital part of security for systems, applications, and data when they are most exposed to attacks during active execution, while they are up and running on your network. Runtimes, often left unmonitored and sometimes lacking in any form of logging or auditing, pose a critical security challenge. Of course, runtime security is not exclusive to containers; it is a requirement for applications, containers, physical servers, virtual machines, and more. Every component within your infrastructure requires continuous monitoring of all potential security risks to quickly detect threats and vulnerabilities posed by potential attackers.

In the face of increasingly sophisticated and dynamic security threats, the reliance solely on static security measures is no longer adequate. This is where runtime security comes in, providing dynamic, real-time protection precisely when it is most crucial: with live systems. Through constant monitoring of the runtime environment, these systems can spot anomalies, suspicious activities, and unauthorized processes and allow or block actions based on a set of policies.

To secure workloads you need to follow key practices, such as permitting only authorized processes in a container, implementing measures to prevent and alert on any unauthorized resource access, or inspecting network traffic to detect any hostile activities. For example, you can limit what processes can access files or directories in a container, denying access to a database file to any process that isn't part of MySQL.

Runtime security has multiple pieces to consider: using KubeArmor is just one of the tools to help protect your workloads and clusters. *Figure 13.1* shows a picture of the components that make up a runtime environment from the CNCF security V2 whitepaper. You can find the whole paper on the CNCF website at [https://www.cncf.io/wp-content/uploads/2022/06/CNCF\\_cloud-native-security-whitepaper-May2022-v2.pdf](https://www.cncf.io/wp-content/uploads/2022/06/CNCF_cloud-native-security-whitepaper-May2022-v2.pdf).

We have covered many of the various tools to secure your runtimes in previous chapters, including network policies, identity and access management, secrets, and policy security using **Gatekeeper**. Combining these options with the added security that KubeArmor supplies, you can secure your clusters from malicious activity.



*Figure 13.1: CNCF runtime security landscape*

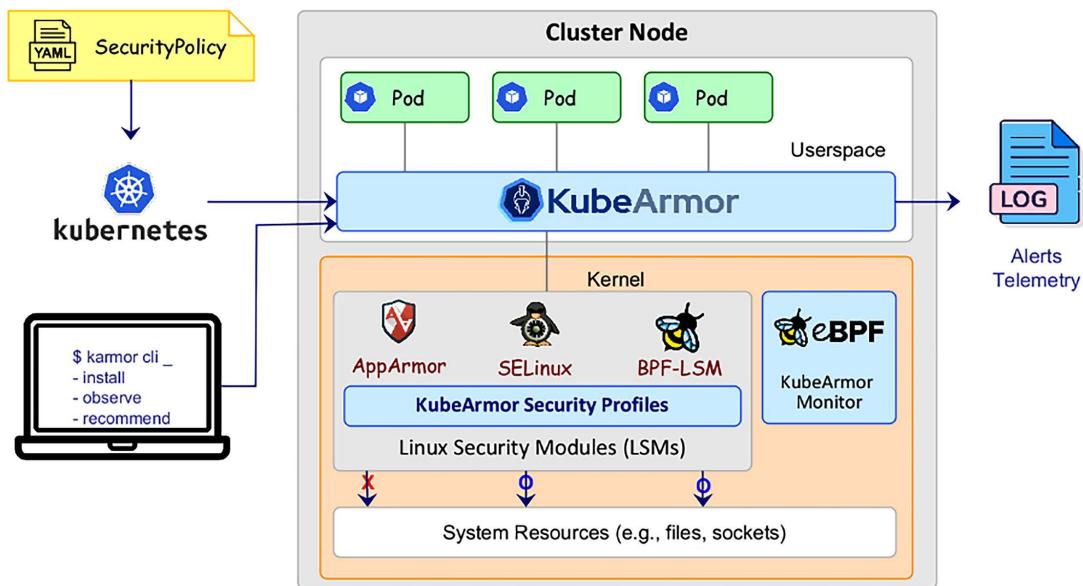
In summary, KubeArmor is a runtime security tool that provides dynamic, real-time protection for your systems against an ever-expanding spectrum of threats and vulnerabilities. Its purpose is to protect the security and stability of your infrastructure, upholding the integrity of your operations in the face of countless cybersecurity threats.

## Introducing KubeArmor

Before we jump into KubeArmor, we need to define a few base concepts you need to be aware of. If you are new to Linux, you may not be familiar with these, and even if you are a Linux veteran, the concepts may still be new to you.

### Introduction to Linux Security

In this chapter, you will primarily see two references that need to be understood to understand how KubeArmor protects clusters. The first term is **eBPF**, which stands for the **extended Berkley Packet Filter**, and the second one is **LSM**, which stands for **Linux Security Module**. In *Figure 13.2*, you can see how access from a pod goes through KubeArmor before it hits the host's kernel. This is what allows KubeArmor to secure your runtimes: sitting between the pod runtime and the kernel, to take action before a request is executed.



*Figure 13.2: KubeArmor's high-level design*

Now, we need to explain, at a high level, what eBPF and LSMs are and how they help to secure a cluster.

Have you ever wondered how Linux handles the constant stream of data in and out of the system? How it monitors performance, and how it protects itself from security risks? Well, that's where eBPF comes in: it handles all these responsibilities and more!

Think of eBPF as a digital traffic cop. Your computer resembles a busy intersection where data is in constant motion. eBPF acts as a traffic controller, capable of controlling the data flow, inspecting it for issues, and tracking ongoing activities.

One advantage of eBPF is its use of “virtual machines,” rather than requiring direct edits to the kernel to add features like monitoring network traffic. eBPF primarily uses programs that are written in a restricted subset of C and are executed within the kernel. While C is the most commonly used language for creating eBPF programs, you can also create them using other languages, including:

- Go
- Lua
- Python
- Rust

Using a language other than C involves transpiling to C or adding the required libraries that abstract the C programming. The final decision on selecting a language is ultimately up to your use cases, standards, and expertise.

In summary, eBPF provides a number of powerful functions, without the need to modify the kernel directly. It is highly secured and isolated, providing a security boundary through its use of virtual machines, similar to a standard virtual machine that runs a full operating system.

The other term we mentioned was **LSM**, which stands for **Linux Security Module**. Two of the most common LSMs today are SELinux, which is primarily used by Red Hat systems, and AppArmor, which is used by a number of systems, including Ubuntu, SUSE, and Debian.

Like the previous eBPF section, we are going to provide a high-level overview of LSMs with a focus on AppArmor since we are using Ubuntu as our server operating system.

LSMs are used to connect the kernel with security policies and modules, providing enforcement of **mandatory access controls (MACs)** and additional security policies within a Linux system. They provide a framework for security, providing hooks into the kernel, and allowing external modules the ability to intercept and secure systems calls, file operations, and other various kernel activities. LSMs are meant to be very flexible and extensible, allowing you to select and create modules that meet your specific requirements, rather than a set of policies that a vendor thinks you should implement.

Given that both eBPF and LSMs offer security functionalities, you might be wondering how, or if, they are different.

Despite their apparent similarities at a high level, they diverge significantly. eBPF employs kernel-embedded virtual machines for execution, allowing the creation of programs capable of executing low-level tasks such as packet filtering, tracing, and performance monitoring. eBPF is commonly employed for network-related tasks, performance optimization, or the development of custom kernel-level functions.

LSMs are components executed by the kernel, operating externally to the kernel itself. The core purpose of LSMs is to enhance system security through the enforcement of policies, including MACs, and other measures designed to safeguard system resources. These modules have the ability to increase cluster security by restricting access to various elements, ranging from files and processes to the flow of network traffic.

You can create policies without a tool like KubeArmor if you know enough about the specific LSM, like AppArmor. Imagine if you use multiple Linux vendors, you would need to know each LSM that each vendor is compatible with. This makes creating policies a challenge, and that's where AppArmorKubeArmor can help.

KubeArmor streamlines the task of creating LSM policies, saving you from having to know the syntax between different LSMs. When you create a policy with KubeArmor, it automatically generates the corresponding LSM policy on the host system. This guarantees that, irrespective of the underlying LSM in operation, you can create a uniform set of policies that provide a consistent security standard across multiple Linux distributions and LSMs.

As you can imagine, KubeArmor uses both eBPF and LSMs to help you secure your environments. Now that we know about what both eBPF and LSMs provide, we can move on to introducing KubeArmor.

## Welcome to KubeArmor

Securing any environment can be a difficult task. When it comes to protecting your clusters, it's not something you can simply address after the fact; it should be a part of the initial design and discussions. Many organizations tend to postpone the security aspect of their environments because of the perceived skills, effort, and time required to implement security solutions. However, it's essential to establish a security foundation before your cluster goes into production. This can present a challenge to organizations, and that's where KubeArmor steps in to assist.

By deploying KubeArmor, you can increase the security and regulatory compliance of containerized applications. KubeArmor serves as a runtime security solution designed to secure containerized workloads by enforcing security protocols and promptly identifying and allowing or denying any activity.

The features of KubeArmor are always evolving: by the time you read this book, KubeArmor will likely have additional features that we do not cover in this chapter.

So, what are some of the features that KubeArmor provides to enhance our security?

### Container security

Containers are the cornerstone of modern applications, making their security a primary objective. This isn't to say that we think non-containerized applications don't need security: of course they do, but non-containerized apps have a lot of security options provided by operating systems and third-party vendors. Containers, as we know them today, are relatively new and many of the toolsets are still catching up.

KubeArmor provides security by continuously monitoring container behavior in real time, mitigating risks like container escapes, binary execution, and privilege escalations.

### Inline mitigation versus post-attack mitigation

There are a number of products on the market today that are very good at detecting anomalies, but they do not have the ability to block or allow the request before it actually executes. This is a post-attack mitigation process, which means the action will be allowed or denied and the anomaly will be logged.

This would be like having a door without a lock and when someone walks into the building, all you would get is an alert from a security camera. The person would still be allowed into the building since there is no lock on the door.

Many of the offerings that only detect events can be integrated with other systems to prevent the action(s). For example, a system detects that someone may have injected a crypto miner into a running container. The event would be detected by the anomaly engine and based on that event, you could trigger a custom written routine to create a network policy that would deny all egress and ingress traffic. This would block the application from network activity, stopping the pod from mining and it would save the current state of the pod since we didn't destroy it; we just stopped all network traffic to and from the pod.

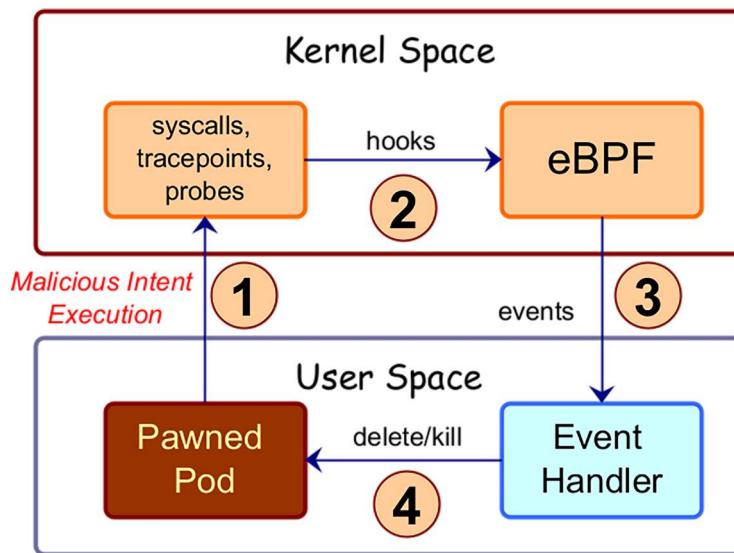


Figure 13.3: Post-attack mitigation

In Figure 13.3, you can see the flow of a post-mitigation attack. The flow of the mitigation is as follows:

- Post-exploit mitigation works by actioning suspicious activity in response to an alert indicating malicious intent.
- The attacker is allowed to execute a binary or other actions. Since they have access, they may be able to disable security controls, logging, etc. to avoid detection.
- Assuming the action has been detected, we send it to an event handler that can execute an action based on the event. However, it's important to point out that by the time a malicious process is actioned, sensitive contents may have already been deleted, encrypted, or transmitted.
- Based on the event, the handler will execute an action like deleting the pod or perform other actions like creating a network policy to block communication without deleting the pod.

One key differentiator of KubeArmor is its ability to not only detect the runtime event but to take action on the event, to block or allow it based on various parameters. Similar to post-attack mitigation, you would still see the attempted action(s) logged, which may be required as evidence to document the malicious activity. However, unlike the previous example of a door with no lock, this door would have a camera and a lock. When someone tries to open the door, the camera will log the attempt: but this time, since the door is locked, the opening action will be denied.

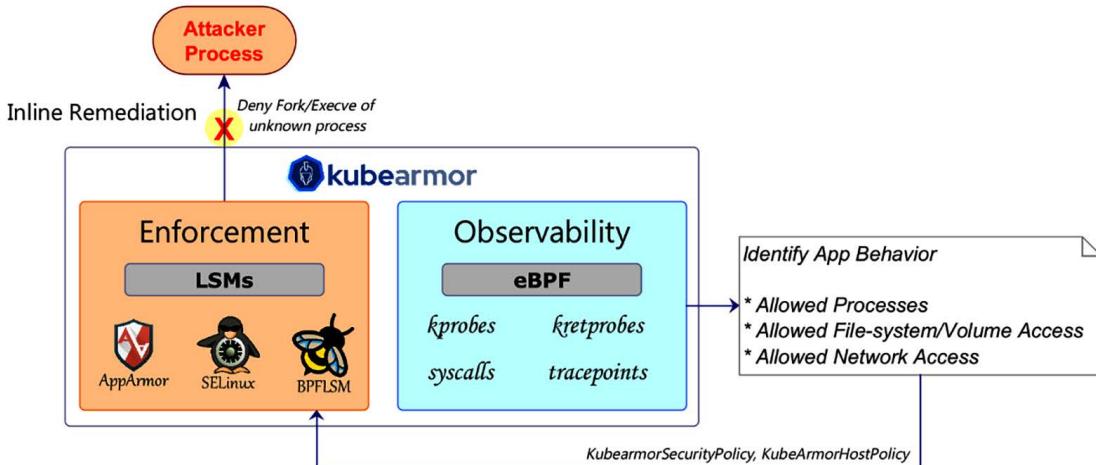


Figure 13.4: Inline mitigation

In *Figure 13.4*, you can see how streamlined the process is: we don't need an external event handler, or any custom components to act on the event. Since KubeArmor handles the events in line, in real time, we can stop an action instantly before an attacker can perform any malicious activity, all in a single product.

As you can see, inline mitigation is a better method of mitigating runtime events. Threats move quickly in today's landscape, and we need to be just as quick at mitigation. If we are attempting to react to events only after they happen, the damage will already have been done and you will just have a log entry that tells you someone has done something malicious.

## Zero-day vulnerability

Zero-day vulnerabilities take time to remediate, not only from a vendor side but from an organizational side as well. If you have the ability to remediate any vulnerability as you wait for an official patch, you should: every minute counts. KubeArmor monitors container activities for any suspicious activity. It can stop activities, without prior knowledge regarding the specific vulnerability or attack pattern.

## CI/CD pipeline integration

KubeArmor easily integrates into continuous integration and continuous deployment (CI/CD) pipelines. Integrating KubeArmor into your pipelines automates security checks through the entire development and deployment lifecycle, delivering a safe and secure image.

## Robust auditing and logging

Logging is very important, and KubeArmor includes comprehensive logs and audit trails of container activities. These logs can be used to report compliance, provide troubleshooting assistance, and assist in forensic examinations.

## Enhanced container visibility

Visibility into container behavior simplifies identifying and responding to security incidents or abnormalities. KubeArmor finds what processes are running in the container and what they are accessing and connecting to.

## Least privilege tenet adherence

KubeArmor is based on the least amount of privilege concept, which is a base security principle. This ensures that containers possess only the necessary permissions and access levels required for their designated functions, consequently curtailing the attack surface and constraining potential damage stemming from a compromised container.

## Policy enforcement

Policies are at the heart of KubeArmor. They provide administrators with the ability to create detailed security policies for containers, fine-tuning the requirements for each different, unique workload. Want to block the ability of any container executing the SSH daemon? Make a simple policy using KubeArmor and no container will be able to execute the SSH daemon.

## Staying in compliance

To help you stay in compliance with standards like CIS, NIST-800-53, and MITRE, KubeArmor includes policies that will secure your clusters based on the defined best standards, all out of the box.

## Policy impact testing

Any policy can be tested before enforcing any settings. This will help you to create a policy that will not cause any workloads to have downtime due to a setting that may impact a running application.

## Multi-tenancy support

It is common for enterprises to run multi-tenant clusters. With multiple teams or applications sharing a Kubernetes cluster, you need to provide a secure environment to all of the users, stopping any attacks or effects that a workload in one namespace may have on another namespace. KubeArmor provides isolation and security among tenants by implementing unique policies at the container level. It's an important tool for securing containerized applications, providing compliance with regulatory requirements, and providing a defence against a large spectrum of security threats.

Now, let's talk about how we can deploy KubeArmor in a cluster and how to use it to secure our workloads.

## Cluster requirements for the exercises

As you learned in *Chapter 2*, KinD is a Kubernetes cluster that runs the components in containers. This nesting does mean that some add-ons like KubeArmor need to have some extra steps to function correctly.

For this chapter, we suggest a new cluster. If you have a previous cluster with Vault installed already, you should delete that cluster and start over with a new one. If you do need to delete an existing cluster, you can execute `kind delete cluster --name cluster01` to delete it and then use the scripts to deploy a new cluster that includes Vault integration.

To make it easier to deploy, we have included all the required scripts in the `chapter13/cluster` directory. To deploy a new cluster, execute `create-cluster.sh` in the `cluster` directory.

We also need Vault for one of the examples. If you want to run the example, you will need to add Vault to your cluster. We have provided an automated Vault deployment in the `chapter13/vault` directory called `deploy-vault.sh`.

Once both have been executed, you will have a brand-new cluster integrated with Vault. It will take time for Vault to deploy fully, so please wait until all of the pods have been created to move on to deploying KubeArmor in the cluster.

## Deploying KubeArmor

Before we can use KubeArmor on our KinD cluster, we will need to patch Calico and the `kubearmor-relay` deployment to work with KinD. AppArmor requires some changes for certain workloads to deploy and run correctly in a KinD cluster. In a standard cluster, these patches would not be required: and once they are deployed, KubeArmor will work as it would on a standard Kubernetes cluster.

KubeArmor can be easily deployed using a single binary, called `karmor`, or via Helm charts. For the book exercises, we will use the `karmor` utility to install KubeArmor. Both deployment methods offer the same protection and configuration options, and once deployed, you interact with KubeArmor the same way, regardless of the deployment method.

We have included a script in the `chapter13` folder called `kubearmor-patch.sh` that will download `karmor`, patch Calico and the `kubearmor-relay` deployments, and deploy KubeArmor.



KubeArmor installs on most Kubernetes clusters without any issues. Since we are using a cluster built on KinD, we need to make a few tweaks to allow AppArmor to work as expected. The scripts do this work for you. The majority of the fixes are to add an annotation to a few deployments, like the Calico Typha controller to unconfined mode. We will discuss the patched deployments and what unconfined provides in this section.

The script downloads `karmor` and moves it to the `/usr/local/bin` directory on the host. This is the utility that we will use to install KubeArmor and interact with it once it has been deployed in the cluster.

Since KubeArmor leverages LSMs, all nodes require an installed LSM, like AppArmor, for KubeArmor to function. On most Ubuntu deployments, AppArmor is already deployed, but since our Kubernetes cluster is running containerized, AppArmor is not included in the image. To resolve this, we need to add AppArmor to our nodes: the script takes care of this by executing `docker exec` in each container that updates the apt repositories, installs AppArmor, and restarts containerd.

The next step in the script will patch the `calico-typha` deployment with an AppArmor policy that is unconfined. Running policies as unconfined means they don't have an AppArmor profile assigned to them, or they are assigned a profile that does not impose any significant restrictions. This allows the process to operate with the standard Linux discretionary access controls, without additional restrictions from AppArmor.

As we mentioned previously, you wouldn't need these patch deployments in a standard Kubernetes cluster, but since we are using KinD, we need to patch `calico-typha` to work correctly with KubeArmor running in our KinD cluster.

With all the requirements and changes deployed, the script continues to install KubeArmor using `karmor install`. This will take a few minutes to deploy all the components and, during the deployment, you will see each step that `karmor` is executing:

```
namespace/kubearmor created
🛡 Installed helm release : kubearmor-operator
⌚ KubeArmorConfig created
⌚ This may take a couple of minutes
🛡 KubeArmor Snitch Deployed!
🛡 KubeArmor Daemonset Deployed!
⌚ Done Checking , ALL Services are running!
⌚ Execution Time : 2m1.206181193s

⌚ Verifying KubeArmor functionality (this may take upto a minute) -.

🛡 Your Cluster is Armored Up!
```

You will see that the installer creates a number of Kubernetes resources, including CRDs, a ServiceAccount, RBAC, Services, and Deployments. Once all resources have been created, it will verify the deployment was successful by telling you that `Your Cluster is Armored Up!`

After a successful deployment, you will have additional pods running in the `kubearmor` namespace, the controller, the relay, and a `kubearmor` pod, one for each of your nodes:

	kubearmor-controller-7cb5467b99-wmlz5	2/2	Running	0	6m
	kubearmor-gvs5f	1/1	Running	0	6m6s
	kubearmor-lpkj6	1/1	Running	0	6m6s
	kubearmor-relay-5ccb6b6ffb-c4dlm	1/1	Running	0	
		6m6s			



You can also deploy KubeArmor using Helm charts. If you want to know more about deploying KubeArmor with Helm, read more about it in KubeArmor's Git repository at [https://github.com/kubearmory/KubeArmor/blob/main/getting-started/deployment\\_guide.md](https://github.com/kubearmory/KubeArmor/blob/main/getting-started/deployment_guide.md).

Each of the pods has a specific function that is explained below:

- **kubearmory:** A daemonset that deploys the kubearmory pod on each node in the cluster. It is a non-privileged DaemonSet with capabilities that allow it to monitor pods and containers and the host.
- **kubearmory-relay:** KubeArmor's relay server collects all messages, alerts, and system logs generated by KubeArmor in each node, and then it allows other logging systems to simply collect those through the service of the relay server. The relay server plays a critical role in ensuring efficient and centralized security monitoring and data collection within Kubernetes environments, making it easier for organizations to maintain robust security postures in their containerized infrastructures.
- **kubearmory-controller:** Admission controller for KubeArmor policy management, including policy management, distribution, synchronization, and logging.

For the chapter, we've opted for the karmor binary installation due to its ease of use, making it a convenient choice for deploying KubeArmor quickly. Additionally, we need the same karmor binary for the exercises throughout the chapter. This approach not only simplifies the learning process but also underscores the versatility and practicality of the karmor tool in managing KubeArmor deployments and operations.

Now that we have KubeArmor deployed, we will discuss configuring logging before we start to create policies to secure our cluster.

## Enabling KubeArmor logging

By default, KubeArmor is not enabled to log events or alerts to `STDOUT`. Later in the chapter, we will go over how to watch logging events in the console interactively, which is useful for troubleshooting issues with policies in real time, but it is not an efficient way to log a history of policy events.

Most logging solutions made for Kubernetes will pick up logged events from `STDOUT` and `STDERROR`. By enabling KubeArmor's logging options, you will have a history of events in your standard logging solution. Using these events, you can create alerts and produce a history of changes and events when a security audit occurs.

KubeArmor offers three events that can be logged:

- **Alert:** When a policy is violated, an event will be logged with information including the action, policy name, pod name, namespace, and more
- **Log:** Creates a log event when a pod executes a syscall, file access, process creation, network socket events, etc
- **Message:** Creates log entries generated by the KubeArmor daemon

The process to enable logging is different between deployments of KubeArmor. We used the karmor executable to deploy, so we need to edit the deployment, adding two environment variables: one for standard logging, `ENABLE_STDOUT_LOGS`, and one for alerts, `ENABLE_STDOUT_ALERTS`. Both of these require a value of true to be enabled. To enable logging, we need to edit or patch the deployment of the relay server. This has already been done by our included script that deployed KubeArmor. The script will use a standard YAML file to patch the deployment. The patching file is shown below:

```
spec:  
  template:  
    spec:  
      containers:  
        - name: kubearmor-relay-server  
          env:  
            - name: ENABLE_STDOUT_LOGS  
              value: "true"  
            - name: ENABLE_STDOUT_ALERTS  
              value: "true"
```

Next, using the patch file, the script executes a `kubectl patch` command:

```
kubectl patch deploy kubearmor-relay -n kubearmor --patch-file patch-relay.yaml
```

Once patched, all of the enabled logs will be shown in the relay-server pod logs. An example of an event is shown below:

```
{"Timestamp":1701200947,"UpdatedTime":"2023-11-28T19:49:07.625696Z",  
"ClusterName":"default","HostName":"cluster01-worker","NamespaceName":  
"my-ext-secret","Owner": {"Ref": "Pod", "Name": "nginx-secrets", "Namespace":  
"my-ext-secret"}, "PodName": "nginx-secrets", "Labels": "app=nginx-web",  
"ContainerID": "88f324db1f6ffa01f42b2811288b6f8b0e66001f41c5101ce578f69c  
bd932e5e", "ContainerName": "nginx-web", "ContainerImage": "docker.io/library/nginx  
:latest@sha256:10d1f5b58f74683ad34eb29287e07dab1e90f10af243f151bb50aa5  
dbb4d62ee", "HostPPID": 1441261, "HostPID": 1441503, "PPID": 43, "PID": 50,  
"ProcessName": "/usr/bin/cat", "PolicyName": "nginx-secret", "Severity":  
"1", "Type": "MatchedPolicy", "Source": "/usr/bin/cat /etc/secrets/", "Operation":  
"File", "Resource": "/etc/secrets/", "Data": "syscall=SYS_OPENAT fd=-100  
flags=0_RDONLY", "Enforcer": "AppArmor", "Action": "Block", "Result": "Permission  
denied"}
```

From the example log entry, you can see that the information for the event contains everything that you need to know for the activity. It includes a review of the activity including:

- The source namespace
- The Kubernetes host
- Pod name
- Process name

- The violated policy name
- Operation
- The resource that was acted on
- The results of the action, allowed or denied

By itself, this may not include additional information that you need to know the full activity from end to end. For example, it doesn't include the user of the initial activity. Like many events in Kubernetes, you need to correlate events from multiple log files to create the full story of the executed activity. In this example, you would need to correlate the activity from the event that audited the initial `kubectl exec` command with the pod and time of the runtime violation that KubeArmor logged.

At this point, we have KubeArmor configured, and we can get into creating and testing policies.

## KubeArmor and LSM policies

As we mentioned, KubeArmor is a tool that helps you create policies for Linux LSMs. Since it creates standard LSMs, any policy that you create and deploy will be stored on the node(s) where the OS stores LSM policies. Since we are using KinD, the nodes are running Ubuntu, which uses AppArmor as the LSM. AppArmor policies are stored in the `/etc/apparmor.d` directory on the host.

The output below shows an example directory from a node that has had a few KubeArmor policies created:

```
kubearmor-local-path-storage-local-path-provisioner-local-path-provisioner  
kubearmor-my-ext-secret-nginx-secrets-nginx-web  
kubearmor-calico-apiserver-calico-apiserver-calico-apiserver  
kubearmor-tigera-operator-tigera-operator-tigera-operator  
kubearmor-calico-system-calico-kube-controllers-calico-kube-controllers  
kubearmor-vault-vault-agent-injector-sidecar-injector  
kubearmor-calico-system-calico-node-calico-node  
kubearmor-vault-vault  
kubearmor-calico-system-csi-node-driver-calico-csi  
kubearmor-calico-system-csi-node-driver-csi-node-driver-registrar      lsb_  
release  
kubearmor-cert-manager-cert-manager-cainjector-cert-manager-cainjector  
nvidia_modprobe  
kubearmor-cert-manager-cert-manager-cert-manager-controller  
tunables
```

If you looked at any of the policies, you would see a standard AppArmor formatted policy. We aren't going to go into the details of creating an AppArmor policy, but the output below shows an example of a policy created by KubeArmor:

```
## == Managed by KubeArmor == ##
#include <tunables/global>profile kubearmor-vault-vault flags=(attach_
disconnected,mediate_deleted) {
    ## == PRE START == ##
    #include <abstractions/base>
    umount,
    file,
    network,
    capability,
    ## == PRE END == ##

    ## == POLICY START == ##
    ## == POLICY END == ##

    ## == POST START == ##
    /lib/x86_64-linux-gnu/*,*/ rm,

    deny @{PROC}/*,*/[0-9]*,sys/kernel/shm*} wkx,
    deny @{PROC}/sysrq-trigger rwkIx,
    deny @{PROC}/mem rwkIx,
    deny @{PROC}/kmem rwkIx,
    deny @{PROC}/kcore rwkIx,

    deny mount,

    deny /sys/[^f]*/** wkIx,
    deny /sys/f[^s]*/** wkIx,
    deny /sys/fs/[^c]*/** wkIx,
    deny /sys/fs/c[^g]*/** wkIx,
    deny /sys/fs/cg[^r]*/** wkIx,
    deny /sys/firmware/efi/efivars/** rwkIx,
    deny /sys/kernel/security/** rwkIx,
    ## == POST END == ##
}
```

It's possible that your nodes may have policies that were not created using KubeArmor. In order to know what policies were created and managed by KubeArmor and what policies were not, you need to look at the first line of the policy. If the policy was created by KubeArmor, it will start with `## == Managed by KubeArmor == ##`, and policies that do not start with this line were not created by KubeArmor.

Now let's move on to the next section, on creating a `KubeArmorSecurityPolicy`.

## Creating a KubeArmor Security Policy

It's time to create some policies! When KubeArmor is deployed, it creates three Custom Resource Definitions and one of those is `kubearmorpolicies.security.kubearmor.com`, which is used to create new policy resources.

Let's jump right into an example policy. You do not need to deploy this to your cluster; it's being used to show an example policy.

If we want to block any attempted access to create a file in the `/bin` directory of our containers in the `demo` namespace, the format of this policy is shown below:

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: block-write-bin
  namespace: demo
spec:
  action: Block
  file:
    matchDirectories:
      - dir: /bin/
        readOnly: true
        recursive: true
  message: Alert! An attempt to write to the /bin directory denied.
```

Breaking down this policy, we can see that it's using the `security.kubearmor.com/v1` API and it's a `KubeArmorPolicy` type. The metadata section has common options, naming the object `block-write-bin` in the `demo` namespace.

The `spec` section is where we actually start to create a new policy. There are a number of options available to you for creating policies. You can learn about all of the options on the KubeArmor website: [https://docs.kubearmor.io/kubearmor/documentation/security\\_policy\\_specification](https://docs.kubearmor.io/kubearmor/documentation/security_policy_specification).

The action `spec` allows you to define what the policy enforces. The options are `Block`, `Allow`, and `Audit`: each of the options is described in the table below.

Available Action	Description
Block	Tells KubeArmor to block the actions that are included in the policy (default if no action is provided)
Allow	Tells KubeArmor to allow the actions that are included in the policy
Audit	Tells KubeArmor to only audit the actions of the policy. The actions on the policy will be allowed, but in our example, we would receive a logged event when someone creates a file under the <code>/bin</code> directory. This is handy for testing how a policy will affect a workload in the cluster.

Table 13.1: Available actions for policies

KubeArmor operates on the principle of enforcing the least permissive access. When you specify the allow action in a policy, it generates an allow list that permits access exclusively to the object(s) specified within the policy. For instance, if you were to establish an allow policy for a file named `demo/allowed-file`, any process within the container would have permission to access that particular file. All other files accessed within the container would trigger an audit event because they do not belong to the allowed list.

You might be questioning the example, where if you set up an allow policy and someone tries to read a different file, it won't reject the request but will instead log the access for auditing purposes. The default security posture, within an allow policy, pertains to how it manages access attempts not listed in the allowed entries. By default, KubeArmor's security posture is set to audit mode.

It's crucial to bear in mind that when you establish an allow policy, any access requests that would typically be denied will not face denial; instead, they will merely trigger an audit alert. As a result, if you configure an allow rule to restrict access to a specific file, all other files will remain accessible.

The default posture behavior can be changed at the global level or on a per namespace level. To make the global default posture block, instead of audit, you need to edit the KubeArmor config, which is stored in a ConfigMap called `kubearmor-config` in the `kubearmor` namespace. In the config, you can set the default security posture for each option, file, network, and capabilities:

```
defaultFilePosture: block      #Can be block or audit  
defaultNetworkPosture: block    #Can be block or audit  
defaultCapabilitiesPosture: block #Can be block or audit
```

Depending on your cluster configuration and the logic design of a cluster, you may want to change the default posture on specific namespaces. To set the policy on a namespace, you need to add an annotation of `kubearmor-file-posture=<value>`. If we wanted to add a policy to an existing `demo` namespace, we would just need to run `kubectl annotate`, as shown below:

```
kubectl annotate ns default kubearmor-file-posture=block --overwrite
```

If you were creating a new namespace using a manifest, you would just add the annotation to the manifest before applying the file to create the namespace.

After defining the policy action, we need to add what objects we want to block, allow, or audit.

There are four objects that we can create policies for. They are:

- Process
- File
- Network
- Capabilities



The KubeArmor website has documentation on the policies and options located at [https://docs.kubearmor.io/kubearmor/documentation/security\\_policy\\_specification](https://docs.kubearmor.io/kubearmor/documentation/security_policy_specification).

In our example, our goal is to prevent any form of write access within the `/bin` directory. To achieve this, we will utilize the `file` object. Following the object declaration, you specify a `match` condition that will trigger the policy action. In this instance, we've configured the `matchDirectories` action specifically for the `/bin` directory, indicating to KubeArmor that the policy's evaluation should only occur if the action is within that directory.

Continuing, there are optional settings for `readOnly` and `recursive`. In our scenario, we have enabled both. When `readOnly` is set to true, it permits the reading of any file located under `/bin`, but any other actions will be denied. Enabling the `recursive` option instructs KubeArmor to assess both the `/bin` directory and all of its subdirectories.

Finally, you can define the `message` option, which will add a custom message in the KubeArmor logs when the policy has been triggered. In our example, we added the `message` option to add “**Alert! An attempt to write to the `/bin` directory denied.**” when an attempt is made to do anything other than read a file under the directory.

You may be wondering about the `allow` action and how we said it creates an allow list, allowing only access to the objects in the policy, and denying access to every other file in the container. The example of a single file isn't a great example for the real world, but it does explain what you granted access to and what was denied by `allow` policies. An `allow` policy will lock down a container tightly when used correctly. When used incorrectly, your application would likely crash, being denied access to a file not in the allowed list. You can imagine that creating an allow list for an app could require a large number of objects, many of which would be a challenge to find on your own.

Let's use a real-world example policy to close out this section.

Foowidgets want to secure their secrets. They have created a policy that all secrets must be stored in an external secret manager like Vault. As we discussed in the secrets chapter, you can read your secret in from Vault without having a base64-encoded secret in the namespace. A lot of people assume this secures your secret, but they overlook that someone will be able to exec into the container and read most files, including files that store secrets.

How do we enhance the security of our secrets, even using an external secrets manager like Vault? The answer is KubeArmor!

We can address Foowidgets' requirement by creating a policy that will allow only the required running process access to the file that contains the secret, while any other process will be denied.

In the `chapter13/nginx-secrets` directory, there is a script called `create-nginx-vault.sh`, which will create an NGNIX webserver that will display a secret file and the contents when you open a webpage path, `/secrets/myenv`. The secret that is shown on the page is pulled down from Vault and mounted in the pod using a volume at `/etc/secrets/myenv`.

When you execute the script, the last line will show you the `nip.io` URL for the webserver. Open the URL in any browser, or curl the `http://secret.<nip.io>/secrets/myenv` URL to prove that the secret shows in the output. You should see output similar to the below:

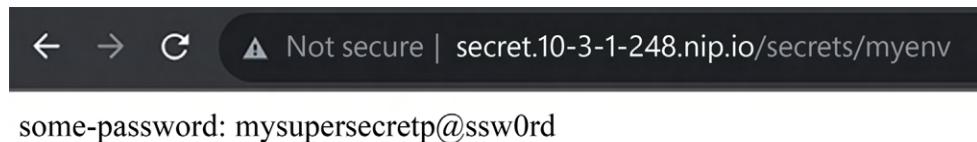


Figure 13.5: NGINX showing contents of secret file

Verifying that the container is working as expected, we can exec into the container and attempt to read the secret file using `cat`:

```
kubectl exec -it nginx-secrets -n my-ext-secret -- bash
```

The secret is mounted in the pod at `/etc/secrets` in the `myenv` file:

```
cat /etc/secrets/myenv
```

This will output the contents of the file:

```
some-password: mysupersecretp@ssw0rd
```

Hold on! I thought that employing an external secret manager would ensure the security of Kubernetes secrets. Although it may not store the data in an easily discoverable Secret within the namespace, an individual with container execution access can still retrieve the secret.

This issue is one of the shortcomings of systems like Vault; simply using Vault doesn't necessarily guarantee the security of the secret.

To illustrate this with a real-world scenario, let's consider a requirement by our company, Foowidgets. They want to restrict secret access exclusively to the processes that require access to the secret. This can be accomplished by creating a new KubeArmor policy that permits only the application to access the file containing the secret. In our example container, we intend to grant the NGINX process permission to read the secret file while preventing other processes from doing so.

To accomplish this, we have created an example policy file called `nginx-secrets-block.yaml`. This will deploy into the `my-ext-secret` namespace:

```
apiVersion: security.kubearmor.com/v1
kind: KubeArmorPolicy
metadata:
  name: nginx-secret
  namespace: my-ext-secret
spec:
  selector:
    matchLabels:
      app: nginx-web
  file:
```

```

matchDirectories:
  - dir: /
    recursive: true
  - dir: /etc/nginx/
    recursive: true
    fromSource:
      - path: /usr/sbin/nginx
  - dir: /etc/secrets/
    recursive: true
    fromSource:
      - path: /usr/sbin/nginx
  - dir: /etc/nginx/
    recursive: true
    action: Block
  - dir: /etc/secrets/
    recursive: true
    action: Block
process:
  matchPaths:
    - path: /usr/sbin/nginx
action:
  Allow

```

To show the policy in action, we have included a script called `redeploy-nginx-vault.sh` in the `chapter13/nginx-secrets` directory, which will delete the previous NGINX deployment and then create a new deployment with the KubeArmor policy to secure the Vault secret used by NGINX.

Execute the script and wait until the new deployment and policy have been created. It's important that we confirm that the outcome is what we expect from the new policy. To verify the policy, we will attempt to access the secret by executing `kubectl exec -it nginx-secrets -n my-ext-secret -- bash` to enter the pod.

Once in the pod, we can attempt to view the secret by using `cat`:

```
cat /etc/secrets/myenv
```

You'll notice that access to the file is no longer allowed. KubeArmor will intercept the request and, based on the policy, refuse access to the `/etc/secrets/myenv` file:

```
root@nginx-secrets:/etc/secrets# cat /etc/secrets/myenv
cat: /etc/secrets/myenv: Permission denied
```

Take note that even though you have root privileges within the container, you can't access the `myenv` file in the `/etc/secrets` directory. The policy blocks any access not explicitly allowed to the directory or its files.

So far, everything seems to be going well. However, now we must verify the website to ensure that the secret information still appears. If the site displays the same content as before implementing our policy, it demonstrates that the secret is allowed to be read by the NGINX binary. To verify this, navigate to the same URL you previously used to test the site, either by browsing or using the curl command. If you still have the same browser window open, simply refresh it.

The screenshot below verifies that the website is functioning correctly and continues to display the value stored in the myenv file in the /etc/secrets directory. This confirms that the NGINX binary has the necessary access to the secret file:

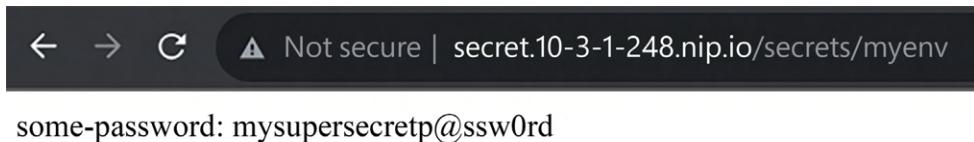


Figure 13.6: NGINX can still read secret

KubeArmor simplifies the creation of LSM policies for both developers and operators. The potential applications are endless, granting you the capability to enhance the security of workloads down to the granularity of individual files or processes. Now that we've covered the process of policy creation, let's proceed to explore the primary tool that you'll employ to engage with KubeArmor.

## Using karmor to interact with KubeArmor

We installed KubeArmor using the karmor utility. Along with installing and uninstalling KubeArmor in a cluster, it is used for a number of other actions. The table below is an overview of the main options that you should understand. Each one will be explained in detail in its own section.

Options	Description
Install	Installs KubeArmor in a cluster
Logs	Provides an interactive method to view logs, or to send the logs to a file
Probe	Lists the support features for the cluster
profile	Runs an interactive utility that displays the process, file, network, and syscalls that KubeArmor has observed
recommend	Creates a directory that contains recommended policies that can be deployed in a cluster. This will download additional containers to create the recommendations. It could take some time depending on the number and size of the running containers.
selfupdate	Updates the karmor CLI
summary	Shows observations from the discovery engine
sysdump	Used to collect a system dump to help troubleshooting
uninstall	Uninstalls KubeArmor from the cluster

<code>version</code>	Shows the version of the karmor binary
<code>Vm</code>	Used for commands that can be used against VMs that run with Kubevirt, which runs <code>kvmservices</code>

Table 13.2: *karmor command options*

The list may make KubeArmor look like it doesn't have a lot of options, but most of the options are incredibly powerful and some will take time to run in larger clusters. In the next sections, we will explain the karmor options and what they provide to secure your cluster.

## karmor install and uninstall

As you would imagine, the `karmor install` command will deploy KubeArmor into the cluster from your current `kubeconfig` file, while the `karmor uninstall` command will remove KubeArmor from the cluster.

We do need to call out that `karmor uninstall`, by default, will remove KubeArmor from the cluster, but it will leave any LSM policies that were created on the hosts in an inactive state. To fully remove KubeArmor from the cluster, including all created policies, you need to add the `--force` flag to the `uninstall` command.

## karmor probe

The probe option will list the KubeArmor features in the current cluster.

When you check for the supported probes, karmor will output information including each node and its active LSM and the default posture of each namespace and pod.

## karmor profile

KubeArmor's profile provides you with an interactive console to view what processes, files, network connections, and syscalls are in use. The screen below shows an abbreviated output from a probe with the `File` tab selected.

Max Rows: 30					
Process	File	Network	Syscall		
Namespace	ContainerName	ProcessName	File	Result	
calico-apiserv...	calico-apiserver	/code/filecheck	/	Passed	
calico-system   69ffb568cc9cfe5a083120519b8...  /bin/calico-node   /sys/kernel/mm/transparent_hugepage/hpage_...  Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /bin/calico-node   /usr/bin/calico-node   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /bin/calico-node   /lib64/   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /bin/calico-node   /etc/   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /usr/bin/calico-node   /sys/kernel/mm/transparent_hugepage/hpage_...  Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /usr/bin/calico-node   /usr/bin/calico-node   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /usr/bin/calico-node   /lib64/   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /usr/bin/calico-node   /etc/   Passed					
calico-system   69ffb568cc9cfe5a083120519b8...  /usr/sbin/ipset   /etc/   Passed					

Figure 13.7: *karmor profile output*

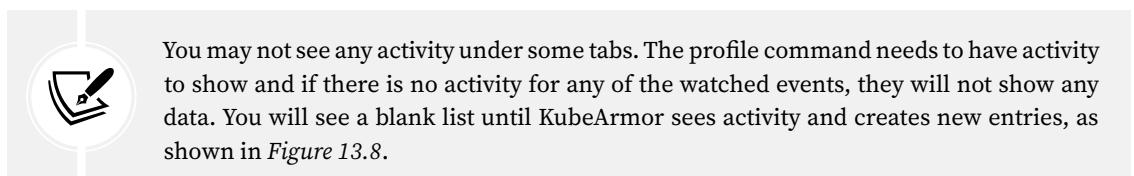
By default, a probe will output information for all namespaces. If you have clusters with a lot of namespaces and pods, you can limit the output to a single namespace or certain pods.

To limit the output to a single namespace, add the option `-n` or `--namespace <namespace to prove>`, and to limit the output to just a pod, use `-p` or `--pod <pod name to probe>`.

If you want to see this in action, let's say that you wanted to watch a new namespace called `demo` for activity. You would execute the `recommend` command and add `-n demo`.

On the host, execute the probe command shown below:

```
karmor profile -n demo
```



Max Rows: 30					
Process	File	Network	Syscall	Process	Result
Namespace	ContainerName	ProcessName		Process	Result

*Figure 13.8: KubeArmor's profile console*

Open another connection to the host, so we can create a new NGINX deployment. There is a script in the `chapter13/nginx` directory called `nginx-ingress.sh` that will create a new namespace called `demo` with an NGINX deployment and an ingress rule. Execute the script and, at the end, it will show you the ingress URL to use.

Now that we have created a deployment, your other terminal should show activity in the **Process** tab, as shown in *Figure 13.9*.

Max Rows: 30					
Process	File	Network	Syscall	Process	Result
Namespace	ContainerName	ProcessName		Process	Result
demo	096c339e624c710c754ff36e52a5..	/bin/date		/bin/	Passed
demo	096c339e624c710c754ff36e52a5..	/bin/rm		/bin/	Passed
demo	096c339e624c710c754ff36e52a5..	/usr/bin dirname		/usr/bin/	Passed
demo	096c339e624c710c754ff36e52a5..	/usr/bin id		/usr/bin/	Passed
demo	096c339e624c710c754ff36e52a5..	/usr/bin openssl		/usr/bin/	Passed
demo	nginx	/bin/cp		/bin/	Passed
demo	nginx	/bin/date		/bin/	Passed
demo	nginx	/bin/mount		/bin/	Passed
demo	nginx	/bin/rm		/bin/	Passed
demo	nginx	/bin/sed		/bin/	Passed
demo	nginx	/usr/bin dirname		/usr/bin/	Passed
demo	nginx	/usr/bin id		/usr/bin/	Passed
demo	nginx	/usr/bin jq		/usr/bin/	Passed
demo	nginx	/usr/bin openssl		/usr/bin/	Passed
demo	nginx	/usr/bin realpath		/usr/bin/	Passed

*Figure 13.9: KubeArmor's profile console*

This will populate events in the **Process** tab as the NGINX pod starts up and processes are started. In your other window, you will see the profile update, in real time, with the processes that started in the demo namespace.

KubeArmor's probe is a powerful tool that provides information that would otherwise be very challenging to collect.

## karmor recommend

The **recommend** command is to provide security policy recommendations based on established industry compliance standards and attack frameworks like CIS, MITRE, NIST, STIGs, and various others. All of the workloads specified in the **recommend** command will be tested against any policy templates included with KubeArmor.

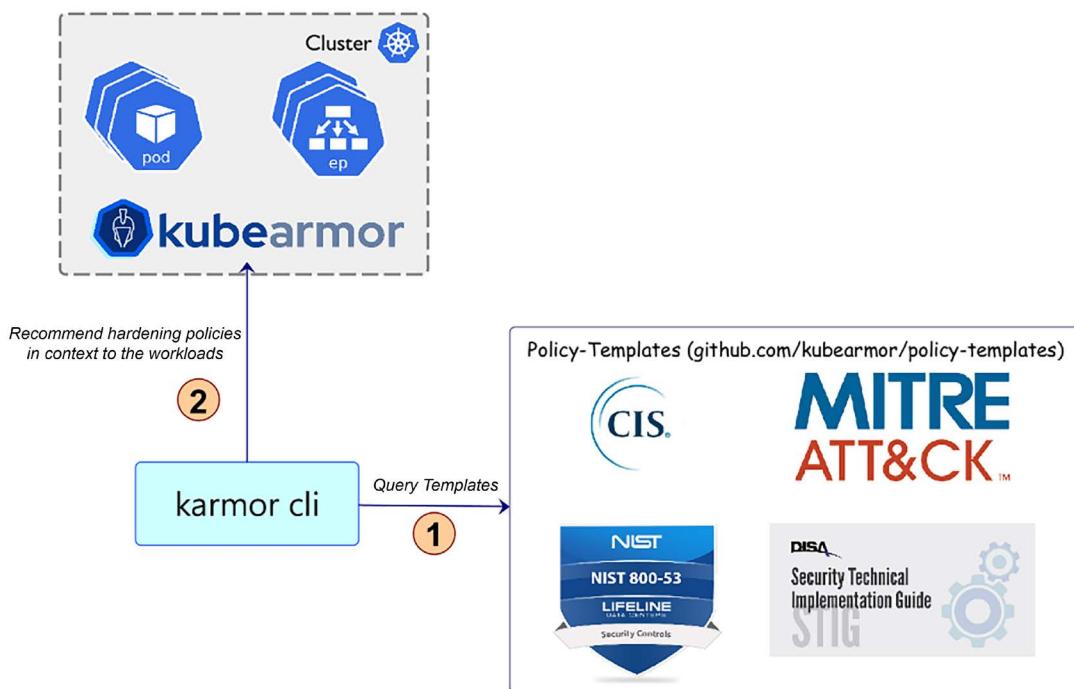


Figure 13.10: KubeArmor recommend policies

Since each pod and container are evaluated, you have the option to filter the execution to target not only the cluster but by namespace, container image, or pod. An example output of `karmor recommend` running is shown in *Figure 13.11*:

```

INFO[0001] Found outdated version of policy-templates    Current Version=v0.2.3
INFO[0001] Downloading latest version [v0.2.4]
INFO[0002] policy-templates updated                      Updated Version=v0.2.4
INFO[0002] pulling image                                image="registry.k8s.io/coredns/coredns:v1.10.1"
v1.10.1: Pulling from coredns/coredns
Digest: sha256:a0ead06651cf580044ae0a0feba63591858fb2e43ade8c9dea45a6a89ae7e5e
Status: Image is up to date for registry.k8s.io/coredns/coredns:v1.10.1
INFO[0012] dumped image to tar                          tar=/tmp/karmor4125017639/pCSntFrF.tar
INFO[0014] No runtime policy generated for kube-system/coredns/registry.k8s.io/coredns/coredns:v1.10.1
created policy out/kube-system-coredns/registry-k8s-io-coredns-v1-10-1-maint-tools-access.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-trusted-cert-mod.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-system-owner-discovery.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-write-under-bin-dir.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-write-under-dev-dir.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-cronjob-cfg.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-pkg-hmnr-exec.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-k8s-client-tool-exec.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-remote-file-copy.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-write-in-shm-dir.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-write-etc-dir.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-shell-history-mod.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-file-integrity-monitoring.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-impair-defense.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-network-service-scanning.yaml ...
created policy out/kube-system-coredns/registry-k8s-io-coredns-coredns-v1-10-1-remote-services.yaml ...
INFO[0015] pulling image                                image="gcr.io/kubebuilder/kube-rbac-proxy:v0.8.0"
v0.8.0: Pulling from kubebuilder/kube-rbac-proxy
Digest: sha256:db06cc4c084dd0253134f156ddaaaf53ef1c3fb3cc809e5d81711baaa4029ea4c
Status: Image is up to date for gcr.io/kubebuilder/kube-rbac-proxy:v0.8.0

```

Figure 13.11: karmor recommend output

From the output, you can see that karmor will pull the image for each container to test against the policies. All policies that are created by karmor are, by default, saved in a directory called out in the current working directory. You can change where the policies will be created by adding the `-o` or the `--output` switch to the `recommend` command. Since the recommendations are broken down by each action, you may generate a large number of files. To show an example, we run a `recommend` command against our KinD cluster's `kube-system` namespace, which generates the directory structure shown below:

```

./out
├── kube-system-coredns
├── kube-system-kubearmor-controller
└── kube-system-kubearmor-relay

```

Along with a directory for each deployment, you will see a `report.txt` file that contains all recommended policies for various standards, including NIST, MITRE, PCI\_DSS, CIS, etc. We will discuss the report and its options in a later section. For now, we want to focus on the created policies.

Let's take a closer look at the first directory in the list, which contains policies for the `core-dns` deployment in the `kube-system` namespace. As you can see from the output, 16 policies were created by the `recommend` command:

```

kube-system-coredns/
├── registry-k8s-io-coredns-coredns-v1-10-1-cronjob-cfg.yaml
├── registry-k8s-io-coredns-coredns-v1-10-1-file-integrity-monitoring.yaml
└── registry-k8s-io-coredns-coredns-v1-10-1-impair-defense.yaml

```

```
|-- registry-k8s-io-coredns-coredns-v1-10-1-k8s-client-tool-exec.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-maint-tools-access.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-network-service-scanning.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-pkg-mngr-exec.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-remote-file-copy.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-remote-services.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-shell-history-mod.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-system-owner-discovery.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-trusted-cert-mod.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-write-etc-dir.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-write-in-shm-dir.yaml  
|-- registry-k8s-io-coredns-coredns-v1-10-1-write-under-bin-dir.yaml  
└-- registry-k8s-io-coredns-coredns-v1-10-1-write-under-dev-dir.yaml
```

If you look at the filenames, you can tell what type of action and process each policy uses. For example, let's look at the `registry-k8s-io-coredns-coredns-v1-10-1-write-etc-dir.yaml` policy. From the filename, we can see that this policy was created to add an action to writing in the `/etc` directory. Looking into the file, we will see that this policy contains a block action on directories in `/etc` and it also locks down `/etc` to a read-only state, for anything that matches the label `k8s-app=kube-dns`:

```
apiVersion: security.kubearmor.com/v1  
kind: KubeArmorPolicy  
metadata:  
  name: coredns-registry-k8s-io-coredns-coredns-v1-10-1-write-etc-dir  
  namespace: kube-system  
spec:  
  action: Block  
  file:  
    matchDirectories:  
      - dir: /etc/  
        readOnly: true  
        recursive: true  
    message: Alert! File creation under /etc/ directory detected.  
  selector:  
    matchLabels:  
      k8s-app: kube-dns  
  severity: 5  
  tags:  
    - NIST_800-53_SI-7  
    - NIST  
    - NIST_800-53_SI-4
```

- NIST\_800-53
- MITRE\_T1562.001\_disable\_or\_modify\_tools
- MITRE\_T1036.005\_match\_legitimate\_name\_or\_location
- MITRE\_TA0003\_persistence
- MITRE
- MITRE\_T1036\_masquerading
- MITRE\_TA0005\_defense\_evasion

There are a few added fields in this policy that we haven't discussed previously, namely the severity and the tags. Unlike other tools that may add severity to a triggered event, KubeArmor allows you to set your own severity for policies. When you create a policy, you can assign it a severity rating from 1 to 10, allowing you to create your own rating based on your organizational requirements.

The tags section was generated by the recommend command. By default, when you run a recommendation, it will test all of the objects against all of the included hardening policies, including MITRE TTPs, STIGs, NIST, and CIS. The policies that are created are based on the standards supplied during the recommendation collection. If you don't specify any policies to check, karmor will create policies for all standards, including policies that you may or may not need.

Depending on your organization and security requirements, you can limit the hardening policies to only the policies that you want to include. This is done by adding either the -t or --tag flags to the recommend command, followed by the standard or standards. For example, if we wanted to run a recommend against the kube-system namespace and only include the CIS and PCI-DSS standards, we would execute:

```
karmor recommend -n kube-system -t PCI-DSS,CIS
```

Like all other recommend commands, this will create an out directory in the working directory with policies and a report.txt. If you took a look at the report, you would see a list of recommended actions around PCI-DSS and CIS standards for each pod. The figure below is an abbreviated example of a report.txt from the recommend command we ran against kube-system.

Deployment	kube-system/coredns			
Container	registry.k8s.io/coredns/coredns:v1.10.1			
OS	linux			
Arch	amd64			
Distro				
Output Directory	out/kube-system-coredns			
policy-template version	v0.2.3			
POLICY	SHORT DESC	SEVERITY	ACTION	TAGS
registry-k8s-io-coredns-v1-10-1-cronjob-cfg.yaml	System and Information Integrity - System Monitoring Detect access to cronjob files	5	Audit	NIST SI-4 NIST 800-53_SI-4 CIS CIS Linux CIS_5.1_Configure_Cron

Figure 13.12: Recommend example for NIST and CIS

Since we added tags to our command, karmor only created policies that are required to meet the standards for NIST and CIS. This will create fewer policies than running without any tags since it will only generate policies based on the specified tags versus all standards if you do not supply a tag.

The last example we will discuss is using recommend to create policies and a report for an image that is not running in the cluster. So far, we have run recommendations against objects in the cluster, but KubeArmor offers the ability to create policies based on any container image. To run a recommendation against an image, you need to add the `-i` or `--image` to your command. For example, we want to run karmor against the `bitnami/nginx` image:

```
karmor recommend -i bitnami/nginx
```

This will pull down the image and run it against all of the included KubeArmor policies. The policies will be created in the `out` directory, just like the previous examples:

```
out
└── bitnami-nginx-latest
    Notice that the directory name does not contain a namespace; it only has the
    image name and tag that we tested against, bitnami/nginx:latest. This example
    run created a number of policies since we ran it against all included policies:
    ├── access-ctrl-permission-mod.yaml
    ├── cis-commandline-warning-banner.yaml
    ├── cronjob-cfg.yaml
    ├── file-integrity-monitoring.yaml
    ├── file-system-mounts.yaml
    ├── impair-defense.yaml
    ├── k8s-client-tool-exec.yaml
    ├── maint-tools-access.yaml
    ├── network-service-scanning.yaml
    ├── pkg-mngr-exec.yaml
    ├── remote-file-copy.yaml
    ├── remote-services.yaml
    ├── shell-history-mod.yaml
    ├── system-network-env-mod.yaml
    ├── system-owner-discovery.yaml
    ├── trusted-cert-mod.yaml
    ├── write-etc-dir.yaml
    ├── write-in-shm-dir.yaml
    ├── write-under-bin-dir.yaml
    └── write-under-dev-dir.yaml
```

If we ran the same test and included the tag for only the CIS policies, we would generate fewer policies:

```
├── access-ctrl-permission-mod.yaml
├── cis-commandline-warning-banner.yaml
└── cronjob-cfg.yaml
```

```
|── file-system-mounts.yaml  
└── system-network-env-mod.yaml
```

As demonstrated, the recommend command empowers you to increase the security of your workloads in accordance with any standards required by your organization, government regulations, or other pertinent criteria.

## karmor logs

The logs option provides a real-time log of KubeArmor's activities, which is beneficial when you want to watch events without seeing hundreds of other logged activities. When you execute a karmor log, a logger will start up and watch for KubeArmor activity. Since it's a real-time log, it will run interactively in your shell, waiting for activity:

```
local port to be used for port forwarding kubearmor-relay-7676f9684f-65211:  
32879  
Created a gRPC client (localhost:32879)  
Checked the liveness of the gRPC server  
Started to watch alerts
```

As events are observed by KubeArmor, they will be shown in the output. For example, we create a policy that will block any write attempts to the /bin directory in all containers in the demo namespace. We exec into the container and attempt to create a file called test in the directory. As we can see in the output below, the attempt was denied:

```
I have no name!@nginx-web-57794669f5-gd4r4:/app$ cd /bin  
I have no name!@nginx-web-57794669f5-gd4r4:/bin$ touch test  
touch: cannot touch 'test': Permission denied
```

Since this was an action that KubeArmor had a policy for, it will also log the activity in the session running the karmor logs. The logs contain a lot of information. Below is an example of a logged event:

```
-- Alert / 2023-10-09 14:14:32.192243 --  
ClusterName: default  
HostName: cluster01-worker  
NamespaceName: demo  
PodName: nginx-web-57794669f5-gd4r4  
Labels: app=nginx-web  
ContainerName: nginx  
ContainerID: 8048c5fb3fd2425e401505cb4c12d147fddd71e6587ba3f3c488e609b28819a8  
ContainerImage: docker.io/bitnami/nginx:latest@  
sha256:4ce786ce4a547b796cf23efef62b54a910de6fd41245012f10e5f75e85ed3563c  
Type: MatchedPolicy  
PolicyName: DefaultPosture
```

```

Source: /usr/bin/touch test
Resource: test
Operation: File
Action: Block
Data: syscall=SYS_OPENAT fd=-100 flags=0_WRONLY|0_CREAT|0_NOCTTY|0_NONBLOCK
Enforcer: AppArmor
Result: Permission denied
HostPID: 2.731789e+06
HostPPID: 2.731618e+06
Owner: map[Name:nginx-web Namespace:demo Ref:Deployment]
PID: 59
PPID: 53
ParentProcessName: /bin/bash
ProcessName: /bin/touch
UID: 1001

```

By default, the log output is set to text, which may be difficult to sort through when there are a lot of logged events. If you would prefer the logs to be in JSON format, you can add the flag `--json` to the `logs` command. The format that is best to use for your requirements usually depends on the system you are using for storing your logged events. In most cases, JSON is the format preferred by most logging systems.

To show the difference, we execute the same test from the previous log entry, an attempt to create a file under `/bin`. This will change the output from text to JSON, as shown below:

```
{
  "Timestamp": 1696861352,
  "UpdatedTime": "2023-10-09T14:22:32.445655Z",
  "ClusterName": "default",
  "HostName": "cluster01-worker",
  "NamespaceName": "demo",
  "Owner": {
    "Ref": "Deployment",
    "Name": "nginx-web",
    "Namespace": "demo"
  },
  "PodName": "nginx-web-57794669f5-gd4r4",
  "Labels": "app=nginx-web",
  "ContainerID": "8048c5fb3fd2425e401505cb4c12d147fddd71e6587ba3f3c488e609b28819a8",
  "ContainerName": "nginx",
  "ContainerImage": "docker.io/bitnami/nginx:latest@sha256:4ce786ce4a547b796cf23efe62b54a910de6fd41245012f10e5f75e85ed3563c",
  "HostPPID": 2731618,
  "HostPID": 2739719,
  "PPID": 53,
  "PID": 60,
  "UID": 1001,
  "ProcessName": "/bin/touch",
  "PolicyName": "DefaultPosture",
  "Type": "MatchedPolicy",
  "Source": "/usr/bin/touch test",
  "Operation": "File",
  "Resource": "test",
  "Data": "syscall=SYS_OPENAT fd=-100 flags=0_WRONLY|0_CREAT|0_NOCTTY|0_NONBLOCK",
  "Enforcer": "AppArmor",
  "Action": "Block",
  "Result": "Permission denied"
}
```

The decision to use text or JSON is usually dependent on the tools you plan to use to parse the data. JSON is a popular format for logging since it makes parsing the data much easier than using a text format.

You may be looking at the logging abilities using the karmor log and wondering how useful it is to have the output going to the console only.

As a default behavior, the logs are directed to `stdout`. While real-time log viewing is valuable for monitoring events as they happen, it's not always feasible to continuously observe events. It is more common to send your logs to a file, which can then be viewed or sent to an external system for processing. KubeArmor offers the flexibility to modify your log output preferences by using the `--logPath` flag. This flag lets you specify the desired destination for the log file, enabling you to redirect log data to a designated file location.

When you specify the path of the log file, it must include the entire path and filename that you want to use. The `logPath` option can be used with other options, like setting the log format to JSON. The example command below will send the logs to the current user's home directory using the filename `karmor.logs` in JSON format:

```
karmor logs --json --logPath ~/karmor.logs
```

If you specify a new directory in `logPath`, it must be created before sending any logs; KubeArmor will not create the directory for you. If you fail to create the directory before logging, you will receive an error on the log output screen and no events will be logged.

When the first event is caught by KubeArmor, it will create the file and will continue to collect data until you stop logging:

```
-rw-rw-r-- 1 surovich surovich 2602 Oct 9 14:49 karmor.logs
```

Like other `karmor` options, logging will watch the entire cluster by default, which may generate a lot of events that may make it difficult to find the events you really need to see. Instead of logging the entire cluster, you can filter the logs by adding a flag to the `logs` option. The available flags include:

Flag	Description
<code>--labels</code>	Filter the logs by a label
<code>--namespace</code>	Filter the logs by namespace
<code>--pod</code>	Filter the logs by pod

*Table 13.3: Limiting logs to certain objects*

Limiting the scope of the objects being logged offers the distinct advantage of tailoring the log data to focus exclusively on the specific object(s) you require detailed information for. By implementing this approach, you can significantly enhance the precision and relevance of the logged data, ensuring that it directly aligns with your specific monitoring, analysis, and troubleshooting needs. Limiting the objects being logged enables you to streamline your monitoring efforts, making it more efficient and effective in providing insights into the targeted object(s) while reducing the noise and clutter caused by unnecessary log entries.

## karmor vm

Did you know that with Kubernetes you can run virtual machines? These VMs are deployed and managed differently than what you may be used to when working with hypervisors from VMWare and Microsoft. Instead of running an OS straight on the hypervisor, KubeVirt VMs actually run inside of a container. They look like a standard pod running any other Docker image, but instead of a microservice, it's an entire operating system supporting both Windows and Linux.

KubeVirt is a complex topic and we can't cover it in this section alone. You can learn more about KubeVirt on their website, <https://kubevirt.io/>.

The KubeArmor team saw the need to expand runtime security to include support for VMs running with KubeVirt. This is a powerful feature for organizations that run VMs in Kubernetes, extending the same security that KubeArmor provides for containers to VMs.

## Summary

In this chapter, we looked at strengthening the security of our runtime environment, enhancing your overall security posture. It's a common misconception that an organization's clusters are secured since many of them tend to overlook the content running within containers or the implications of a user connecting to a running pod using `kubectl exec`.

This chapter also described in detail how one of the most effective approaches to container security involves tightly controlling the container's processes, exclusively allowing the execution of only the necessary processes while denying access to all other files. By leveraging a tool like KubeArmor, you can grant access to specific files from a restricted set of binaries, blocking access to and securing all other processes.

## Questions

1. Which of the following are LSMs?
  - a. Accuknox
  - b. AppArmor
  - c. SELinux
  - d. LSMLinux
2. LSMs and eBPF provide the same features.
  - a. True
  - b. False
3. Which karmor option provides real-time information in an easy-to-see console?
  - a. Monitor
  - b. Trace
  - c. Profile
  - d. Probe

4. Which of the following is NOT a feature provided by KubeArmor to enhance security in Kubernetes clusters?
  - a. Restricting process execution
  - b. File access control
  - c. Network traffic encryption
  - d. Creating security policies

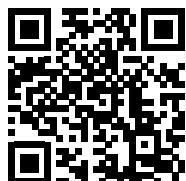
## Answers

1. b - AppArmor
2. b - False
3. c - Profile
4. c - Network traffic encryption

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>





# 14

## Backing Up Workloads

Backup products for Kubernetes are vital components of our ever-evolving journey into the world of container orchestration and cloud-native computing. In this chapter, we will explore using Velero's capabilities and how it can help you ensure resilience and reliability for your workloads. Velero, meaning "safety" or "protection" in Italian, is a great name since it provides a safety net for your applications, allowing you to confidently run them in a dynamic, ever-changing environment.

As you dive deeper into Kubernetes and microservices, you will quickly realize the advantages of backing up, restoring, and migrating applications. While Kubernetes is a remarkable system for deploying and managing containerized applications, it doesn't inherently provide tools for data protection and disaster recovery. This gap is filled by **Velero**, which presents a complete solution for safeguarding your Kubernetes workloads and the data connected with them.

Velero was originally known as Heptio Ark. Heptio was a company co-founded by two of Kubernetes' original creators, Joe Beda and Craig McLuckie. Since then, it has become part of the VMware Tanzu portfolio, demonstrating its importance to the Kubernetes ecosystem.

In this chapter, we will explore the key features and use cases of Kubernetes and Velero, from basic backup and restore operations to more advanced scenarios like cross-cluster migrations. Whether you are just beginning your Kubernetes journey or are a seasoned Kubernetes operator, Velero is a tool worth learning.

In this chapter, we will cover the following topics:

- Understanding Kubernetes backups
- Performing an etcd backup
- Introducing and setting up VMware's Velero
- Using Velero to back up workloads and PVCs
- Managing Velero using the CLI
- Restoring from a backup

## Technical requirements

To carry out the hands-on experiments in this chapter, you will need the following:

- An Ubuntu 22.04+ server running Docker with a minimum of 8 GB of RAM.
- A KinD cluster built to the specifications in *Chapter 2*.
- Scripts from the `chapter14` folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Understanding Kubernetes backups

Backing up a Kubernetes cluster requires backing up not only the workloads running on the cluster. You need to consider any persistent data and the cluster itself. Remember that the cluster state is maintained in an etcd database, making it a very important component that you need to back up in order to recover from any disasters.

Creating a backup of the cluster and the running workloads allows you to do the following:

- Migrate clusters.
- Create a development cluster from a production cluster.
- Recover a cluster from a disaster.
- Recover data from persistent volumes.
- Namespace and deployment recovery.

In this chapter, we will provide the details and tools to back up your etcd database, your namespace, the objects in them, and any persistent data you have attached to your workloads.

Recovering a cluster from a complete disaster in an enterprise usually involves backing up custom SSL certificates for various components, such as Ingress controllers, load balancers, and the API server. Since the process of backing up all custom components is different in many environments, we will focus on the procedures that are common among most Kubernetes distributions.

As you know, the cluster state is maintained in etcd, and if you lose all of your etcd instances, you will lose your cluster. In a multi-node control plane, you should have a minimum of three etcd instances, providing redundancy for the cluster. If you lose a single node, the cluster will remain running, and you can replace the failed node with a new node. Once the new instance has been added, it will receive a copy of the etcd database and your cluster will be back to full redundancy.

In the event that you lose all of your etcd servers without any backup of the database, you would lose the cluster, including the cluster state and all of the workloads. Since etcd is so important, the `etcdctl` utility includes a built-in backup function. In the next section, we will show you how to take an etcd backup using the `etcdctl` utility.

## Performing an etcd backup

Since we are using KinD for our Kubernetes cluster, we can create a backup of the etcd database, but we will not be able to restore it.

Our etcd server is running in a pod on the cluster called `etcd-cluster01-control-plane`, located in the `kube-system` namespace. During the creation of the KinD cluster, we added an extra port mapping for the control plane node, exposing port 2379, which is used to access etcd. In your own production environment, you may not have the etcd port exposed for external requests, but the process of backing up the database will still be similar to the steps explained in this section.

## Backing up the required certificates

Most Kubernetes installations store certificates in `/etc/kubernetes/pki`. In this respect, KinD is no different, so we can back up our certificates using the `docker cp` command.

We have said it a few times: etcd is very important! So, it stands to reason that accessing the database directly probably has some security around it. Well, it does, and to access it, you need to provide the correct certificates when you execute a command against the database. In an enterprise, you should store these keys in a secure location. For our example, we will pull the certificates from the KinD nodes.

We have included a script in the `chapter14/etcd` directory called `install-etcd-tools.sh` that will execute the steps to download and execute the backup of the etcd database. To execute the script, change to the `chapter14/etcd` directory and execute the installation script.

Running the script will download the etcd tools, extract them, and move them to `usr/bin` so we can execute them easily. It will then create a directory for the certificates and copy them into the newly created directory, `/etcd/certs`. The certificates that we will use for backing up etcd are:

- `ca.crt`
- `healthcheck-client.crt`
- `healthcheck-client.key`

When you execute commands using the `etcdctl` utility, you will need to provide the keys or your actions will be denied.

Now that we have the certificates required to access etcd, the next step is to create a backup of the database.

## Backing up the etcd database

The creators of etcd created a utility that backs up and restores the etcd database, called `etcdctl`. For our purposes, we will only use the backup operation; however, since etcd is not exclusive to Kubernetes, the utility has a number of options that you will not use as a Kubernetes operator or developer. If you want to read more about this utility, you can visit the `etcd-io` Git repository at <https://github.com/etcd-io/etcd>.

To back up a database, you will need the `etcdctl` utility and the certificates required to access the database, which we copied from the control plane server.

The script that we executed in the last section downloaded `etcdctl`, and we moved it into `usr/bin`. To create a backup of the database, make sure you are in the `chapter14/etcd` directory and that the `certs` directory exists with the downloaded certificates.

To back up etcd, we execute the `etcdctl snapshot save` command:

```
etcdctl snapshot save etcd-snapshot.db --endpoints=https://127.0.0.1:2379  
--cacert=./certs/ca.crt --cert=./certs/healthcheck-client.crt --key=./certs/  
healthcheck-client.key
```



Older versions of `etcdctl` required you to set the API version to 3 using `ETCDCTL_API=3` since they default to the version 2 API. `etcd 3.4` changed the default API to 3, so we do not need to set that variable before using `etcdctl` commands.

It shouldn't take too long for the database to be copied over. If it takes more than a few seconds, you should try the same command with the `--debug=true` flag. Adding the debug flag will provide more output during the execution of the snapshot. The most common reason for the snapshot failing is an incorrect certificate. Below is an example of the verbose output from a snapshot command that had an incorrect certificate:

```
2023/11/17 21:39:19 INFO: [core] Creating new client transport to "{Addr:  
\\"127.0.0.1:2379\\", ServerName: \\"127.0.0.1:2379\\", }": connection error:  
desc = "transport: authentication handshake failed: tls: failed to verify  
certificate: x509: certificate signed by unknown authority (possibly because of  
\\"crypto/rsa: verification error\\" while trying to verify candidate authority  
certificate \\"etcd-ca\\")"
```

Notice the `x509` error. This is likely caused by an incorrect certificate in your `etcdctl` command. Check that you have the correct certificates, and re-run the command.

If the command is successful, you will receive output similar to this:

```
{"level":"info","ts":"2023-11-17T21:44:38.316265Z","caller":"snapshot/  
v3_snapshot.go:65","msg":"created temporary db file","path":"etcd-snapshot.  
db.part"}  
{"level":"info","ts":"2023-11-17T21:44:38.329699Z","logger":"client","caller":  
"v3@v3.5.10/maintenance.go:212","msg":"opened snapshot stream; downloading"}  
{"level":"info","ts":"2023-11-17T21:44:38.329756Z","caller":"snapshot/v3_  
snapshot.go:73","msg":"fetching snapshot","endpoint":"https://127.0.0.1:2379"}  
{"level":"info","ts":"2023-11-17T21:44:38.45673Z","logger":"client","caller":  
"v3@v3.5.10/maintenance.go:220","msg":"completed snapshot read; closing"}  
{"level":"info","ts":"2023-11-17T21:44:38.461743Z","  
caller":"snapshot/v3_snapshot.go:88","msg":"fetched  
snapshot","endpoint":"https://127.0.0.1:2379","size":"6.6 MB","took":"now"}  
{"level":"info","ts":"2023-11-17T21:44:38.46276Z","caller":"snapshot/v3_  
snapshot.go:97","msg":"saved","path":"etcd-snapshot.db"}  
Snapshot saved at etcd-snapshot.db
```

Next, we can verify that the database was copied over successfully by trying a simple `etcdctl` command that will provide a summary of the backup:

```
etcdctl --write-out=table snapshot status etcd-snapshot.db
```

This will output an overview of the backup:

HASH	REVISION	TOTAL KEYS	TOTAL SIZE
224e9348	6222	1560	6.6 MB

For this example, we only backed up the etcd database once. In a real-life scenario, you should create a scheduled process that executes a snapshot of etcd at regular intervals and stores the backup file in a safe, secure location.

Due to how KinD runs the control plane, we cannot use the restore procedures in this section. We are providing only the backup steps in this section so that you know how to back up an etcd database in an enterprise environment.

So far, you have learned about the critical importance of backing up both workloads and persistent data in Kubernetes, including the etcd database. Having a good backup strategy allows you to facilitate cluster migrations, create new development clusters from production clusters, and recover from disasters. By knowing these strategies, you can ensure improved disaster recovery preparedness, enhanced operational efficiency, and data security. Mastering these techniques will equip you to manage and recover Kubernetes clusters more effectively, ensuring a resilient and reliable environment.

Now, let's move on and introduce the tool we will use to demonstrate Kubernetes backups: Velero.

## Introducing and setting up VMware's Velero

Velero is an open-source backup solution for Kubernetes that was originally developed by a company called Heptio. As VMware has enhanced its support for Kubernetes, it has purchased multiple companies, and Heptio is one of its acquisitions, bringing Velero into the VMware portfolio.

VMware has moved most of its offerings around Kubernetes under the Tanzu umbrella. This can be a little confusing for some people since the original iteration of Tanzu was a deployment of multiple components that added Kubernetes support to vSphere clusters. Since the initial incarnation of Tanzu, it has come to include components such as Velero, Harbor, and the **Tanzu Application Platform (TAP)**, none of which require vSphere to function; they will run natively in any standard Kubernetes cluster.

Even with all of the ownership and branding changes, the base functions of Velero have remained. It offers many features that are only available in commercial products, including scheduling, backup hooks, and granular backup controls – all for no charge.

While Velero is free, it has a learning curve since it does not include an easy-to-use GUI like most commercial products. All operations in Velero are carried out using their command-line utility, an executable called `velero`. This single executable allows you to install the Velero server, create backups, check the status of backups, restore backups, and more. Since every operation for management can be done with one file, restoring a cluster's workloads is a very easy process. In this chapter, we will create a second KinD cluster and populate it with a backup from an existing cluster.

But before that, we need to take care of a few requirements.

## Velero requirements

Velero consists of a few components with which you create a backup system:

- **The Velero CLI:** This provides the installation of Velero components. It is used for all backup and restore functions.
- **The Velero server:** This is responsible for executing backup and restore procedures.
- **Storage provider plug-ins:** These are used for backing up and restoring to specific storage systems.

Outside of the base Velero components, you will also need to provide an object storage location that will be used to store your backups. If you do not have an object storage solution, you can deploy MinIO, which is an open-source project that provides an S3-compatible object store. We will deploy MinIO in our KinD cluster to demonstrate the backup and restore features provided by Velero.

## Installing the Velero CLI

The first step of deploying Velero is to download the latest Velero CLI binary. We have included a script to install the Velero binary in the `chapter14` directory called `install-velero-binary.sh`, which will download the Velero binary, move it to `/usr/bin`, and then output the version of Velero to verify that the binary has been installed correctly. As of the writing of this chapter, the latest version of Velero is 1.12.1.

You can safely ignore the last line, which shows an error in finding the Velero server. Right now, all we have installed is the Velero executable, and it can't find the server yet. In the next section, we will install the server to complete the installation.

## Installing Velero

Velero has minimal system requirements, most of which are easily met:

- A Kubernetes cluster running version 1.16 or higher
- The Velero executable
- Images for the system components
- A compatible storage location
- A volume snapshot plugin (optional)

Depending on your infrastructure, you may not have a compatible location for the backups or snapshotting volumes. Fortunately, if you do not have a compatible storage system, there are open-source options that you can add to your cluster to meet the requirements.

In the next section, we will explain the natively supported storage options and since our example will use a KinD cluster, we will install open-source options to add compatible storage to use as a backup location.

## Backup storage location

Velero requires an S3-compatible bucket to store backups. There are a number of officially supported systems, including all object store offerings from AWS, Azure, and Google.

In the following table, the **Support** column means that the plugin provides a compatible location for storing Velero backups. The **Volume Snapshot Support** column means that the plugin supports backing up persistent volumes using snapshots. If the CSI in use does not provide snapshot support, data will be backed up using a standard file system backup via Restic or Kopia. Snapshots offer several advantages, with the most significant being their ability to maintain application consistency. Velero ensures that snapshots are captured in a manner that maintains the state of the application, minimizing the likelihood of data corruption.

Along with the officially supported providers, there are a number of community- and vendor-supported providers from companies such as DigitalOcean, Hewlett-Packard, and Portworx. The following table lists all of the current providers:

Vendor	Object Store	Volume Snapshot Support	Support
Amazon	AWS S3	AWS EBS	Official
Google	Google Cloud Storage	GCE Disks	Official
Microsoft	Azure Blob Storage	Azure Managed Disks	Official
VMware	Not Supported	vSphere Volumes	Official
Kubernetes CSI	Not Supported	CSI Volumes	Official
Alibaba Cloud	Alibaba Cloud OSS	Alibaba Cloud	Community
DigitalOcean	DigitalOcean Object Storage	DigitalOcean Volumes Block Storage	Community
HP	Not Supported	HPE Storage	Community
OpenEBS	Not Supported	OpenEBS cStor Volumes	Community
Portworx	Not Supported	Portworx Volumes	Community
Storj	Storj Object Storage	Not Supported	Community

Table 14.1: Velero storage options

If you do not have an object storage solution, you can deploy the open-source S3 provider MinIO, which is what we will use for our S3 target in this chapter.

Now that the Velero executable has been installed and our KinD cluster has persistent storage, thanks to the auto-provisioner from Rancher, we can move on to the first requirement – adding an S3-compatible backup location for Velero.

## Deploying MinIO

MinIO is an open-source object storage solution that is compatible with Amazon’s S3 cloud services API. You can read more about MinIO in its GitHub repository at <https://github.com/minio/minio>.

If you install MinIO using a manifest from the internet, be sure to verify what volumes are declared in the deployment before trying to use it as a backup location. Many of the examples on the internet use `emptyDir: {}`, which is not persistent.

We have included a modified MinIO deployment from the Velero GitHub repository in the `chapter14` folder. Since we have persistent storage on our cluster, we edited the volumes in the deployment to use **Persistent Volume Claims (PVCs)**, which will use the auto-provisioner for Velero’s data and configuration.

To deploy the MinIO server, change directories to `chapter14` and execute `kubectl create`. The deployment will create a Velero namespace, PVCs, and MinIO on your KinD cluster. It may take some time for the deployment to complete. We have seen the deployment take anything from a minute to a few minutes, depending on the host system:

```
kubectl create -f minio-deployment.yaml
```

This will deploy the MinIO server and expose it as `minio` on port 9000/TCP, with the console on port 9001/TCP, as follows:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
console	ClusterIP	10.102.216.91	<none>	9001/TCP	42h
minio	ClusterIP	10.110.216.37	<none>	9000/TCP	42h

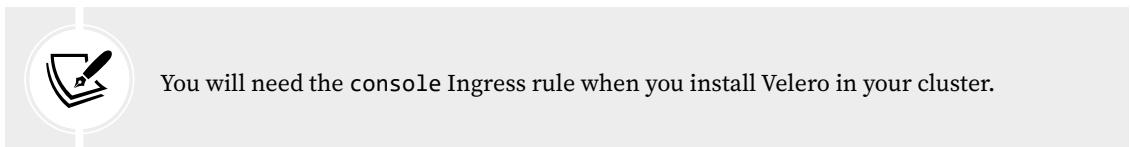
The MinIO server can be targeted by any pod in the cluster, with correct access keys, using `minio.velero.svc` on port 9000.

With MinIO deployed, we need to expose the console using an Ingress rule so we can log in to look at buckets and verify that backups are working as expected.

## Exposing MinIO and the console

By default, your MinIO storage will only be available inside the cluster it has been deployed in. Since we will demonstrate restoring to a different cluster at the end of the chapter, we need to expose MinIO using an Ingress rule. MinIO also includes a dashboard that allows you to browse the contents of the S3 buckets on the server. To allow access to the dashboard, you can deploy an Ingress rule that exposes the MinIO console.

We have included a script in the `chapter14` folder called `create-minio-ingress.sh` that will create an Ingress rule using the `nip.io` syntax of `minio-console.w.x.y.z.nip.ip` and `minio.w.x.y.z.nip.ip`, with your host IP.



Once deployed, you can use a browser on any machine and open the URL you used for the Ingress rule. On our cluster, the host IP is 10.2.1.161, so our URL is `minio-console.10.2.1.161.nip.io`:

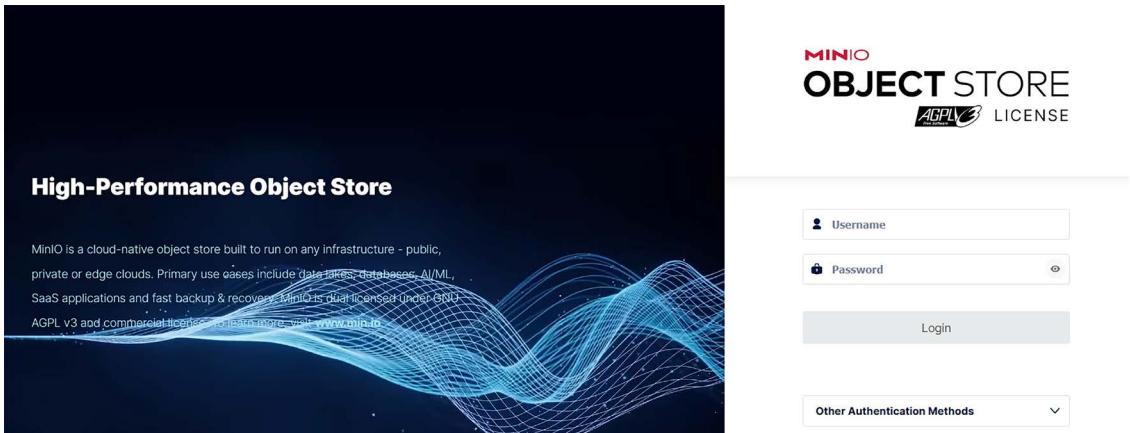


Figure 14.1: MinIO dashboard

To access the dashboard, supply the access key and secret key from the MinIO deployment. If you used the MinIO installer from the GitHub repository, the username and password have been defined in the manifest. They are `packt/packt123`.

Once logged in, you will see a list of buckets and any items that are stored in them. You should see a bucket named `velero`, which is the bucket we will use to back up our cluster. This bucket was created during the initial MinIO deployment – we added a line to the deployment that creates the `velero` bucket and the required permissions for the `packt` user.

A screenshot of the MinIO browser. The sidebar menu includes 'User', 'Object Browser' (which is selected), 'Access Keys', and 'Documentation'. The main area is titled 'Object Browser' and contains a table with a single row. The table has columns for 'Name', 'Objects', and 'Size'. The row shows 'velero' in the 'Name' column, '0' in the 'Objects' column, and '0.0 B' in the 'Size' column. There is also a 'Filter Buckets' input field at the top of the table.

Figure 14.2: MinIO browser

If you are new to object storage, it is important to note that while this deploys a storage solution in your cluster, it **will not** create a `StorageClass` or integrate with Kubernetes in any way. All pod access to the S3 bucket is done using the URL that we will provide in the next section.

Now that you have an S3-compatible object store running, you need to create a configuration file that Velero will use to target your MinIO server.

## Installing Velero

To deploy Velero in your cluster, you can use the Velero binary or a Helm chart. We have chosen to install Velero using the binary.

Before we start the installation, we need to create a credentials file that will contain the `access_key` and `secret_access_key` for the S3 target on MinIO.

Create a new credential file in the `chapter14` folder called `credentials-velero` with the following content:

```
[default]aws_access_key_id = packt
aws_secret_access_key = packt123
```

Next, we can deploy Velero using the Velero executable and the `install` option to deploy Velero with the option to back up persistent volumes.

Execute the Velero installation using the following command from inside the `chapter14` folder to deploy Velero. Please note that you need to provide your `nip.io` ingress name for MinIO. We exposed both MinIO and the console when we created the Ingress rule earlier. Be careful to use the ingress name that contains `minio.w.x.y.z.nip.io`; do not use the `minio-console` ingress or Velero will fail to find the S3 bucket.

```
velero install --provider aws --plugins velero/velero-plugin-for-aws:v1.2.0 --bucket velero --secret-file ./credentials-velero --use-volume-snapshots=false --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=http://minio.velero.svc:9000 --use-node-agent --default-volumes-to-fs-backup
```

Let's explain the installation options and what the values mean:

Option	Description
<code>--provider</code>	Configures Velero to use a storage provider. Since we are using MinIO, which is S3-compatible, we are passing <code>aws</code> as our provider.
<code>--plugins</code>	Tells Velero the backup plugin to use. For our cluster, since we are using MinIO for object storage, we selected the AWS plugin.
<code>--bucket</code>	The name of the S3 bucket that you want to target.
<code>--secret-file</code>	Points to the file that contains the credentials to authenticate with the S3 bucket.
<code>--use-volume-snapshots</code>	Will enable or disable volume snapshots for providers that support snapshots. Currently, Velero only supports object storage with snapshots; if snapshots are not supported, this should be set to <code>false</code> . Since we are not interested in snapshots for our examples, we set this to <code>false</code> .

--backup-location-config	The S3 target location where Velero will store backups. Since MinIO is running in the same cluster as Velero, we can target S3 using the name <code>minio.velero.svc:9000</code> . In a production environment, you would use MinIO in the same cluster – you will likely have an external S3 target to store your backups. Using the Kubernetes service name will cause Velero <code>describe</code> commands to have some errors since it tries to query the cluster using the name provided, and you cannot access <code>minio.velero.svc</code> from outside of the cluster.
--use-node-agent	Add this flag if you want to back up persistent volumes using Velero's node agent.
--default-volumes-to-fs-backup	Configures Velero to support opting out of backing up persistent volumes. If this isn't added during deployment, you can still use the option during a Velero backup to back up volumes. This will be explained more in the <i>Backing up PVCs</i> section of this chapter.

Table 14.2: Velero install options

When you execute the install, you will see a number of objects being created, including a number of **CustomResourceDefinitions (CRDs)** and other objects that Velero uses to handle backup and restore operations.

If you run into issues with your Velero server starting up correctly, there are a few CRDs and Secrets that you can look at that may have incorrect information. In the following table, we explain some of the common objects that you may need to interact with when using Velero:

CustomResourceDefinition	Name	Description
<code>backups.velero.io</code>	<code>Backup</code>	Each backup that is created will create an object called <code>backup</code> , which includes the settings for each backup job.
<code>backupstoragelocations.velero.io</code>	<code>BackupStorageLocation</code>	Each backup storage location creates a <code>BackupStorageLocation</code> object that contains the configuration to connect to the storage provider.
<code>schedules.velero.io</code>	<code>Schedule</code>	Each scheduled backup creates a <code>Schedule</code> object that contains the schedule for a backup.
<code>volumesnapshotlocations.velero.io</code>	<code>VolumeSnapshotLocation</code>	If enabled, the <code>VolumeSnapshotLocation</code> object contains the information for the storage used for volume snapshots.

Secret Name	Description
cloud-credentials	Contains the credentials to connect to the storage provider in Base64 format. If your Velero pod fails to start up, you may have an incorrect value in the <code>data.cloud</code> spec.
velero-repo-credentials	If you are using the Restic plugin, this will contain your repository password, similar to <code>cloud-credentials</code> . If you experience issues connecting to the volume snapshot provider, verify that the repository password is correct.

Table 14.3: Velero's CRDs and Secrets

While most of your interaction with these objects will be through the Velero executable, it is always a good practice to understand how utilities interact with the API server. Understanding the objects and what their functions are is helpful if you do not have access to the Velero executable but you need to view, or potentially change, an object value to address an issue quickly.

Now that we have Velero installed and a high-level understanding of Velero objects, we can move on to creating different backup jobs for a cluster.

## Using Velero to back up workloads and PVCs

Velero supports running a one-time backup with a single command or on a recurring schedule. Whether you choose to run a single backup or a recurring backup, you can back up all objects or only certain objects using `include` and `exclude` flags.

### Backing up PVCs

Since data is becoming increasingly common on Kubernetes clusters, we will back up all of the cluster workloads, including any PVCs that are in the cluster. When we installed Velero, we added the `--use-node-agent` option, which created a `DaemonSet` that creates a node agent on each cluster node. The `DaemonSet` deploys a pod containing modules that can perform file system backups, including a data mover, which may be `Restic` or `Kopia` (*default*) on each node, and a new secret is created in the `velero` namespace called `velero-repo-credentials`. This secret contains a `repository-password` that will be used for your backups. This is a generated password, and you can change it to anything you want – however, if you plan to change the password, do it before creating any backups. If this password is changed after you have created any backups, Velero will not be able to read the old backups.

The default `ServiceAccount` token, `Secrets`, and `ConfigMaps` can be mapped to volumes. These are not volumes that contain data and will not be backed up using the node agent. Like any other base Kubernetes objects, they will be backed up when Velero backs up the other namespace objects.



The data movers are responsible for copying the data from the volumes. Previous Velero releases used Restic as the data mover, but it has been enhanced to include both Restic and Kopia in the node DaemonSet. By default, Kopia will be used as the data mover, but if you want to use Restic, you can change the default by adding the option `--data-mover restic` to your Velero backup create command. There is some debate around which data mover to use and Kopia has become the leader, so it has become the default.

Velero can be configured to back up PVCs via two different approaches:

- **Opt-out:** Velero will back up all PVCs, unless a workload is annotated with the volume name(s) to ignore.
- **Opt-in:** Only workloads that have an annotation with the volume's name will be backed up. This is Velero's default configuration.

Let's take a closer look at these two approaches.

## Using the opt-out approach

This is the approach we will use for the exercises. When using this approach, all PVCs will be backed up unless you specify an annotation in the pod, adding `backup.velero.io/backup-volumes-excludes`. For example, if you had 3 PVCs named `volume1`, `volume2`, and `volume3` in a namespace and you wanted to exclude `volume2` and `volume3` from being backed up, you would need to add the following annotation to the pod spec in your deployment:

```
backup.velero.io/backup-volumes-excludes=volume2,volume3
```

Since we added only `volume2` and `volume3` to the exclusion, Velero will ignore those volumes from the backup, but it will back up the PVC named `volume1` since it was not included in the list.

During the installation of Velero, we set `--default-volumes-to-fs-backup`, which tells Velero to back up all persistent data, unless a volume has an annotation to exclude it from being backed up. If you didn't set that option during your Velero deployment, you can tell Velero to use the opt-out approach for a single backup by adding the same option, `--default-volumes-to-fs-backup`, to the `backup` command.

```
velero backup create BACKUP_NAME --default-volumes-to-fs-backup OTHER_OPTIONS
```

When a backup is created using this option, Velero will back up every persistent volume that is attached to pods, unless it has been excluded in the `excludes` annotation.

## Using the opt-in approach

If you deployed Velero without the `--default-volumes-to-fs-backup` option, persistent volumes will not be backed up unless you add an annotation to tell Velero to back up the required volumes.

Similarly to how you opted out in the previous example, you can add an annotation to your deployment to instruct Velero to back up your volume or volumes. The annotation that you need to add is `backup.velero.io/backup-volumes`, and the following example tells Velero to back up two volumes, one called `volume1` and the other called `volume2`:

```
kubectl -n demo annotate deploy/demo backup.velero.io/backup-volumes=volume1,volume2
```

When you run your next backup, Velero will see the annotation and will add the two persistent volumes to the backup job.

## Limitations of backing up data

Velero cannot back up a volume that is using hostPath for the persistent data. The local-path-provisioner maps persistent disks to a hostPath by default, meaning that Velero will not be able to back up or restore the data. Luckily, there is an option to change the type from hostPath to local, which will work with Velero. When you create a new PVC, you can add an annotation of volumeType: local. The example below shows a PVC manifest that would be created as a local type, rather than hostPath:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: test-claim
  namespace: demo
  annotations:
    volumeType: local
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

This change is not needed in many cases, but since it's required when using the local-path-provisioner, we will need to add the annotation to any PVCs that we want to test with Velero.

With Velero deployed with the ability to back up our persistent data, let's jump into creating a one-time backup of our cluster.

## Running a one-time cluster backup

To create an initial backup, you can run a single Velero command that will back up all of the namespaces in the cluster and if there are any PVCs that have not been annotated to be ignored, they will also be backed up.

Executing a backup without any flags to include or exclude any cluster objects will back up every namespace and all of the objects in the namespace.

We will use what we learn in this section to perform a restore to show Velero in action. For our backup, we will back up the entire cluster, including the PVCs.

Before we start a backup, we are going to add a deployment with a PVC, where we will add a few empty files to verify that restoring data works as expected.

In the `chapter14/pvc-example` directory, there is a manifest called `busybox-pvc.yaml`. To deploy the example, you should execute the command from within the `chapter14/pvc-example` directory:

```
kubectl create -f busybox-pvc.yaml
```

The script will create a new namespace called `demo` with the `busybox-pvc` pod deployed using a PVC named `test-claim`, mounted in the `/mnt` directory of the container.

Right now, the PVC has one file, `original-data`, in it. We need to add a few other files to test a restore a little later. First, let's verify the current contents of the PVC using `kubectl exec` to list the directory contents. If you are following along on your own cluster, you will need to change the `busybox-pvc` pod name to whatever is in use on your cluster. You can get the pod name using `kubectl get pods -n demo`:

```
kubectl get pods -n demo
```

This will list the `busybox` pod information. You will need the pod name to execute the `exec` command to create the files in the PVC:

NAME	READY	STATUS	RESTARTS	AGE
busybox-pvc-6cb895b675-grnxq	1/1	Running	0	4m29s
kubectl exec -it busybox-pvc-f7bfbcc44-vfsnf -n demo -- ls /mnt -la				
total 8				
drwxrwxrwx 2 root root 4096 Dec 7 15:41 .				
drwxr-xr-x 1 root root 4096 Dec 7 15:41 ..				
-rw-r--r-- 1 root root 0 Dec 7 15:41 original-data				

Now, let's add two additional files called `newfile1` and `newfile2` to the pod. To do this, we will use another `kubectl exec` command that will touch files in the `/mnt` directory.

```
kubectl exec -it busybox-pvc-f7bfbcc44-vfsnf -n demo -- touch /mnt/newfile1  
kubectl exec -it busybox-pvc-f7bfbcc44-vfsnf -n demo -- touch /mnt/newfile2
```

Again, to verify the data has been written successfully, we can list the contents using `kubectl exec`:

```
kubectl exec -it busybox-pvc-f7bfbcc44-vfsnf -n demo -- ls /mnt -la  
total 8  
drwxrwxrwx 2 root root 4096 Dec 7 15:48 .  
drwxr-xr-x 1 root root 4096 Dec 7 15:46 ..  
-rw-r--r-- 1 root root 0 Dec 7 15:48 newfile1  
-rw-r--r-- 1 root root 0 Dec 7 15:48 newfile2  
-rw-r--r-- 1 root root 0 Dec 7 15:41 original-data
```

Great! We can see that the two new files have been created. Now that we have new data in the pod, we can move on to backing up our cluster.

To create a one-time backup, execute the `velero` command with the `backup create <backup name>` option. In our example, we have named the backup `initial-backup`:

```
velero backup create initial-backup
```

The only confirmation you will receive from this is that the backup request was submitted:

```
Backup request "initial-backup" submitted successfully.  
Run `velero backup describe initial-backup` or `velero backup logs initial-backup` for more details.
```

Fortunately, Velero also tells you the command to check the backup status and logs. The last line of the output tells us that we can use the `velero` command with the `backup` option and either `describe` or `logs` to check the status of the backup operation.

The `describe` option will show all of the details of the job. An example is shown below:

```
velero backup describe initial-backup  
Name:           initial-backup  
Namespace:      velero  
Labels:         velero.io/storage-location=default  
Annotations:    velero.io/resource-timeout=10m0s  
                  velero.io/source-cluster-k8s-gitversion=v1.28.0  
                  velero.io/source-cluster-k8s-major-version=1  
                  velero.io/source-cluster-k8s-minor-version=28  
  
Phase:          Completed  
  
Namespaces:  
  Included:    *  
  Excluded:   <none>  
  
Resources:  
  Included:    *  
  Excluded:   <none>  
  Cluster-scoped: auto  
  
Label selector: <none>  
  
Or label selector: <none>  
  
Storage Location: default
```

```
Velero-Native Snapshot PVs: auto
Snapshot Move Data: false
Data Mover: velero

TTL: 720h0m0s

CSISnapshotTimeout: 10m0s
ItemOperationTimeout: 4h0m0s

Hooks: <none>

Backup Format Version: 1.1.0

Started: 2023-12-06 18:46:57 +0000 UTC
Completed: 2023-12-06 18:48:15 +0000 UTC

Expiration: 2024-01-05 18:46:57 +0000 UTC

Total items to be backed up: 641
Items backed up: 641

Velero-Native Snapshots: <none included>

kopia Backups (specify --details for more information):
Completed: 4
```

Notice the last section. This section tells us that Velero backed up 4 PVCs using the Kopia data mover. We will show this more when we perform a backup.

To reinforce the previous section, where we mentioned some of the CRDs that Velero uses, we also want to explain where the Velero utility retrieves this information from.

Each backup that is created will create a backup object in the Velero namespace. For our initial backup, a new backup object named `initial-backup` was created. Using `kubectl`, we can describe the object to see similar information that the Velero executable will provide.

As shown in the preceding output, the `describe` option shows you all of the settings for the backup job. Since we didn't pass any options to the backup request, the job contains all the namespaces and objects. Some of the most important details to verify are the phase, the total items to be backed up, and the items backed up.

If the status of the phase is anything other than `success`, you may not have all the items that you want in your backup. It's also a good idea to check the backed-up items; if the number of items backed up is less than the items to be backed up, our backup did not back up all of the items.

You may need to check the status of a backup, but you may not have the Velero executable installed. Since this information is in a CR, we can describe the CR to retrieve the backup details. Running `kubectl describe` on the backup object will show the status of the backup:

```
kubectl describe backups initial-backup -n velero
```

If we jump to the bottom of the output from the `describe` command, you will see the following:

```
Name:           initial-backup
Namespace:      velero
Labels:         velero.io/storage-location=default
Annotations:   velero.io/resource-timeout: 10m0s
                velero.io/source-cluster-k8s-gitversion: v1.28.0
                velero.io/source-cluster-k8s-major-version: 1
                velero.io/source-cluster-k8s-minor-version: 28
API Version:   velero.io/v1
Kind:          Backup
Metadata:
  Creation Timestamp: 2023-12-06T18:46:57Z
  Generation:        15
  Resource Version:  35606
  UID:              005da7ae-f270-470e-8907-c122815af365
Spec:
  Csi Snapshot Timeout:      10m0s
  Default Volumes To Fs Backup: true
  Hooks:
  Included Namespaces:
    *
  Item Operation Timeout:  4h0m0s
Metadata:
  Snapshot Move Data:  false
  Storage Location:   default
  Ttl:                 720h0m0s
  Volume Snapshot Locations:
    default
Status:
  Completion Timestamp: 2023-12-06T18:48:15Z
  Expiration:          2024-01-05T18:46:57Z
  Format Version:      1.1.0
  Phase:               Completed
Progress:
  Items Backed Up:  641
```

Total Items:	641
Start Timestamp:	2023-12-06T18:46:57Z
Version:	1
Warnings:	3
Events:	<none>

In the output, you can see that the phase is completed, the start and completion times, and the number of objects that were backed up and included in the backup.

It's good practice to use a cluster add-on that can generate alerts based on information in log files or the status of an object, such as [AlertManager](#).

You always want a successful backup, and if a backup fails, you should look into the failure immediately.

To verify that the backup is correctly stored in our S3 target, go back to the [MinIO](#) console, and if you are not already in the **Bucket** view, click **Buckets** on the left-hand side. If you are already on the **Bucket** screen, press *F5* to refresh your browser to update the view. Once the view has been refreshed, you should see that the **velero** bucket has objects stored in it. Clicking on the bucket will show you another screen, where you will see two folders, one for backup and one for Kopia. Velero will split the data from the Kubernetes objects by storing data in the **kopia** (or **restic**, if that was used as the data mover) folder. All Kubernetes objects will be stored in the **backups** folder.

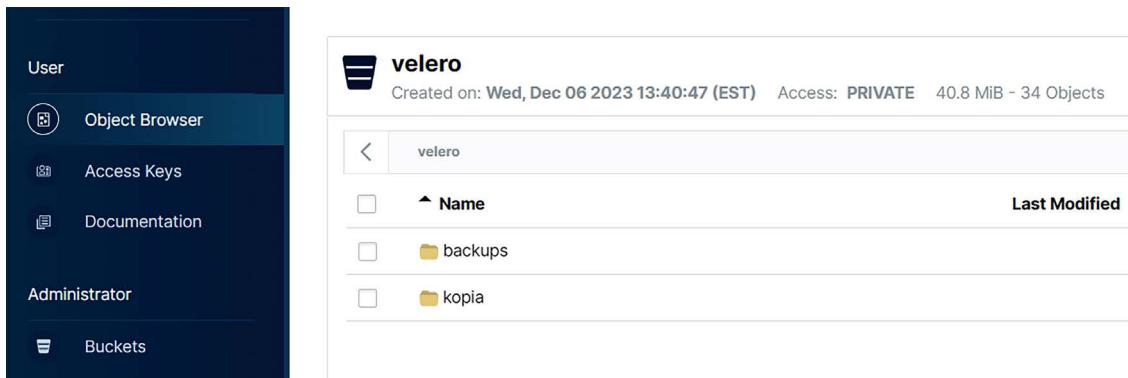


Figure 14.3: Bucket details

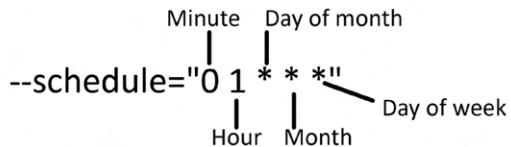
Since the overview of the **velero** bucket shows storage usage and a number of objects, and we see the **backups** and **kopia** folders, we can safely assume that the initial backup was successful. We will use this backup to restore a deleted namespace in the *Restoring from a backup* section of this chapter.

A one-off backup is not something you will likely run often. You should back up your cluster on a regular, scheduled basis. In the next section, we will explain how to create a scheduled backup.

## Scheduling a cluster backup

Creating a one-time backup is useful if you have a cluster operation scheduled or if there is a major software upgrade in a namespace. Since these events will be rare, you will want to schedule backing up the cluster at regular intervals, rather than random one-time backups.

To create a scheduled backup, you use the `schedule` option and create a tag with the Velero executable. Along with the `schedule` and creating the tag, you need to provide a name for the job and the `schedule` flag, which accepts cron-based expressions. The following schedule tells Velero to back up at 1 A.M. every day:



*Figure 14.4: Cron scheduling expression*

Using the information in *Figure 14.4*, we can create a backup that will run at 1 A.M., using the following `velero schedule create` command:

```
velero schedule create cluster-daily-1 --schedule="0 1 * * *"
```

Velero will reply that a schedule has been successfully created:

```
Schedule "cluster-daily-1" created successfully.
```

If you are not familiar with cron and the options that are available, you should read the `cron` package documentation at <https://godoc.org/github.com/robfig/cron>.

cron will also accept some shorthand expressions, which may be easier than using the standard cron expressions. The following table contains the shorthand for predefined schedules:

Shorthand value	Description
<code>@yearly</code>	Executes once a year at midnight on January 1 <sup>st</sup>
<code>@monthly</code>	Executes once a month, on the first day of the month, at midnight
<code>@weekly</code>	Executes once a week, on Sunday morning at midnight
<code>@daily</code>	Executes daily at midnight
<code>@hourly</code>	Executes at the beginning of each hour

*Table 14.4: cron shorthand scheduling*

Using the values from the shorthand table to schedule a backup job that executes daily at midnight, we use the following Velero command:

```
velero schedule create cluster-daily-2 --schedule="@daily"
```

This will create a backup job that backs the cluster up at midnight, each night. You can verify that the job was created and the last time it ran by looking at the `schedules` object, using `kubectl get schedules -n velero`:

NAMESPACE	NAME	STATUS	SCHEDULE	LASTBACKUP	AGE	PAUSED
velero	cluster-daily	Enabled	@daily		63s	

Scheduled jobs will create a backup object when the job is executed. The backup name will contain the name of the schedule, with a dash and the date and time of the backup. Backup names follow the standard naming of YYYYMMDDhhmmss. Using the name from the preceding example, our initial backup was created with the name `cluster-daily-20231206200028`. Here, 20231206200028 is the date the backup ran, and 200028 is the time the backup ran in UTC time. This is the equivalent of `2021-09-30 20:00:28 +0000 UTC`.

All of our examples so far have been configured to back up all of the namespaces and objects in the cluster. You may need to create different schedules or exclude/include certain objects based on your specific clusters.

In the next section, we will explain how to create a custom backup that will allow you to use specific tags to include and exclude namespaces and objects.

## Creating a custom backup

When you create any backup job, you can provide flags to customize what objects will be included in or excluded from the backup job. Some of the most common flags are detailed here:

Flag	Description
<code>--exclude-namespaces</code>	Comma-separated list of namespaces to exclude from the backup job. <i>Example:</i> <code>--exclude-namespaces web-dev1,web-dev2</code> .
<code>--exclude-resources</code>	Comma-separated list of resources to exclude, formatted as <code>resource.group</code> . <i>Example:</i> <code>--exclude-resources storageclasses.storage.k8s.io</code> .
<code>--include-namespaces</code>	Comma-separated list of namespaces to include in the backup job. <i>Example:</i> <code>--include-namespaces web-dev1,web-dev2</code> .
<code>--selector</code>	Configures the backup to include only objects that match a label selector. Accepts a single value only. <i>Example:</i> <code>--selector app.kubernetes.io/name=ingress-nginx</code> .
<code>--ttl</code>	Configures how long to keep the backup in hours, minutes, and seconds. By default, the value is set for 30 days or <code>720h0m0s</code> . <i>Example:</i> <code>--ttl 24h0m0s</code> . This will delete the backup after 24 hours.

Table 14.5: Velero backup flags

To create a scheduled backup that will run daily and include only Kubernetes system namespaces, we would create a scheduled job using the `--include-namespaces` flag:

```
velero schedule create cluster-ns-daily --schedule="@daily" --include-namespaces ingress-nginx,kube-node-lease,kube-public,kube-system,local-path-storage,velero
```

Since Velero commands use a CLI for all operations, we should start by explaining the common commands you will use to manage backup and restore operations.

## Managing Velero using the CLI

All Velero operations must be done using the Velero executable. Velero does not include a UI for managing backups and restores. Managing a backup system without a GUI can be a challenge at first, but once you get comfortable with the Velero management commands, it becomes easy to perform operations.

The Velero executable accepts two options:

- Commands
- Flags

A **command** is an operation such as `backup`, `restore`, `install`, and `get`. Most initial commands require a second command to make a complete operation. For example, a `backup` command requires another command, such as `create` or `delete`, to form a complete operation.

There are two types of flags – command flags and global flags. **Global flags** are flags that can be set for any command, while **command flags** are specific to the command being executed.

Like many CLI tools, Velero includes built-in help for every command. If you forget some syntax or want to know what flags can be used with a command, you can use the `-h` flag to get help:

```
velero backup create -h
```

The following is the abbreviated help output for the `backup create` command:

```
Create a backup

Usage:
  velero backup create NAME [flags]

Examples:
  # Create a backup containing all resources.
  velero backup create backup1

  # Create a backup including only the nginx namespace.
  velero backup create nginx-backup --include-namespaces nginx

  # Create a backup excluding the velero and default namespaces.
  velero backup create backup2 --exclude-namespaces velero,default

  # Create a backup based on a schedule named daily-backup.
```

```
velero backup create --from-schedule daily-backup

# View the YAML for a backup that doesn't snapshot volumes, without sending
it to the server.
velero backup create backup3 --snapshot-volumes=false -o yaml

# Wait for a backup to complete before returning from the command.
velero backup create backup4 --wait
```

We find Velero's help system to be very helpful; once you get comfortable with Velero's basics, you will find that the built-in help provides enough information for most commands.

## Using common Velero commands

Since many readers may be new to Velero, we want to provide a quick overview of the most commonly used commands to get you comfortable with using Velero.

### Listing Velero objects

As we have mentioned, Velero management is driven by using the CLI. You can imagine that as you create additional backup jobs, it may become difficult to remember what has been created. This is where the `get` command comes in handy.

The CLI can retrieve, or `get`, a list of the following Velero objects:

- Backup locations
- Backups
- Plugins
- Restores
- Schedules
- Snapshot locations

As you may expect, executing `velero get <object>` will return a list of the objects managed by Velero:

```
velero get backups
```

Here is the output:

NAME	STATUS	ERRORS	WARNINGS
initial-backup	Completed	0	0

Each `get` command will produce a similar output, containing the names of each object and any unique values for the objects. This command is useful for getting a quick look at what objects exist, but it's usually used before executing the next command, `describe`.

## Retrieving details for a Velero object

After you get the name of the object that you want the details of, you can use the `describe` command to get the details of the object. Using the output from the `get` command in the previous section, we want to view the details of the `initial-backup` job:

```
velero describe backup initial-backup
```

The output of the command provides all the details for the requested object. You will find yourself using the `describe` command to troubleshoot issues such as backup failures.

## Creating and deleting objects

Since we have already used the `create` command a few times, we will focus on the `delete` command in this section.

To recap, the `create` command allows you to create objects that will be managed by Velero, including backups, schedules, restores, and locations for backups and snapshots. We have created a backup and a schedule, and in the next section, we will create a restore.

Once an object is created, you may discover that you need to delete it. To delete objects in Velero, you use the `delete` command, along with the object and name you want to delete.



Since we do not have a backup called `sales` on our KinD cluster, the example command will not find a backup called `sales`.

In our `get backups` output example, we had a backup called `sales`. To delete that backup, we would execute the following `delete` command:

```
velero delete backup sales
```

Since a delete is a one-way operation, you will need to confirm that you want to delete the object. Once you have confirmed the deletion, it may take a few minutes for the object to be removed from Velero since it waits until all associated data is removed:

```
Are you sure you want to continue (Y/N)? y
Request to delete backup "sales" submitted successfully.
The backup will be fully deleted after all associated data (disk snapshots,
backup files, restores) are removed.
```

As you can see in the output, when we delete a backup, Velero will delete all of the objects for the backup, including the snapshot's backup files and restores.

There are additional commands that you can use, but the commands covered in this section are what you really need to get comfortable with Velero. For reference, the list below shows common Velero commands and a brief description of what each command does:

**Installing and uninstalling Velero:**

- `velero install`: Installs Velero server components into the Kubernetes cluster.

**Managing backups:**

- `velero backup create <NAME>`: Creates a backup with the specified name.
- `velero backup describe <NAME>`: Describes the details of a specific backup.
- `velero backup delete <NAME>`: Deletes a specified backup.
- `velero backup logs <NAME>`: Displays logs for a specific backup.
- `velero backup download <NAME>`: Downloads the backup logs for troubleshooting purposes.
- `velero backup get`: Lists all backups.

**Managing restores:**

- `velero restore create --from-backup <BACKUP_NAME>`: Creates a restore from a specified backup.
- `velero restore describe <NAME>`: Describes the details of a specific restore.
- `velero restore delete <NAME>`: Deletes a specified restore.
- `velero restore logs <NAME>`: Displays logs for a specific restore.
- `velero restore get`: Lists all restores.

**Scheduling backups:**

- `velero schedule create <NAME> --schedule <CRON_SCHEDULE>`: Creates a scheduled backup using cron syntax.
- `velero schedule describe <NAME>`: Describes the details of a specific schedule.
- `velero schedule delete <NAME>`: Deletes a specified schedule.
- `velero schedule get`: Lists all schedules.

**Managing plugins:**

- `velero plugin add <PLUGIN_IMAGE>`: Adds a plugin to the Velero server.
- `velero plugin get`: Lists all plugins.

**Snapshot locations:**

- `velero snapshot-location create <NAME>`: Creates a new snapshot location with the specified name.
- `velero snapshot-location get`: Lists all snapshot locations.
- `velero snapshot-location describe <NAME>`: Describes the details of a specific snapshot location.
- `velero snapshot-location delete <NAME>`: Deletes a specified snapshot location.

**Backup locations:**

- `velero backup-location create <NAME>`: Establishes a new backup location with the specified name.

- `velero backup-location get`: Lists all backup locations.
- `velero backup-location describe <NAME>`: Describes the details of a specific backup location.
- `velero backup-location delete <NAME>`: Removes a specified backup location.

#### Managing restic repositories:

- `velero restic repo get`: Lists all restic repositories.
- `velero restic repo describe <NAME>`: Describes the details of a specific restic repository.
- `velero restic repo forget <NAME>`: Manually removes restic backup snapshots.
- `velero restic repo prune <NAME>`: Removes unused data from a restic repository to free up space.
- `velero restic repo garbage-collect <NAME>`: Runs a garbage collection operation on the specified repository.

#### Utility commands:

- `velero version`: Displays the current version of Velero.
- `velero client config set`: Configures the default settings for the Velero client.
- `velero client config get`: Displays the current client configuration.
- `velero completion <SHELL>`: Generates a shell completion script for the specified shell, enhancing CLI usability.

Now that you can create and schedule backups and know how to use the help system in Velero, we can move on to using a backup to restore objects.

## Restoring from a backup

In this section, we will explain the process of restoring from a backup using Velero. Having a backup is akin to having car insurance or homeowner's insurance—it's essential to have, yet you hope you never need to use it. When the unexpected happens, you'll be grateful it's there. In the realm of data backups, finding yourself needing to restore data without a backup is a scenario we often refer to as a “resume-building event.” To run a restore from a backup, you use the `create restore` command with the `--from-backup <backup name>` tag.

Earlier in the chapter, we created a single, one-time backup, called `initial-backup`, which includes every namespace and object in the cluster. If we decided that we needed to restore that backup, we would execute a restore using the Velero CLI:

```
velero restore create --from-backup initial-backup
```

The output from the `restore` command may seem odd:

```
Restore request "initial-backup-20231207163306" submitted successfully.
```

```
Run `velero restore describe initial-backup-20231207163306` or `velero restore logs initial-backup-20231207163306` for more details.
```

At first, it may seem like a backup request was made since Velero replies with "initial-backup-20231207163306" submitted successfully, but you may be wondering why the restore isn't called initial-backup. Velero uses the backup name to create a restore request, and since we named our backup initial-backup, the restore job name will use that name and append the date and time of the restore request.

You can view the status of the restore using the `describe` command:

```
velero restore describe initial-backup-20211001002927
```

Depending on the size of the restore, it may take some time to restore the entire backup. During the restore phase, the status of the backup will be `InProgress`. Once the restore is complete, the status will change to `Completed`.

## Restoring in action

With all of the theory behind us, let's use two examples to see Velero restores in action. For the examples, we will start with a simple deployment with a persistent volume that we will delete and restore on the same cluster. The second example will be more complex; we will back up a few namespaces in our main KinD cluster and restore them to a new KinD cluster.

### Restoring a deployment from a backup

In the backup section of this chapter, we created a backup of the cluster after we created a `busybox` deployment with a PVC attached. We added data to the PVC before we backed it up, and now we want to make sure that the backup is complete and that restoring a namespace is successful.

To test the restore, we will simulate a failure by deleting the `demo` namespace and then use our backup to restore the entire namespace, including the PVC data.

### Simulating a failure

To simulate an event that would require a backup of our namespace, we will delete the entire namespace using `kubectl`:

```
kubectl delete ns demo
```

It may take a minute to delete the objects in the namespace. Once you have returned to a prompt, the deletion should be complete.

Verify that the namespace has been deleted before moving on. Run a `kubectl get ns` and verify that the `demo` namespace is no longer listed.

With the confirmation that the `demo` namespace has been deleted, we will demonstrate how to restore the entire namespace and objects from the backup.

## Restoring a namespace

Imagine this is a real-life scenario. You receive a phone call that a developer has accidentally deleted every object in their namespace and they do not have the source files.

Of course, you are prepared for this type of event. You have several backup jobs running in your cluster, and you tell the developer that you can restore it to the state it was in last night from a backup.

We want to restore just the one namespace, `demo`, rather than the entire cluster. We know our backup name is `initial-backup`, so we will need to use that as our backup file when we execute a restore. To limit the restore to just a namespace, we will add the `--include-namespaces demo` flag to our command.

```
velero restore create --from-backup initial-backup --include-namespaces demo
Restore request "initial-backup-20231207164723" submitted successfully.
Run `velero restore describe initial-backup-20231207164723` or `velero restore
logs initial-backup-20231207164723` for more details.
```

This will start a restore from the `initial-backup`. It shouldn't take too long to restore since it's a single namespace and the PVC only has a few empty files to restore.

First, check to make sure that the namespace has been recreated. If you execute `kubectl get ns` `demo`, you should see the `demo` namespace in the list:

```
kubectl get ns demo
NAME    STATUS    AGE
demo   Active   2m47s
```

Great! That's the first step. Now, let's make sure the pods were restored. We will need the name to look at the PVC contents:

```
kubectl get pods -n demo
NAME                  READY   STATUS    RESTARTS   AGE
busybox-pvc-f7bfbcc44-vfsnf   1/1     Running   0          4m
```

Looking good so far. Finally, let's use `kubectl exec` to look at the `/mnt` directory in the pod. We are hoping to see the new files we created before the backup:

```
kubectl exec -it busybox-pvc-f7bfbcc44-vfsnf -n demo -- ls /mnt -la
Defaulted container "busybox-pvc" out of: busybox-pvc, restore-wait (init)
total 12
drwxrwxrwx    3 root      root        4096 Dec  7 16:47 .
drwxr-xr-x    1 root      root        4096 Dec  7 16:47 ..
drwxr-xr-x    2 root      root        4096 Dec  7 16:47 .velero
-rw-r--r--    1 root      root         0 Dec  7 15:48 newfile1
-rw-r--r--    1 root      root         0 Dec  7 15:48 newfile2
-rw-r--r--    1 root      root         0 Dec  7 16:47 original-data
```

As we can see from the output of the `ls` command, the two files we added before executing the backup, `newfile1` and `newfile2`, are in the pod, proving that the backup works for restoring the namespace, including any persistent data.

Congratulations! You just saved the developer a lot of work because you had a backup of the namespace!

Restoring objects like the previous example is a common exercise, and backing up and restoring in the same cluster is the only thing some operators may think backups are good for. While that may be the most common use case for backups, they can also be used for a number of other activities, like using a backup from one cluster into another, different cluster.

In the next section, we will use a backup from one cluster and restore the data to a different cluster. This is beneficial for a few scenarios, including migrating an application from one cluster to another or restoring the application and data to a development cluster to perform upgrade tests.

## Using a backup to create workloads in a new cluster

Restoring objects in a cluster is just one use case for Velero. While it is the main use case for most users, you can also use your backup files to restore a workload or all workloads on another cluster. This is a useful option if you need to create a new development or disaster recovery cluster.

Remember that Velero backup jobs are only the namespaces and objects in the namespaces. To restore a backup to a new cluster, you must have a running cluster running Velero before you can restore any workloads.

## Backing up the cluster

By this point in the chapter, we assume that you have seen this process a few times and that you know how to use the Velero CLI. If you need a refresher, you can go back a few pages in the chapter for reference, or use the CLI help function.

For this example, we will not work with any data. Instead, we just want to demonstrate restoring a backup from one cluster to a different cluster.

First, we should create a few namespaces and add some deployments to each one to make it more interesting. We have included a script in the `chapter14` folder called `create-backup-objects.yaml` that will create the namespaces and the objects for you. Run that script to create the namespaces and deployment.

Once the namespaces and deployment have been created, let's create a new backup called `namespace-demo` that will back up only the four new namespaces that we created with the script:

```
velero backup create namespace-demo --include-namespaces=demo1,demo2,demo3,de  
mo4
```

Before moving on, verify that the backup has been completed successfully. You can verify the backup by executing the `describe` command against the `namespace-demo` backup:

```
velero backup describe namespace-demo
```

In the output, you will see that the backup includes the four namespaces and there are 40 objects in the backup. An abbreviated output is shown below.

```
Phase: Completed

Namespaces:
 Included: demo1, demo2, demo3, demo4
 Excluded: <none>

Resources:
 Included: *
 Excluded: <none>
 Cluster-scoped: auto

Label selector: <none>

Or label selector: <none>

Storage Location: default

Started: 2023-12-07 16:58:02 +0000 UTC
Completed: 2023-12-07 16:58:11 +0000 UTC

Expiration: 2024-01-06 16:58:02 +0000 UTC

Total items to be backed up: 36
Items backed up: 36

Velero-Native Snapshots: <none included>
 Included: demo1, demo2, demo3, demo4
 Excluded: <none>

Started: 2021-10-01 00:44:30 +0000 UTC
Completed: 2021-10-01 00:44:42 +0000 UTC
Expiration: 2021-10-31 00:44:30 +0000 UTC

Total items to be backed up: 40
Items backed up: 40
```

You now have a new backup that contains the four new namespaces and their objects. Now, using this backup, we will restore the four namespaces to a new cluster.

First, we need to deploy a new KinD cluster that, which will use to restore our demo1, demo2, demo3, and demo4 namespaces.

## Building a new cluster

Since we are only demonstrating how Velero can be used to create workloads on a new cluster from a backup, we will create a simple single-node KinD cluster as our restore point.

This section gets a little complex since you will have two clusters in your `kubeconfig` file. Follow the steps carefully if you're new to switching config contexts.

Once we have completed this exercise, we will delete the second cluster since we will not need to have two clusters. This exercise will be interactive. You will need to execute each step:

1. Create a new KinD cluster called `velero-restore`:

```
kind create cluster --name velero-restore
```

This will create a new single-node cluster that contains both the control plane and worker node, and it will set your cluster context to the new cluster.

2. Once the cluster has deployed, verify that your context has been switched to the `velero-restore` cluster:

```
kubectl config get-contexts
```

The output is as follows:

CURRENT	NAME	CLUSTER	AUTHINFO
	kind-cluster01	kind-cluster01	kind-cluster01
*	kind-velero-restore	kind-velero-restore	kind-velero-restore

3. Verify that the current context is set to the `kind-velero-restore` cluster. You will see an \* in the current field of the cluster that is being used.
4. Finally, verify the namespaces in the cluster using `kubectl`. You should only see the default namespaces that are included with a new cluster:

NAME	STATUS	AGE
default	Active	4m51s
kube-node-lease	Active	4m54s
kube-public	Active	4m54s
kube-system	Active	4m54s
local-path-storage	Active	4m43s

Now that we have created a new cluster, we can start the process of restoring the workloads. The first step is to install Velero on the new cluster, pointing to the existing S3 bucket as the backup location.

## Restoring a backup to the new cluster

With our new KinD cluster up and running, we need to install Velero to restore our backup. We can use most of the same manifests and settings that we used in the original cluster, but since we are in a different cluster, we need to change the S3 target to the external URL we used to expose MinIO.

## Installing Velero in the new cluster

We already have the `credentials-velero` file in the `chapter14` folder, so we can jump right into installing Velero using the `velero install` command. To follow these steps, you should be in the `chapter14` directory:

1. Be sure to change the `s3Url` to your MinIO Ingress rule for your original KinD cluster that was created earlier in the chapter. If you forgot the ingress name, change your context to `kind-cluster01` and use `kubectl` to look at the rules in the `velero` namespace, `kubectl get ingress -n velero`. This will show you the full `nip.io` for MinIO (remember, don't use the `minio-console` rule):

```
velero install --provider aws --plugins velero/velero-plugin-for-aws:v1.2.0 --bucket velero --secret-file ./credentials-velero --use-volume-snapshots=false --backup-location-config region=minio,s3ForcePathStyle="true",s3Url=http://minio.10.2.1.161.nip.io --use-node-agent --default-volumes-to-fs-backup
```

2. The install will take a few minutes, but once the pod is up and running, view the log files to verify that the Velero server is up and running and connected to the S3 target:

```
kubectl logs deployment/velero -n velero
```

3. If all of your settings were correct, the Velero log will have an entry saying that it has found backups in the backup location that need to be synced with the new Velero server (the number of backups may be different for your KinD cluster):

```
time="2021-10-01T23:53:30Z" level=info msg="Found 2 backups in the backup location that do not exist in the cluster and need to be synced" backupLocation=default controller=backup-sync logSource="pkg/controller/backup_sync_controller.go:204"
```

4. After confirming the installation, verify that Velero can see the existing backup files using `velero get backups`:

NAME	STATUS	ERRORS	WARNINGS
initial-backup	Completed	0	0
namespace-demo	Completed	0	0

Your backup list will differ from ours, but you should see the same list that you had in the original cluster.

At this point, we can use any of the backup files to create a restore job in the new cluster.

## Restoring a backup in a new cluster

In this section, we will use the backup that was created in the previous section and restore the workloads to a brand new KinD cluster to simulate a workload migration.

The backup that was created of the original cluster, after we added the namespaces and deployment, was called `namespace-demo`:

1. Using that backup name, we can restore the namespaces and objects by running the `velero create restore` command:

```
velero create restore --from-backup=namespace-demo
```

2. Wait for the restore to complete before moving on to the next step. To verify that the restore was successful, use the `velero describe restore` command with the name of the restore job that was created when you executed the `create restore` command. In our cluster, the restore job was assigned the name `namespace-demo-20211001235926`:

```
velero restore describe namespace-demo-20211001235926
```

3. Once the phase has changed from `InProgress` to `Completed`, verify that your new cluster has the additional demo namespaces using `kubectl get ns`:

NAME	STATUS	AGE
calico-apiserver	Active	23m
calico-system	Active	24m
default	Active	24m
demo1	Active	15s
demo2	Active	15s
demo3	Active	15s
demo4	Active	15s
ingress-nginx	Active	24m
kube-node-lease	Active	24m
kube-public	Active	24m
kube-system	Active	24m
local-path-storage	Active	24m
tigera-operator	Active	24m
velero	Active	5m18s

4. You will see that the new namespaces were created, and if you look at the pods in each namespace, you will see that each has a pod called `nginx`. You can verify that the pods were created using `kubectl get pods`. For example, to verify the pods in the `demo1` namespace, enter the following: `kubectl get pods -n demo1`.

The output is as follows:

NAME	READY	STATUS	RESTARTS	AGE
nginx	1/1	Running	0	3m30s

Congratulations! You have successfully restored objects from one cluster into a new cluster.

## Deleting the new cluster

Since we do not need two clusters, let's delete the new KinD cluster that we restored the backup to:

1. To delete the cluster, execute the `kind delete cluster` command:

```
kind delete cluster --name velero-restore
```

2. Set your current context to the original KinD cluster, `kind-cluster01`:

```
kubectl config use-context kind-cluster01
```

Now that we have cleaned up the temporary second cluster, we have completed the chapter.

## Summary

Backing up clusters and workloads is a requirement for any enterprise cluster. Having a backup solution allows you to recover from a disaster or human error. A typical backup solution allows you to restore any Kubernetes object, including namespaces, persistent volumes, RBAC, services, and service accounts. You can also take all of the workloads from one cluster and restore them on a completely different cluster for testing or troubleshooting.

In this chapter, we reviewed how to back up the etcd cluster database using `etcdctl` and the snapshot feature. We also went into detail on how to install Velero in a cluster to back up and restore workloads. We closed out the chapter by copying workloads from an existing backup by restoring an existing backup on a new cluster.

Coming up in the next chapter, we will introduce you to monitoring your clusters and workloads.

## Questions

1. True or false – Velero can only use an S3 target to store backup jobs.
  - a. True
  - b. False
2. If you do not have an object storage solution, how can you provide an S3 target using a backend storage solution such as NFS?
  - a. You can't – there is no way to add anything in front of NFS to present S3.
  - b. Kubernetes can do this using native CSI features.
  - c. Install MinIO and use the NFS volumes as persistent disks in the deployment.
  - d. You don't need to use an object store; you can use NFS directly with Velero.
3. True or false – Velero backups can only be restored on the same cluster where the backup was originally created.
  - a. True
  - b. False

4. What utility can you use to create an etcd backup?
  - a. Velero.
  - b. MinIO.
  - c. There is no reason to back up the etcd database.
  - d. etcdctl.
  
5. Which command will create a scheduled backup that runs every day at 3 A.M.?
  - a. velero create backup daily-backup
  - b. velero create @daily backup daily-backup
  - c. velero create backup daily-backup -schedule="@daily3am"
  - d. velero create schedule daily-backup --schedule="0 3 \* \* \*"

## Answers

1. a
2. a
3. b
4. d
5. d



# 15

## Monitoring Clusters and Workloads

So far in this book, we've spent a considerable amount of time standing up different aspects of an enterprise Kubernetes infrastructure. Once it's stood up, how do you know it's healthy? How do you know it's running? Do you know when there's a problem before your users do, or are you first finding out when someone can't access a critical system? Monitoring is a critical aspect of any well-run infrastructure that has its own unique challenges in the Kubernetes and Cloud Native world. In this chapter, we're going to look at two specific aspects of monitoring. First, we're going to work with the Prometheus project and its integration with Kubernetes to understand how to inspect our cluster and what to look for. Next, we're going to centralize our logs using the popular ELK stack. Along the way, we'll include typical enterprise discussions around security and compliance to make sure we're working within our enterprise's requirements.

In this chapter, we're going to cover the following main topics:

- Managing Metrics in Kubernetes
- Log Management in Kubernetes

Next, let's review the technical requirements.

### Technical Requirements

This chapter will involve a larger workload than previous chapters, so a more powerful cluster will be needed. This chapter has the following technical requirements:

- An Ubuntu 22.04+ server running Docker with a minimum of 8 GB of RAM, though 16 GB is suggested
- Scripts from the chapter15 folder from the repo, which you can access by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition>

## Getting Help

We do our best to test everything, but there are sometimes half a dozen systems or more in our integration labs. Given the fluid nature of technology, sometimes things that work in our environment don't work in yours. Don't worry, we're here to help! Open an issue on our GitHub repo at <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/issues> and we'll be happy to help you out!

## Managing Metrics in Kubernetes

Once upon a time, monitoring and metrics were a complex and very proprietary corner of the industry. While there were some open-source projects that did monitoring, the majority of “enterprise” systems were large, cumbersome, and proprietary. There were a few standards, such as SNMP, but for the most part, every vendor had their own agents, their own configurations, their own...everything. If you wanted to write an application that generated metrics or alerts, then you needed to write to their SDK. This led to monitoring being one of the centralized services, like databases, but required much deeper understanding of what's being monitored. Changes were difficult and ultimately, many systems followed either **you only live once (YOLO)** monitoring or very basic high-level monitoring that “checked the compliance box,” but didn't provide much value.

Then came the Prometheus project, which made two critical improvements to the monitoring process that really changed the way we approach monitoring. The first change was to do everything via simple HTTP requests. If you want to monitor something, it needs to expose a URL that provides metrics data. It doesn't matter whether it's a website or a database. The second major impact was that these metrics endpoints provide data text format that makes it easy to dynamically generate a response regardless of the monitoring system. We'll dive into the details later, but this format is so powerful and flexible that it has been adopted by SaaS monitoring systems in addition to Prometheus. Datadog, AWS CloudWatch, and so on all support the Prometheus endpoint and format, making it much easier to start with Prometheus and move to a provided solution without changing your applications.

In addition to opening the ability to monitor disparate systems, Prometheus made it easier for operators to interact with that data by providing it via APIs. Now, common visualization tools, such as Grafana, can access that data without a vendor's proprietary UI. These tools build on the base offering of Prometheus, expanding your capabilities to monitoring and alerting as well.

Now that we've explained why Prometheus has had such a large impact on the monitoring world, we'll next walk through how your Kubernetes clusters provide metrics data and how to leverage it.

## How Kubernetes Provides Metrics

Kubernetes provides a `/metrics` URI on the API server. This API requires an authorized token to be able to access it. To access this endpoint, let's create a `ServiceAccount`, `ClusterRole`, and `ClusterRoleBinding`:

```
$ kubectl create sa getmetrics  
$ kubectl create clusterrole get-metrics --non-resource-url=/metrics --verb=get
```

```
$ kubectl create clusterrolebinding get-metrics --clusterrole=get-metrics  
--serviceaccount=default:getmetrics  
$ export TOKEN=$(kubectl create token getmetrics -n default)  
$ curl -v --insecure -H "Authorization: Bearer $TOKEN" https://0.0.0.0:6443/  
metrics  
# HELP aggregator_discovery_aggregation_count_total [ALPHA] Counter of number  
of times discovery was aggregated  
.  
.  
.
```

This is going to take a while; there are too many metrics that are collected to document here. We'll talk about some individual metrics after we get some context on how the metrics are created and consumed by Prometheus. The main point to understand now is that all metrics from your cluster come from a single URL and that those metrics require authentication. You could disable this requirement by making the /metrics endpoint available to the system (unauthenticated user, which is what all unauthenticated requests are assigned to), but this would now open your cluster up to potential escalation attacks. It's better to keep this resource protected.

If you spend even a moment looking at this data, you'll see there is an incredible amount. We're going to first dive into deploying Prometheus to make it easier for you to interact with this data. Thankfully, deploying a full monitoring stack on Kubernetes is pretty simple!

## Deploying the Prometheus Stack

So far, we've found how to access the Kubernetes metrics endpoint; next, we're going to deploy Prometheus using the kube-prometheus-stack project (<https://github.com/prometheus-community/helm-charts/tree/main/charts/kube-prometheus-stack>) from the Prometheus Community. This chart combines the Prometheus project with Grafana and Alertmanager to create a mostly complete monitoring solution. We'll walk through some of the gaps later in the chapter. First, let's get Prometheus deployed:

```
$ cd chapter15/simple  
$ ./deploy-prometheus-charts.sh
```

This script creates the monitoring namespace, deploys the Helm chart, and creates an Ingress object with the host `prometheus.apps.X-X-X-X.nip.io`, where X-X-X-X is your API server's IP address but with dashes instead of dots. For me, my API server is running on 192.168.2.82, so, to access the Prometheus UI in my browser, I go to <https://prometheus.apps.192-168-2-82.nip.io/>.

Now that Prometheus is running and we can access it, the next step is to walk through some of Prometheus' capabilities.

## Introduction to Prometheus

The first thing you'll notice when accessing Prometheus is there's no login screen. Prometheus has no concept of security. Anyone with access to your URL will have access to your Prometheus in the current setup. We'll take care of this problem later in the chapter as we look at operationalizing Prometheus.

Having seen that there's no security in Prometheus, the next thing to note is that the main screen, known as the **Graph** view, gives you an **Expression** box. This is where you can look for any of the expressions available using PromQL, Prometheus' query language. For instance, using the `sum by (namespace) (kube_pod_info)` query gives you a list of all the pods in each namespace:

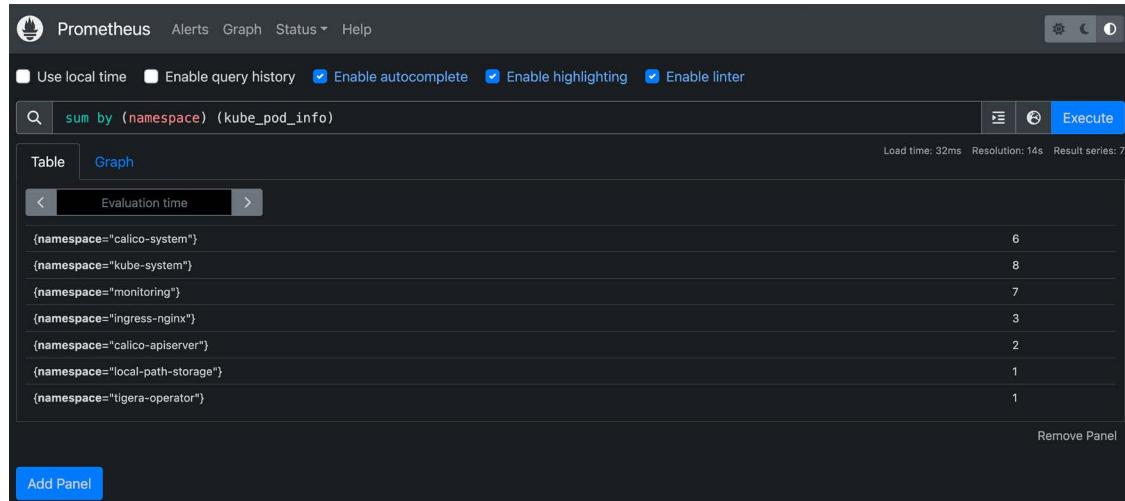


Figure 15.1: Prometheus query

This screen has a menu bar that includes **Alerts** and **Status** menu options as well. If you click on **Alerts**, you'll see several alerts are red and firing. This is because we're running inside of KinD, which has its own networking quirks. Depending on the kind of Kubernetes distribution, you'll find that some alerts are firing on a regular basis and can be ignored.

While Prometheus is configured to generate alerts, it doesn't have any mechanism for notification. It instead relies on an outside system. The typical open source tool used is Alertmanager, but we'll cover that later in the chapter. For now, what's important to know is that alerts are defined in Prometheus and you can check their status from the **Alerts** view.

The screenshot shows the Prometheus 'Alerts' view. At the top, there are three tabs: 'Inactive (138)', 'Pending (0)', and 'Firing (1)'. A search bar labeled 'Filter by name or labels' is next to a magnifying glass icon. To the right, a checkbox for 'Show annotations' is checked. Below the tabs, a list of alerts is displayed in colored boxes. The first section, under '/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulesfiles-0/monitoring-prometheus-kube-prometheus-alertmanager.rules-0e3a0017-de72-43d9-9fb0-beaf0d6c9a2.yaml > alertmanager.rules', contains 138 inactive alerts. The second section, under '/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulesfiles-0/monitoring-prometheus-kube-prometheus-config-reloaders-3dbff23e-1e28-4809-ac25-aa899fec83db.yaml > config-reloaders', contains 1 inactive alert. The third section, under '/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulesfiles-0/monitoring-prometheus-kube-prometheus-etcd-6561ab7b-7302-4a10-9575-1dc782f0a770.yaml > etcd', contains 1 pending alert. The alerts are categorized by severity: green for inactive, yellow for pending, and red for firing.

Figure 15.2: Alerts view

Finally, there's the **Status** menu, which provides several views. The one I find myself using the most is the **Targets** view, which will tell you if any targets aren't available.

The screenshot shows the Prometheus 'Targets' view. At the top, there are three tabs: 'All scrape pools' (selected), 'All' (Unhealthy), and 'Collapse All'. A search bar labeled 'Filter by endpoint or labels' is next to a magnifying glass icon. To the right, checkboxes for 'Unknown', 'Unhealthy', and 'Healthy' are checked. Below the tabs, three tables of targets are displayed:

- serviceMonitor/monitoring/prometheus-kube-prometheus-alertmanager/0 (1/1 up)**: Shows one target at <http://10.240.189.140:9093/metrics>. The state is UP. Labels include: container="alertmanager", endpoint="http-web", instance="10.240.189.140:9093", job="prometheus-kube-prometheus-alertmanager", namespace="monitoring", pod="alertmanager-prometheus-kube-prometheus-alertmanager-0", service="prometheus-kube-prometheus-alertmanager". Last Scrape was 23.747s ago, Scrape Duration was 3.353ms, and there were no errors.
- serviceMonitor/monitoring/prometheus-kube-prometheus-alertmanager/1 (1/1 up)**: Shows one target at <http://10.240.189.140:8080/metrics>. The state is UP. Labels include: container="config-reloader", endpoint="reloader-web", instance="10.240.189.140:8080", job="prometheus-kube-prometheus-alertmanager", namespace="monitoring", pod="alertmanager-prometheus-kube-prometheus-alertmanager-0", service="prometheus-kube-prometheus-alertmanager". Last Scrape was 3.587s ago, Scrape Duration was 1.998ms, and there were no errors.
- serviceMonitor/monitoring/prometheus-kube-prometheus-apiserver/0 (1/1 up)**: Shows one target at <https://172.18.0.3:6443/metrics>. The state is UP. Labels include: endpoint="https", instance="172.18.0.3:6443", job="apiserver", namespace="default", service="kubernetes". Last Scrape was 28.721s ago, Scrape Duration was 103.363ms, and there were no errors.
- serviceMonitor/monitoring/prometheus-kube-prometheus-coredns/0 (2/2 up)**: Shows two targets. The first is at <http://10.240.207.194:9153/metrics> (state UP) with labels: container="coredns", endpoint="http-metrics", instance="10.240.207.194:9153", job="coredns", namespace="kube-system", pod="coredns-7dd0ff1f64-4spnf". Last Scrape was 22.715s ago, Scrape Duration was 1.871ms, and there were no errors. The second is at <https://172.18.0.3:8080/metrics> (state UP) with labels: container="coredns", endpoint="https", instance="172.18.0.3:8080", job="coredns", namespace="kube-system", pod="coredns-7dd0ff1f64-4spnf". Last Scrape was 22.715s ago, Scrape Duration was 1.871ms, and there were no errors.

Figure 15.3: Targets view

It was important to get Prometheus up and running before we dove into the details of how metrics are collected or queried. As we saw when we first looked at Kubernetes' metrics, there is a massive amount of data and it's not in a format that's easily analyzed via command-line tools. With the GUI in hand, we can begin to look at how Prometheus collects and stores metrics.

## How Does Prometheus Collect Metrics?

So far, we've polled the metrics endpoint for Kubernetes and gotten the Prometheus stack up and running to query and analyze that data. Earlier, we created a simple query to look for the number of pods by Namespace: `sum by (namespace) (kube_pod_info)`. Looking at the output of our API server metrics pull, we can grep for `kube_pod_info`, and we won't find anything! That's because this particular metric doesn't come directly from the API server. It instead comes from the `kube-state-metrics` project (<https://github.com/kubernetes/kube-state-metrics>), which is deployed with the Prometheus stack chart. This tool generates data about the API server in a way that can be integrated into Prometheus. If we look at its `/metrics` output, we'll find:

```
# HELP kube_pod_deletion_timestamp Unix deletion timestamp
# TYPE kube_pod_deletion_timestamp gauge
# HELP kube_pod_info [STABLE] Information about pod.
# TYPE kube_pod_info gauge
kube_pod_info{namespace="calico-system",pod="calico-typpha-699dc7b758-
bgr5f",uid="33ec61b1-bb56-4a4c-853c-6a0ee56023c2",host_ip="172.18.0.2",pod_
ip="172.18.0.2",node="cluster01-worker",created_by_kind="ReplicaSet",created_
by_name="calico-typpha-699dc7b758",priority_class="system-cluster-
critical",host_network="true"} 1
```

The lines with a hash mark or pound, #, provide metadata for the upcoming metrics. For each metric, the form is:

```
metric_name{annotation1="value1",annotation2="value2"} value
```

The annotations on each metric are what allow Prometheus to be able to index so much information and make it easy to query. Looking at the `kube_pod_info` metrics, we see an annotation for `namespace`. This means that we can ask Prometheus to give us all the instances of the `kube_pod_info` metric, broken up by the annotation of `namespace`. We could also ask for any pods on a specific `host_ip`, `node`, or any other of the annotations.

The type of the `kube_pod_info` metric is a gauge. There are four types of metrics in Prometheus:

- **Counter:** Counters can only increase over time or go back down to zero. An example of a counter would be the number of requests an application responded to over its lifetime. The number of requests will only increase until the pod dies, at which point, it goes back to zero.
- **Gauge:** These metrics can go up or down over time. For instance, the number of open sessions a pod has would be a gauge because it can fluctuate over time.

- **Histogram:** This type is more complex. It's designed to allow you to track ranges, or buckets, of request types. For instance, if you wanted to track response times for requests, you could create buckets for likely times and increase the count for each bucket. This is much more efficient than generating a new metric instance for each request. If we did generate a metric instance for each request, we might have thousands of data points every second that need to be indexed and stored and that data just wouldn't be helpful. Instead of using a histogram, we can categorize ranges and track them that way, saving on processing and data storage.
- **Summary:** Summary metrics are similar to histograms but are managed by the client. Generally speaking, you'll want to use histograms.

When Prometheus collects these metrics, they're stored in an internal database. You'll see that both the Prometheus and Alertmanager pods are part of `StatefulSets`, not `Deployments`. That's because they store data locally. For Prometheus, the data is stored so that you can not only see the latest version of the metric but also the past instances of that metric, too. From the main `Graph` screen in Prometheus, you can click on `Graph` for any result to see the result over time. Our cluster is small and isn't running much, but what if we had an explosion of new pods? That could trigger an alarm. Another area where keeping the history of metrics is important is for alerting. When we get to defining our alerting rules, we'll see that we can specify to only fire or clear an alert if it happens over a certain period of time. Stuff happens; you don't want your pager going off for every packet that gets dropped. Tracking this information is very important for both the value of the data and accurate alerting.

In this section, we looked at how Prometheus collects and stores metrics. Next, we'll dive into common metrics you'll want to keep an eye on for Kubernetes.

## Common Kubernetes Metrics

So far, we've talked about deploying Prometheus with Kubernetes and how Prometheus pulls metrics, but which metrics are important? To say there is a large number of metrics to choose from in Kubernetes is an understatement. There are 212 individual metrics coming from the API server. There are 194 metrics from the kube-state-metrics project. There are also metrics from the kubelet and etcd. Instead of focusing on specific metrics, which will vary based on your projects, I would instead point you to the built-in Grafana that comes with the charts we deployed.

To access Grafana, go to <https://grafana.apps.X-X-X-X.nip.io/>, where X-X-X-X is your server's IP address but with dashes instead of dots. Since my cluster is on 192.168.2.82, I go to <https://grafana.apps.192-168-2-82.nip.io/>. The username is `admin` and the password is `prom-operator`. You can navigate through any of the available dashboards and click to edit them to see how they get their data. For instance, in *Figure 15.4*, I've navigated to the compute resources in the cluster, which breaks down CPU usage by namespace.



These dashboards were all installed as part of the Helm chart we deployed. We'll cover how to create your own dashboards later in the chapter.

From here, I can click on the menu and the edit option:

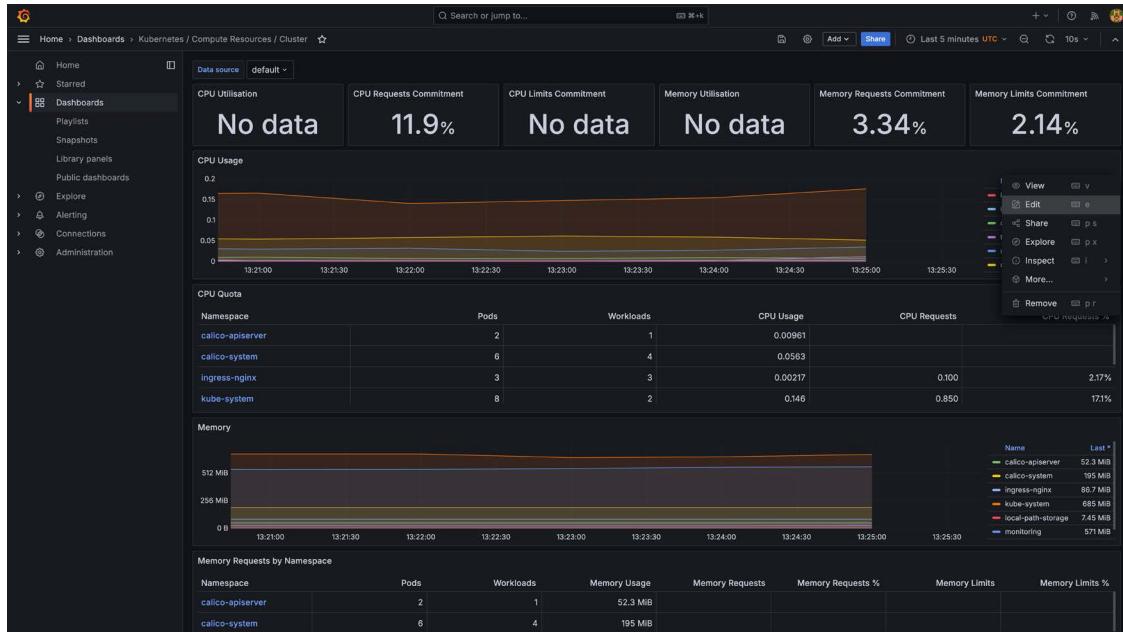


Figure 15.4: Grafana compute resources for the cluster

With the graph editor open, you can now view the PromQL expression used to generate the data:

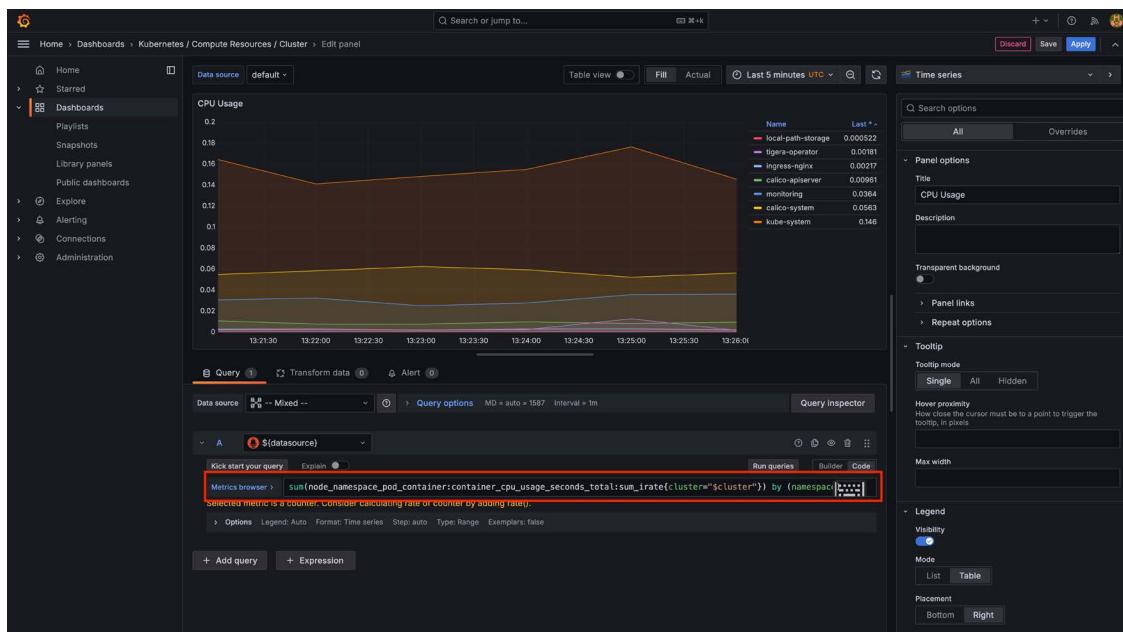


Figure 15.5: Edit screen in Grafana

If you take this expression, you can drop it into the **Graph** screen in Prometheus and see the raw data used to generate the graph:



Figure 15.6: Prometheus with a Grafana query

If you look closely at the query, you'll see that the Prometheus version doesn't reference a cluster. That's because the Grafana dashboards were built with the idea of managing multiple clusters, whereas Prometheus was set up to inspect only a single cluster. The data isn't annotated with the `cluster` attribute so Prometheus can't search on it. We'll talk more about this when we get to Grafana.

Now that we know where we can find examples of important metrics for our cluster and how to test them, we should spend some time on PromQL, the query language for Grafana.

## Querying Prometheus with PromQL

The majority of this chapter so far has been focused on deploying Prometheus and gathering data. We started to dive into how to query data, but we haven't yet dived into the details of the Prometheus query language, called PromQL. If you're familiar with other query languages, this won't look too different.

At a high level, the query language looks similar to the data. You start with a metric and which annotations you want to apply. For instance, looking at the compute query by namespace, first, let's look at what happens when we start with `node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate`:



Figure 15.7: CPU metrics

Since we didn't include any annotations in our query, we received the CPU usage for every pod on the cluster. If we wanted to see the CPU used in a specific namespace, we would specify that the same way it's specified in the metrics data but adding a `{annotation="value"}` to our metric. To see all of the containers' CPU usage in the monitoring namespace, add `{namespace="monitoring"}` to your query:

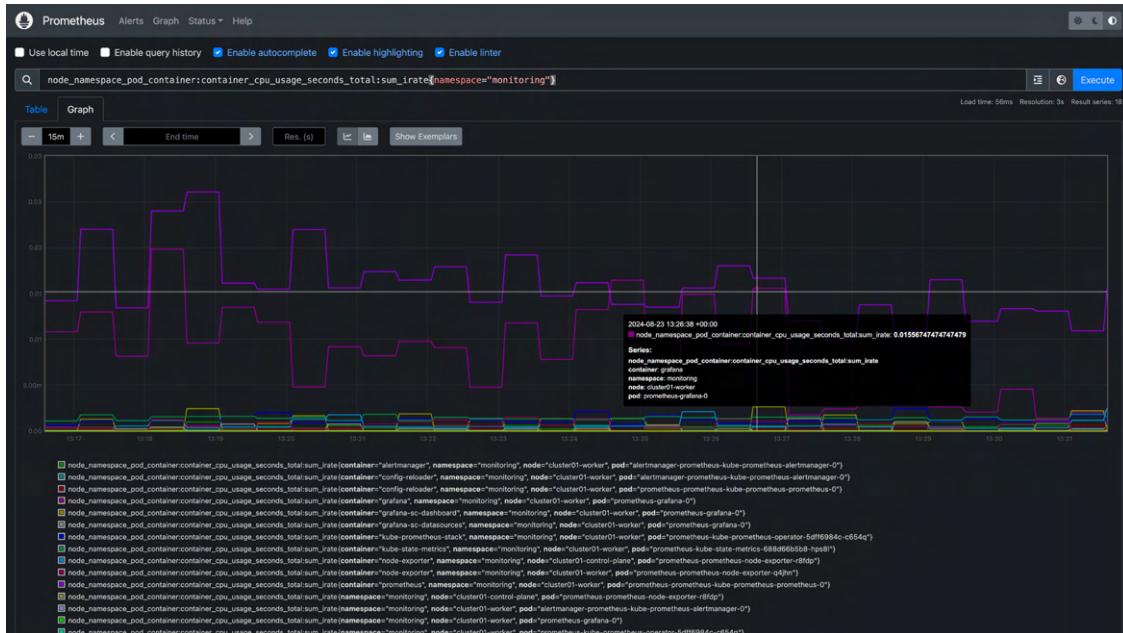


Figure 15.8: CPU in the monitoring namespace

Once you've limited the data you want, you may wish to aggregate the data. The current details show all of the running containers in the monitoring namespace, but that doesn't give you a great idea as to how much total CPU is being used. You can add functions that will aggregate for you, such as the `sum` function:



Figure 15.9: Sum of all CPU usage in the monitoring namespace

Finally, you may want to `sum` by a specific annotation, such as the pod since most of the pods have multiple containers. You can add a grouping using the `by` keyword:



Figure 15.10: CPU usage of pods in the monitoring namespace

In addition to functions, you can perform math operations, too. Let's say you want to know what percentage of total CPU has been used in your cluster. You need to know the CPU utilization at any moment, and the total amount of CPU available. We already know how to get the total CPU being utilized by all the containers in our cluster. Next, we need to know the total available CPU across the cluster. Then, we need to do some math to get the percentage. When doing math with PromQL, you would use the typical infix notation that you would use in most other programming and query languages. For instance, the `(sum(node_namespace_pod_container:container_cpu_usage_seconds_total:sum_irate) / max(count without(cpu,mode,pod) (node_cpu_seconds_total{mode="idle"}))) * 100` query combines multiple metrics and calculation to get the CPU utilization across the cluster:

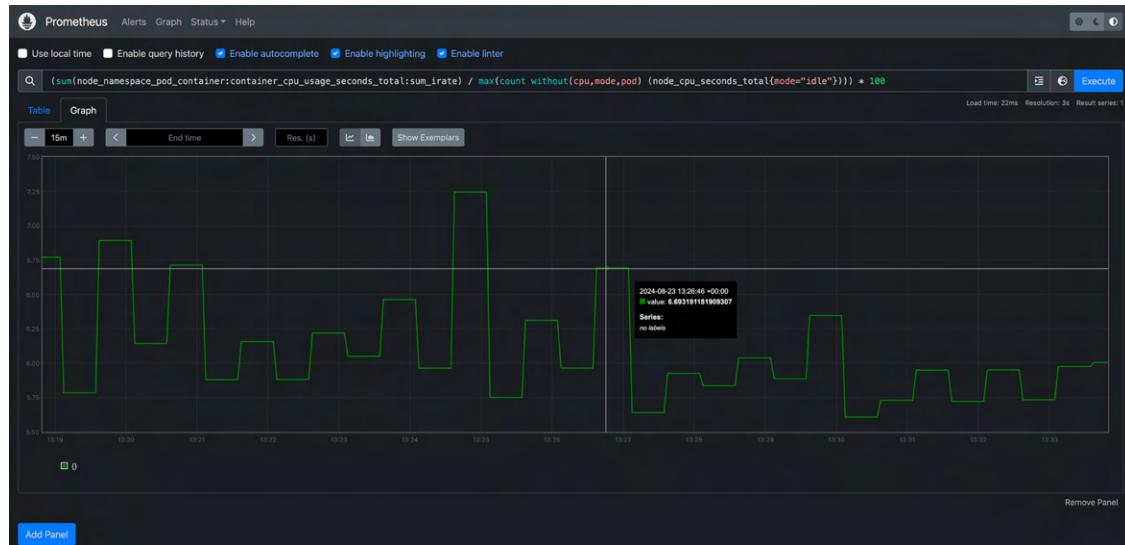


Figure 15.11: Percentage of CPU used across the cluster

Now, we have a way of knowing how much CPU we're using in the cluster. We could incorporate this knowledge into our capacity planning by creating an alert that tells us that our cluster has reached a certain capacity level. This leads us to our next section, which will focus on this very question.

## Alerting with Alertmanager

Thus far, we've deployed Prometheus, integrated it with our Kubernetes cluster, and learned how to query the database for useful information about our cluster. We've seen that Prometheus tracks alerts on the **Alerts** screen of the UI, but how do cluster operators get notified there's an issue?

The Alertmanager project (<https://prometheus.io/docs/alerting/latest/alertmanager/>) is a generic tool that knows how to query for alerts and then send them to the correct people. It's not simply a notification conduit; it also helps with deduplication and grouping, too. Finally, it provides an interface for silencing alerts that don't need to continue firing.

The helm charts we deployed earlier include an instance of Alertmanager and an Ingress for it too. Like with the other projects, you can access it with the `https://alertmanager.apps.X-X-X-X.nip.io/` with the X-X-X-X URL replaced with your cluster's IP address. Since my cluster is on 192.168.2.82, my URL is `https://alertmanager.apps.192-168-2-82.nip.io/`.

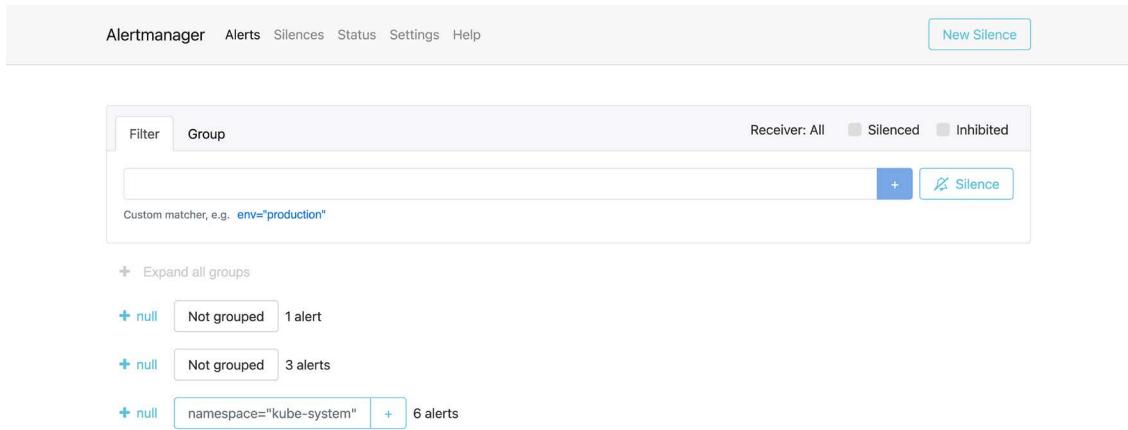


Figure 15.12: Alertmanager UI

Similar to Prometheus, you'll notice there's no authentication because, just like Prometheus, there's no security model. There is more information on that later in the chapter. What you will see is that there are already alerts! That's because running Kubernetes in KinD will lead to some interesting networking issues that aren't expected. If you run the Prometheus stack in a cloud-hosted Kubernetes, you will find similar results.

You'll notice that there are multiple sets of alerts. Alertmanager provides for tagging of alerts so you can better organize them. For instance, you may want to only send alerts for critical issues or route alerts based on where they're from.

You can silence an alert from this UI as well. This is useful when you want to stop notifications because you're working on the problem or because you know of the issue and it's an issue another team needs to address and you don't need to get continuous alerts while that team is addressing the problem. Many teams do not get direct access to this UI because of the lack of security, but we'll cover how to fix that later in the chapter.

While the UI lets you see what alerts there are and silence them, what it won't do is allow you to configure alerts or where to send them. That's done in custom resource objects, which we'll cover in the next section.

## How Do You Know Whether Something Is Broken?

So far, we've seen how to access the Alertmanager UI and we've seen that the alerts are configured in Prometheus, but we've not yet configured an alert. There are two steps to configuring an alert:

1. Create an instance of a `PrometheusRule` to define under what conditions an alert should be generated. This involves creating a PromQL expression to define the data, how long you want the condition to be met, and finally, how to label the alert.
2. Create an `AlertmanagerConfig` object to group and route the alert to a receiver.

We already have plenty of `PrometheusRule` objects thanks to the great set of pre-configured rules that come with the chart we deployed. The next question is how to build an `AlertmanagerConfig`. The tricky part about this is that we need something to send the alerts to. There are plenty of options including email, Slack, and various notification SaaS services. To keep things simple, let's deploy an NGINX server that can act as a webhook that will let us see the JSON payload. Our pagers won't go off, but it will at least give us a feel for what we're seeing. From inside the source repo:

```
$ kubectl apply -f chapter15/alertmanager-webhook/alertmanager-webhook.yaml
```

This will launch an NGINX pod in the `alert-manager-webhook` namespace. Now, let's configure an `AlertmanagerConfig` to send all critical alerts to our webhook:

```
apiVersion: monitoring.coreos.com/v1alpha1
kind: AlertmanagerConfig
metadata:
  name: critical-alerts
  namespace: kube-system
  labels:
    alertmanagerConfig: critical
spec:
  receivers:
    - name: nginx-webhook
      webhookConfigs:
        - sendResolved: true
          url: http://nginx-alerts.alert-manager-webhook.svc/webhook
  route:
    repeatInterval: 30s
    receiver: 'nginx-webhook'
    matchers:
      - name: severity
        matchType: "="
        value: critical
    groupBy: ['namespace']
    groupWait: 30s
    groupInterval: 5m
```

The `receivers` section tells Alertmanager to send all events to our web server. The `route.matchers` section tells Alertmanager which alerts to send. In our example, we will send any alerts with a severity of `critical` being generated from the `kube-system` namespace. When working with `AlertmanagerConfig` objects, the namespace the object is created in automatically gets added to your matchers. You can create this object from `chapter15/alertmanager-webhook/critical-alerts.yaml`. Once created, wait a few minutes. There will eventually be an alert that gets fired from Prometheus, which will result in a log entry like:

```
0.240.189.139 - - [12/Jan/2024:22:15:22 +0000] "POST /webhook HTTP/1.1"
body:"{\\"x22receiver\\x22:\\x22kube-system/critical-alerts/nginx-we... \" 200 2 \"-
"Alertmanager/0.26.0" "-"
```

The JSON in the log message is too large to provide here, but it provides all the information available to Alertmanager. There are very few times when you should be writing your own receiver. There are so many pre-built ones that it's unlikely you'll need to build your own.

Now that we know how to configure Alertmanager to send an alert, next, we'll walk through how to design metrics-based alerts.

## Alerting Your Team Based on Metrics

In the previous section, we walked through how to send an alert to a receiver using Alertmanager. Next, we'll walk through how to generate an alert. Alerts are not configured in Alertmanager but in Prometheus. The only job Alertmanager has is to forward the generated alerts to receivers. The job of determining whether an alert should be fired is up to Prometheus.

The `PrometheusRule` object is used to configure Prometheus to fire an alert. This object defines metadata for the rule, conditions for when the rule will fire, and how often the rule needs to fire for the alert to be sent to Alertmanager. The `kube-prometheus` project that we deployed comes with about forty pre-built rules. These rules are constantly being updated based on experience and you shouldn't update them on your own. You can, however, build your own rules for your own infrastructure.

To demonstrate this, let's deploy OpenUnison into our cluster:

```
$ cd chapter15/user-auth
$ ./deploy_openunison_imp_ impersonation.sh
```

We'll get into the details of what this script does when we get to the *Monitoring Applications* section later in the chapter. For now, know that this script deploys OpenUnison and integrates it with our `kube-prometheus` chart for both adding the login to our apps and providing something to monitor.

Now that we're using OpenUnison to provide authentication for our cluster, if it goes down, you want to know before your users start calling. We deployed the below `PrometheusRule` as part of our deployment script earlier:

```
---
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
  creationTimestamp: null
  labels:
    release: prometheus
  name: openunison-has-activesessions
spec:
  groups:
    - name: openunison.rules
```

```

rules:
- alert: no-sessions
  annotations:
    description: Fires when there are no OpenUnison sessions
    expr: absent(active_sessions)
    for: 1m
  labels:
    severity: openunison-critical
    source: openunison

```

In our PrometheusRule, we created a single group with a single rule. The rule creates an alert called no-sessions that checks for the absence of the active\_sessions metric. This metric is provided by OpenUnison to track how many sessions are currently open. If we simply had something like active\_sessions < 1, then this rule wouldn't fire because there is no active\_sessions metric. The language for specifying the expr is the same PromQL that we used with Prometheus to query our data. This means you can test your expressions in the Prometheus web app before creating your PrometheusRule objects.

Let's fire this rule by deleting the metrics Application object from OpenUnison:

```
$ kubectl delete application metrics -n openunison
```

After about thirty seconds, if we log in to Prometheus and click on the **Alerts** link, we'll see:

Labels	State	Active Since	Value
openunison-session,severity=openunison-critical,source=openunison	PENDING	2024-08-23T13:42:59.510818859Z	1

Figure 15.13: Pending alert in Prometheus

The screenshot shows that there's a pending alert. This is happening because, in our rule, we said that the conditions of the rule must be met for at least one minute. This is an important tuning option to help prevent false positives. Depending on what you're monitoring, you could find you're getting alerts that are being cleared very quickly on their own. After another thirty seconds or so, you'll see that the alert has moved from pending to firing:

The screenshot displays the Prometheus Alerting interface. At the top, there are three filter buttons: 'Inactive (589)', 'Pending (0)', and 'Firing (1)'. A search bar labeled 'Filter by name or labels' is present. Below the filters, a pink header bar indicates the alert is from the file '/etc/prometheus/rules/prometheus-prometheus-kube-prometheus-prometheus-rulesfiles-0/default-openunison-has-activesessions-cf7cfa94-866e-4d7f-913c-84c7d20c32c8.yaml' and is labeled 'openunison.rules'. The alert itself is titled 'no-sessions' and is described as having 1 active entry. It specifies a metric name 'active\_sessions' with a threshold 'absent', a duration 'for: 1m', and labels 'severity: openunison-critical', 'source: openunison', and annotations 'description: Fires when there are no OpenUnison sessions'. The alert is currently in a 'FIRING' state, active since 2024-08-23T13:42:59.510818859Z, with a value of 1. The bottom section lists other inactive alerts under 'AlertmanagerFailedReload' and 'ConfigReloaderSidecarErrors'.

Figure 15.14: Prometheus alert firing

Now that our rule is firing, we can look in the Alertmanager application and see an alert is firing:

The screenshot shows the Alertmanager UI with a single firing alert. The alert details are as follows: alertname = "no-sessions", prometheus = "monitoring/prometheus-kube-prometheus-prometheus", severity = "openunison-critical", and source = "openunison". There are also '+' buttons next to each label entry.

Figure 15.15: Alert firing in Alertmanager

We don't have anything to collect the alert, but if we did, we'd now be receiving alerts that OpenUnison is down! Let's fix the problem by re-adding the monitoring application:

```
$ helm upgrade orchestra-login-portal tremolo/orchestra-login-portal -n
openunison -f /tmp/openunison-values.yaml
```

Once this command is done, the same process will go in reverse. The first time that OpenUnison responds with the active\_sessions metric, the alert will move into a pending status. If everything is OK after a full minute, the alert will be cleared.



You may be asking why we simply deleted the metrics application instead of stopping OpenUnison. The Deployment script added security to our infrastructure, which would have made it harder to access the Prometheus and Alertmanager applications without OpenUnison. While you could have used port forwarding to access both Prometheus and Alertmanager, that could be complicated based on how your cluster is deployed, so we went with a simpler approach.

Now that we know how to generate an alert, what happens if we want to ignore it? We'll cover that in the next section.

## Silencing Alerts

Now that we know how to generate an alert, how do we silence it? There are many reasons why you'd want to silence an alert:

- **Known outage:** You've been informed that ongoing work will cause an outage and there's no reason to act on the alerts.
- **Outage outside your control:** Your outage is caused by a system outside of your control. For instance, if there's an issue with your Active Directory that you have no control over and OpenUnison fails to authenticate because of it, you shouldn't be getting alerts.
- **Ongoing outage:** You know there's an issue; the alert doesn't need to keep firing.

You can enable a silence based on labels provided by your alerts. When you see an alert you want to silence, you can click on the **Silence** button in the Alertmanager application:

The screenshot shows the Alertmanager interface with the following details:

- Header:** Alertmanager, Alerts, Silences, Status, Settings, Help, New Silence (button).
- Filter/Group:** Filter (selected), Group.
- Receiver:** All, Silenced, Inhibited.
- Search Bar:** namespace="kube-system" (with a clear button), +, Silence (button).
- Custom matcher:** Custom matcher, e.g. env="production".
- Alert Details:**
  - Alert Count:** 1 alert.
  - Timestamp:** 2024-08-23T13:42:09.117Z.
  - Labels:** alertname="etcdInsufficientMembers", endpoint="http-metrics", job="kube-etcd", pod="etcd-cluster01-control-plane", prometheus="monitoring/prometheus-kube-prometheus-prometheus", service="prometheus-kube-prometheus-kube-etcd", severity="critical".

Figure 15.16: Create a silence in Alertmanager from an alert

You can now customize the alert, specify who created it, and how long it should last. This silence isn't persisted in the API server as an object, so you can't scan for it via the Kubernetes API (though that would be a great feature).



Security-minded readers may be thinking, could an attacker create a silence to cover their tracks? Of course! You could silence warnings about CPU while running Bitcoin miners, for example. We'll talk more about the security of Prometheus when we get to adding SSO to the monitoring stack at the end of the chapter.

We've worked through most of the operational portions of our monitoring stack. The next step is to visualize all of the data collected. We'll cover that next by looking at Grafana.

## Visualizing Data with Grafana

So far, we've worked with the data collected by Prometheus in an operational way. We've focused on how to react to changes in data in a way that impacts our cluster and our users. While it's great to be able to act on this data, there's too much to be able to process without some help. This is where Grafana comes in; it provides a way for us to build dashboards based on the data from Prometheus (as well as other sources). We already looked at some of the out-of-the-box graphs earlier in the chapter. Now, we'll create our own graphs and integrate those graphs into the kube-prometheus stack we've deployed.

### Creating Your Own Graphs

A graph is a combination of a dataset and a set of visualization rules. The graph itself is defined by JSON. This means that it can be persisted as a Kubernetes object and loaded as part of our stack instead of being stored in a persisted database. The downside to this approach is that you'll need to first generate that JSON. Thankfully, the Grafana web UI makes it easy to do:

1. Log in to Grafana.
2. Create a new dashboard: We created a simple dashboard for OpenUnison's `active_sessions` metric.
3. Once you've created the dashboard, export it to JSON.
4. Create a ConfigMap with the label `grafana_dashboard="1"`.
5. Here are the important parts of `chapter15/user-auth/grafana-custom-dashboard.yaml`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  labels:
    grafana_dashboard: "1"
  name: openunison-activesessions-dashboard-configmap
  namespace: monitoring
data:
  openunison-activesessions-dashboard.json: |-
  {
    "annotations": {
      .
    }
  }
```

Once the ConfigMap is created, Grafana will pick it up almost immediately!

Having created a dashboard, you may have noticed that there are other capabilities in Grafana, such as alerting. Grafana can be used for this process, but that's outside of the scope of the kube-prometheus project and this book.

Now that you are familiar with the various components of the kube-prometheus stack, the next step is to look at how you can use it to monitor applications and systems running in your cluster.

## Monitoring Applications

In the previous sections of this chapter, we focused on working with the operational aspects of the kube-prometheus stack for monitoring and alerting. We integrated OpenUnison into our cluster and created monitors and alerts, but we didn't detail how this worked. We're going to use OpenUnison as a model for integrating other systems into your monitoring stack.

### Why You Should Add Metrics to Your Applications

Before we move forward with how we added metrics and monitoring to OpenUnison, the first question we should answer is why. Your clusters are made of more than just your Kubernetes implementation. Most clusters today have automation frameworks, authentication systems, external integrations, GitOps frameworks, and so on. If any of these components go down, to your users, your cluster is down. From a customer-management perspective, you want to know before they start opening alerts.

In addition to your systems, you may be dependent on outside systems. When these go down, and they impact you and your customers, your customers will come to you first.

This is very true in the authentication world, where, if the login process doesn't "complete," the authentication process is assumed to be the problem. I have dozens of anecdotes to demonstrate this reality, but I'll focus on a couple where downstream monitoring helped me identify the root cause and stay ahead of my customers' tickets. First off, many of my customers use OpenUnison to integrate with Active Directory via LDAP. While Active Directory is a very solid system, the network access is susceptible to issues. An errant firewall rule can cut off access, and adding monitoring of OpenUnison's downstream Active Directory has provided quick evidence that an outage of the login process isn't an OpenUnison issue.

The Prometheus format for metrics has become a de facto standard in the cloud-native world. Even systems that aren't built on Prometheus have built-in support for it, such as commercial systems like Datadog and Amazon CloudWatch. This means that most monitoring systems you'll have deployed have support for Prometheus metric endpoints, even if you're not using Prometheus internally. For systems that aren't web-based, there are often "bolt-on" solutions for monitoring via Prometheus, such as databases.

Having discussed why you should be monitoring your cluster systems, not just Kubernetes, let's step through how we're monitoring our OpenUnison.

## Adding Metrics to OpenUnison

Earlier in the chapter, we redeployed our monitoring stack with an OpenUnison instance. Now, it's time to walk through what that integration looks like. If you haven't already, redeploy your monitoring stack and OpenUnison:

```
$ cd chapter15/user-auth  
$ ./deploy_openunison_imp_ impersonation.sh
```

Prometheus' operator looks for various objects for things to monitor; we're going to focus on the `ServiceMonitor`. If you look in the `monitoring` namespace, you'll notice a dozen or so predefined `ServiceMonitor` objects. The point of a `ServiceMonitor` is to tell Prometheus to look up which pods to monitor based on a `Service` object. This makes sense as a cloud-native pattern, you wouldn't want to hardcode your metrics endpoints. pods get rescheduled, scaled, and so on. Relying on a `Service` object helps Prometheus scale in a cloud-native way. For OpenUnison, here's our `ServiceMonitor` object:

```
---  
apiVersion: monitoring.coreos.com/v1  
kind: ServiceMonitor  
metadata:  
  labels:  
    release: prometheus  
  name: orchestra  
  namespace: monitoring  
spec:  
  endpoints:  
    - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token  
      interval: 30s  
      port: openunison-secure-orchestra  
      scheme: https  
      targetPort: 8443  
      tlsConfig:  
        insecureSkipVerify: true  
  namespaceSelector:  
    matchNames:  
    - openunison  
  selector:  
    matchLabels:  
      app: openunison-orchestra
```

The first thing to point out is that there's a label called `release="prometheus"`. This label is needed for `kube-prometheus` to pick up our monitor. Prometheus is not a multitenant system, so it's reasonable to expect there to be multiple instances for different use cases. Requiring this label makes sure that the `ServiceMonitor` object is picked up by the correct Prometheus operator deployment.

Next, we'll point out that the endpoint lines up with the `openunison-orchestra` Service in the `openunison` namespace. We didn't name it directly, but we did identify it by labels. It's important to make sure you don't get multiple Service objects integrated by having overly broad labels. Finally, we included the `bearerTokenFile` option to tell Prometheus to use its own identity when accessing OpenUnison's metrics endpoint. We'll cover this in more detail in the next section.

If we deployed just this object, the Prometheus operator would complain that it can't load the correct Service objects because it doesn't have RBAC permissions. The next step is to create an RBAC Role and RoleBinding for the operator to be able to look up the Services:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: monitoring-list-services
  namespace: openunison
rules:
- apiGroups:
  - ""
  resources:
  - endpoints
  - pods
  - services
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: monitoring-list-services
  namespace: openunison
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: monitoring-list-services
subjects:
- kind: ServiceAccount
```

```
name: prometheus-kube-prometheus-prometheus
namespace: openunison
```

This should look straightforward if you've already read through our chapter on Kubernetes RBAC. We included services, endpoints, and pods because once you retrieve a Service, you use an Endpoint object to get to a pod that has the correct IP. Each /metrics endpoint is then accessed based on the Pod's IP address, not the Service host. This means you'll need to keep in mind that, if your system uses hostnames for routing, you'll need to accept the /metrics on all hostnames.

Once you have Prometheus configured, you'll start seeing your metrics in a few minutes. If they don't show up, there are three places to look:

- **Prometheus operator:** The operator will show whether there were any issues loading the Service/Endpoint/Pod.
- **Prometheus Pod, configuration reloader container:** The Prometheus pod has a sidecar container that reloads the configuration. Check here next to see whether there was an issue loading the configuration.
- **Prometheus Pod, Prometheus container:** Finally, check the Prometheus container in the Prometheus pod to see whether there is an issue in Prometheus loading the metrics.

Having walked through how to set up a monitor in Prometheus, the next question is why, and how, to secure your metrics endpoints.

## Securing Access to the Metrics Endpoint

Throughout this chapter, we've only tangentially mentioned security. That's because, for the most part, Prometheus follows the **SNMP approach: Security is Not My Problem**. There are some good reasons for this. If you're using your Prometheus stack to debug an outage, you don't want security to break that process. At the same time, making all of the data that can be gleaned from metrics for an attacker publicly visible is dangerous. In their 2019 keynote at KubeCon North America, Ian Coldwater said, "Attackers think in graphs" (Ian is quoting John Lambert: <https://github.com/JohnLaTWC/Shared/blob/master/Defenders%20think%20in%20lists.%20Attackers%20think%20in%20graphs.%20As%20long%20as%20this%20is%20true%2C%20attackers%20win.md>), which comes to mind as I think about this because you can map out an environment based on metrics endpoints! Think about all the data about workloads and distributions, when and where nodes work the hardest, and so on. Take the active\_sessions metric we worked with earlier in the chapter. Simply mapping that number over time will tell you when a spike in usage may not trigger alarms because it's within norms.

The good news is that because Prometheus runs in clusters, it gets its own identity. That's why our ServiceMonitor included the bearerTokenFile option to the Pod's built-in Kubernetes identity. OpenUnison validates this identity against the API server using a SubjectAccessReview. That's why, when you look at the OpenUnison logs, you'll see something like:

```
[2024-01-16 03:18:22,254][XNIO-1 task-4] INFO AccessLog - [AuSuccess] -
metrics - https://10.240.189.139:8443/metrics - username=system:serviceaccount:monitoring:prometheus-kube-prometheus-prometheus,ou=oauth2,o=Tremolo - 20 /
oauth2k8s [10.240.189.165] - [f763bbd1a1c474929d91bfe89a2fd8e5f5b49a1d5]
```

```
[2024-01-16 03:18:22,254][XNIO-1 task-4] INFO AccessLog - [AzSuccess] -  
metrics - https://10.240.189.139:8443/metrics - username=system:serviceaccoun  
t:monitoring:prometheus-kube-prometheus-prometheus,ou=oAuth2,o=Tremolo - NONE  
[10.240.189.165] - [f763bbd1a1c474929d91bfe89a2fd8e5f5b49a1d5]
```

Whenever Prometheus attempts to scrape the metrics from OpenUnison, we know it's with a token that is bound to a running pod and is still valid. When evaluating systems that provide metrics, check to see whether they support some kind of token validation. It's not a bad idea to use NetworkPolicies to limit access, too, but as we've discussed several times, you'll get the best protection based on a Pod's identity.

Having reviewed how to secure your application metrics, the last section on Prometheus will focus on adding security to the kube-prometheus stack.

## Securing Access to Your Monitoring Stack

The kube-prometheus stack is a combination of Prometheus, Alertmanager, and Grafana combined with operators to automate the deployment and management of the stack. When we went through each of the applications in the stack, we pointed out that neither Prometheus nor Alertmanager has any sense of what a user is. Grafana does have its own user model, but kube-prometheus ships with a hardcoded credential. It's assumed that you'll access these tools via the `kubectl port-forward` directive. This is a similar scenario to the Kubernetes dashboard that we secured earlier in the book. While none of these applications use the user's identity to communicate with the API server, they can be abused to provide extensive knowledge about the environment, so usage should be tracked.

For Prometheus and Alertmanager, the easiest approach is to place an authenticating reverse proxy in front of them, something like an OAuth2 proxy, for example. For this chapter, we used OpenUnison because it's a built-in capability and requires fewer things to deploy.

Grafana is more complicated because it does have several options for authentication. It also has its own user authorization model based on teams and roles. The Grafana that ships with the kube-prometheus charts is the Community edition, which only supports two roles: **Admin** and **Viewers**. While Grafana does support OpenID Connect out of the box, that would involve a more complicated helm configuration, and since we're already using OpenUnison's reverse proxy to authenticate Prometheus and Alertmanager, we went with the same approach with Grafana. The user's identity is injected via an HTTP header in the request from OpenUnison to Grafana, with all users being considered administrators. Then, Grafana is configured in the helm chart using the proxy authentication method. So, where originally, we had Ingress objects that pointed directly to our applications, now we have a single Ingress pointing to OpenUnison, which is responsible for authenticating and authorizing access to these applications:

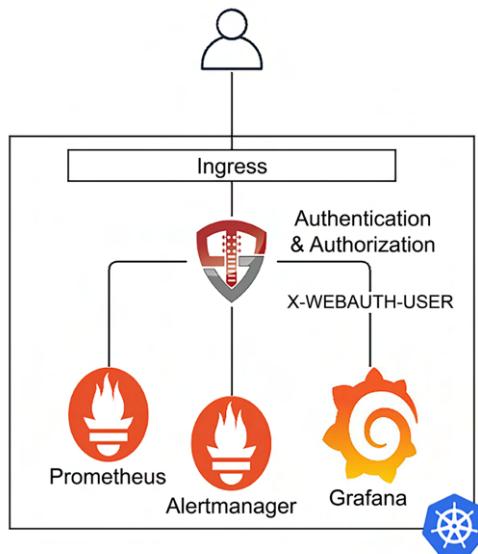


Figure 15.17: Adding SSO to kube-prometheus

A bonus to integrating the kube-prometheus stack into OpenUnison is that you don't need to memorize the URLs because they're included as badges along with the dashboard and tokens:

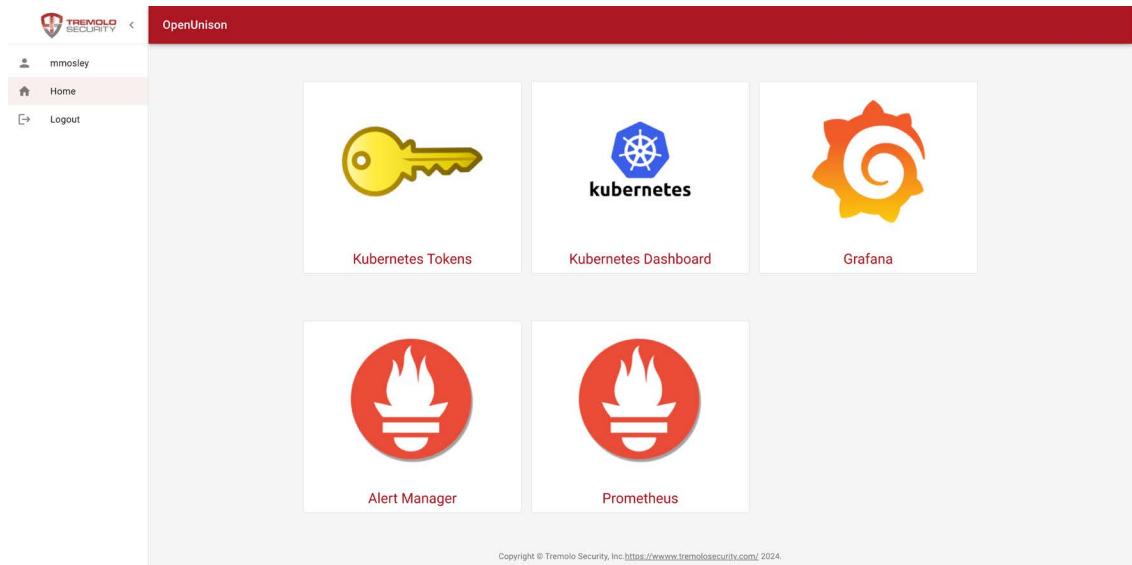


Figure 15.18: OpenUnison with kube-prometheus “badges”

What happens if OpenUnison goes down? It's important to always have a "break glass in case of emergency" plan! You can still fall back on port forwarding access to all three of the applications.

This concludes our section on monitoring Kubernetes using Prometheus. Next, we'll explore how logs work in Kubernetes and how to manage them.

## Log Management in Kubernetes

Throughout this book, after running an exercise, we will often ask you to view the logs of a container by running a command such as:

```
kubectl logs mypod -n myns
```

This allows us to view logs, but what's happening to get the logs? Where are the logs stored and how are they managed? What are the processes to manage archiving logs? It turns out this is a complex topic that often gets overlooked when getting started with Kubernetes. The rest of this chapter will be dedicated to answering these questions. First, let's discuss how Kubernetes stores logs, and then we'll get into pulling those logs into a centralized system.

## Understanding Container Logs

Before we ran containers, logging was relatively straightforward. Your application usually had a library that was responsible for sending data to logs. That library would rotate the logs and often clean out old logs. It wasn't unusual to have multiple logs for multiple purposes. For instance, most web servers had at least two logs. There was an access log to record who made requests to the web server and an error log to track any error or debug messages. In the early 2000s, companies like Splunk came out with systems that would ingest your logs into a time series database that you could use to query them in real time across multiple systems, making log management even easier.

Then came Docker containers, which broke this model. Containers are self-contained and not meant to generate data on their own. Instead of generating log data to a volume, containers were encouraged to pipe all log data to standard out so that the Docker API could be used to watch the logs without directly accessing the volume they were stored on. This standard continued with Kubernetes, so that as an operator, I never need access to the file where logs are stored, just access to the Kubernetes API. While this vastly simplifies accessing a log directly, it makes managing logs much more difficult. First off, an application can no longer break up logs by function, so as an operator, I need to filter out the parts of the logs I want. Additionally, the logs are no longer rotated in a way that is standard or configurable by the application owner. Finally, how do you archive logs in a way that satisfies your compliance requirements? The answer is to pipe your logs into a central log management system. Next, we'll walk through the OpenSearch project, which is the log management system we chose to illustrate how container log management works.

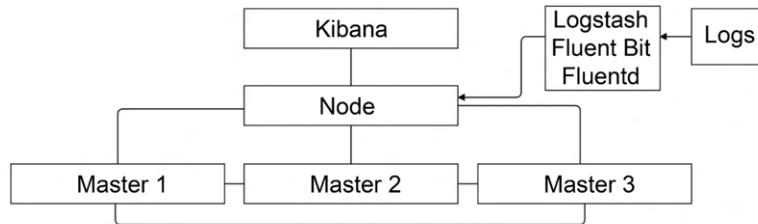
## Introducing OpenSearch

There are several log management systems. Splunk is often the most well known, but there is an entire industry built around log management. There are multiple open source log management systems as well. Probably the best known is the “ELK” stack, which is a combination of:

- **Elasticsearch:** A time series database and indexing system for storing and sorting logs

- **Logstash:** A project for getting logs into Elasticsearch
- **Kibana:** A dashboard and UI for Elasticsearch

The ELK stack is not the only open source log management system. Another project called Graylog is popular as well. Unfortunately, both projects hide their SSO support for OpenID Connect in commercial offerings. In 2021, Amazon forked the ELK stack from Elasticsearch 7.0 into the OpenSearch project. The two projects have since diverged. We decided to focus on OpenSearch for this chapter because it's fully open source and we can show how it integrates into your cluster's enterprise requirements via OpenID Connect.



*Figure 15.19: OpenSearch architecture*

In the above diagram, we see the major components of an OpenSearch deployment:

- **Masters:** This is the engine of OpenSearch that indexes log data.
- **Node:** The nodes are the integration points for services interacting with OpenSearch. It hosts the API used to query indices and to push logs into the cluster.
- **Kibana:** OpenSearch has a bundled Kibana dashboard for interacting with the OpenSearch API via a web application.
- **Logstash/Fluent Bit/Fluentd:** A DaemonSet that tails the logs in the cluster and sends them to OpenSearch.

We're not going too deeply into how OpenSearch works, because it's a complex system that deserves its own book (and there are a few out there). We're going to go deep enough to see how it relates to our enterprise Kubernetes requirements for aggregating logs and meeting our enterprise security requirements using our centralized Active Directory and managing access via our directory groups. Now that we've got an overview of the different components of our OpenSearch cluster, next, we'll deploy it.

## Deploying OpenSearch

We've automated the deployment of OpenSearch using scripts. There's a dependency in OpenSearch for the CRDs from Prometheus' operator, so we're going to need that deployed, too. We're going to start with a fresh cluster:

```
$ cd chapter2
$ kind delete cluster -n cluster01
$ ./create-cluster.sh
.
.
```

```
.
$ cd ../chapter15/simple
$ ./deploy-prometheus-charts.sh
.
.
.
$ cd ../user-auth/
$ ./deploy_openunison_imp_imersonation.sh
```

These scripts:

1. Deploy a new KinD cluster with an NGINX Ingress controller.
2. Deploy the kube-prometheus project for Prometheus, Alertmanager, and Grafana.
3. Deploy “Active Directory” and OpenUnison, and integrate SSO into the Prometheus stack applications.

This is the same place where you would be had you followed throughout this chapter. Next, we’ll deploy OpenSearch:

```
$ cd ../opensearch/
$ ./deploy_opensearch.sh
```

This script does several things:

1. Increases file limits to support both OpenSearch and FluentBit opening every log and tailing them
2. Deploys the OpenSearch operator via Helm
3. Creates OpenUnison configuration objects to integrate OpenSearch
4. Deploys an OpenSearch cluster, configured to use SSO from OpenUnison using OpenID Connect
5. Deploys Fluent Bit via Helm

We’re not going to spend lots of time diving into individual configurations. Given how quickly things change, it would be better to get individual instructions directly from the OpenSearch project. We will instead walk through how these components relate to each other, our cluster, and our enterprise security requirements. Now that everything is deployed, we’re going to walk through how a log gets from your container into OpenSearch, and how you access it.

## Tracing Logs from Your Container to Your Console

With OpenSearch in place and integrated into your cluster, let’s follow how logs get from your `ingress-nginx` container into your console. The first place to look is in the `fluentbit` namespace, where you’ll find a single `DaemonSet` called `fluent-bit`. Recall that a `DaemonSet` is a pod that gets deployed to every node in your cluster. Since we only have one node in our KinD cluster, we only have one pod for the `fluent-bit` `DaemonSet`. This pod is responsible for scanning all of the logs on the node and tailing them, similar to how you might tail a log on a local file system. What’s important about Fluent Bit is that in addition to sending log data to OpenSearch, it’s also adding metadata, which will allow us to easily search for log data inside of OpenSearch.



You might be wondering why we're not using Logstash since it's one of the named components in the ELK stack. Logstash isn't the only log aggregator project. FluentD and FluentBit are both very popular tools for pulling logs from your cluster. FluentD is much heavier than Fluent Bit and has many more capabilities for transforming and parsing log data before sending it to OpenSearch. FluentBit is simpler but is also much smaller. We went with FluentBit for its simplicity and size given we're already filling out the cluster with other tools.

Let's look at the Pod's logs and look for `ingress-nginx`:

```
$ k logs fluent-bit-grhw -n fluentbit | grep nginx
[2024/01/26 01:26:23] [ info] [input:tail:tail.0] inotify_fs_add():
inode=1606570 watch_fd=19 name=/var/log/containers/ingress-nginx-admission-
create-k8fxz_ingress-nginx_create-6....log
[2024/01/26 01:26:23] [ info] [input:tail:tail.0] inotify_fs_add():
inode=1606592 watch_fd=20 name=/var/log/containers/ingress-nginx-admission-
patch-fhpx_nginx-ingress-nginx_patch-0....log
[2024/01/26 01:26:23] [ info] [input:tail:tail.0] inotify_fs_add():
inode=1610601 watch_fd=21 name=/var/log/containers/ingress-nginx-controller-
977d987f8-4xxvr_ingress-nginx_controller-8....log
```

As you can see, FluentBit found the logs on the node. You may be asking whether or not the FluentBit pods need special permissions to access the logs on the node, and the answer is yes! If we look at the `fluent-bit` DaemonSet, we'll see that the `securityContext` is empty, meaning there are no constraints on the pod, and that the `volumes` include `hostMount` directives to where the logs are stored on standard kubeadm deployments. These Pods are privileged and should be protected as such by limiting access to the `fluentbit` namespace and adding policies, such as with GateKeeper, that limit which containers can run in the `fluentbit` namespace, too.

Once the watch is placed on the `ingress-nginx` logs, those logs and additional metadata are sent to the OpenSearch node. As we discussed earlier, the OpenSearch node hosts the API and is the conduit for the masters, which are responsible for managing the indexes. The `fluent-bit` DaemonSet communicates with OpenSearch using the Logstash protocol and a simple authentication using basic authentication.



It would be great for our FluentBit deployment to use its `ServiceAccount` token to communicate with OpenSearch securely in the same way we configured our pods to communicate with Vault, but unfortunately, this feature doesn't exist in either the nodes or in FluentBit. Instead, you should make sure to give your Logstash account an extremely long password and make sure to rotate it with your enterprise policies. You could even leverage a secrets manager...

As the OpenSearch node pulls in the data, it's sent to the masters to be indexed. This is where the magic of OpenSearch happens because this is where all that data gets stored in an index and is made available to you and your cluster's administrators.

Now that the data is in OpenSearch, how are you going to access it? OpenSearch includes a Kibana dashboard for accessing and visualizing log data. The default implementation uses a single admin username and password, but that's not going to work for us! Log data is extremely sensitive, and we want to make sure we're using our enterprise security requirements when accessing it! That said, we'll want to integrate OpenSearch with OpenUnison the same way we've integrated the rest of our cluster management applications. Thankfully, OpenSearch supports OpenID Connect, which makes integration with OpenUnison very straightforward!



OpenSearch supports multiple authentication systems in addition to OpenID Connect, including LDAP. We could use this LDAP functionality to integrate with the “Active Directory” we deployed with OpenUnison. This integration provides some major limitations, though. If our enterprise decides to move off of Active Directory to an identity-as-a-service platform, such as Entra (formerly known as Azure AD) or Okta, this solution wouldn't work anymore. Also, if a multi-factor solution is added, this method would no longer work. Using OpenID Connect with an integration tool like OpenUnison, Dex, or Keycloak will make your deployment much more manageable.

What's interesting about the OpenSearch OpenID Connect implementation is that it is very similar to how the Kubernetes dashboard worked. The Kibana that is bundled with OpenSearch can use OpenID Connect to redirect the user to OpenUnison to authenticate and also knows how to refresh the user's `id_token` to keep the session open. Once authenticated, Kibana then uses the user's token to interact with the OpenSearch nodes using their identity. This means that, in addition to configuring Kibana, we need to configure the OpenSearch nodes to trust OpenUnison's tokens.

There are two configuration points to do this. In `chapter15/opensearch/opensearch-sso.yaml`, you'll find an OpenSearch cluster object with a `spec.dashboard.additionalConfig` that contains the dashboard (Kibana) configuration. If we deployed with just this, we'd be able to authenticate to Kibana, but we wouldn't be able to interact with OpenSearch because the API calls would fail.

Next, there's a Secret called `opensearch-security-config`, which contains a key called `config.yml` that stores the OpenSearch node security main configuration. Here, we tell OpenSearch where to retrieve the OpenUnison OpenID Connect discovery document so that the `id_token` sent by Kibana can be validated. Similar to the Kubernetes dashboard, when using OpenID Connect, the API isn't able to refresh or manage a user's session. The nodes are only validating the user's `id_token`.

We've tracked our data from our container's log into OpenSearch and seen how we will access it. Next, let's log in to Kibana and view our log data!

## Viewing Log Data in Kibana

We've spent quite a bit of time describing how OpenSearch is deployed and how log data is ingested into the OpenSearch cluster from Kubernetes. Next, we'll log in to Kibana and view the logs from our `ingress-nginx` Deployment.

First, open a web browser and enter the URL for your OpenUnison deployment. Just as in prior chapters, it will be `https://k8sou.apps.X-X-X-X.nip.io/`, where X-X-X-X is your cluster's IP address with dashes instead of dots. Since my cluster is running on 192.168.2.93, I navigate to `https://k8sou.apps.192-168-2-93.nip.io/`. Use the username `mmosley` and the password `start123` to log in. You'll now see an OpenSearch badge:

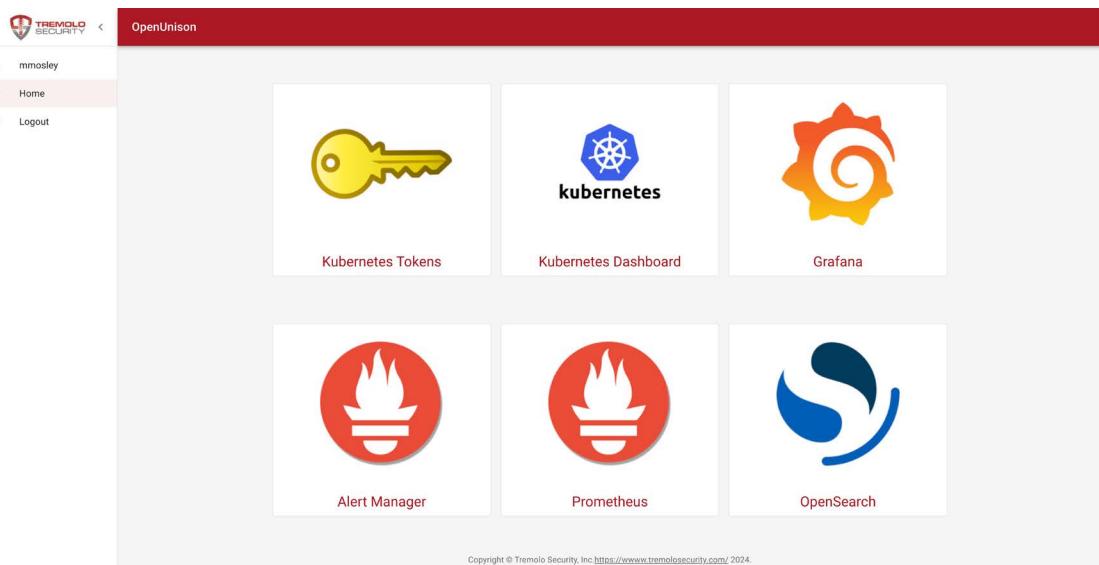


Figure 15.20: OpenUnison with OpenSearch “badge”

Click on the OpenSearch badge. You may be asked to add data, but skip this so we can get straight to our data. Next, click on the three horizontal bars in the upper-left corner to open the menu, scroll down to **Management**, and click on **Dashboards Management**:

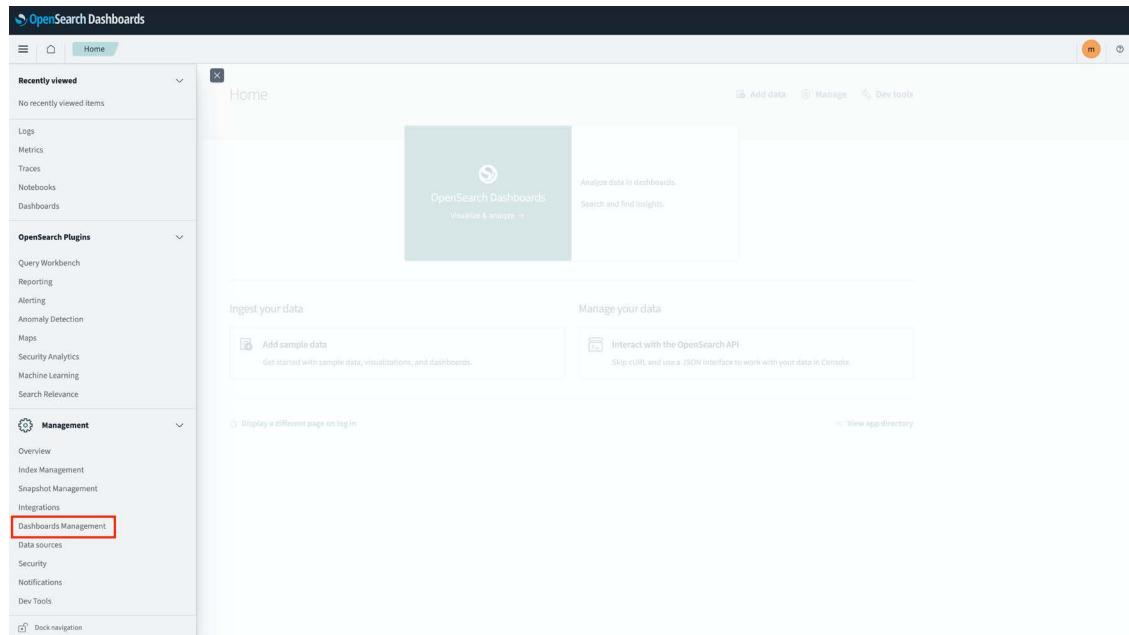


Figure 15.21: OpenSearch Dashboard Management menu

Next, click on **Index patterns** on the left and then **Create index pattern** on the right:

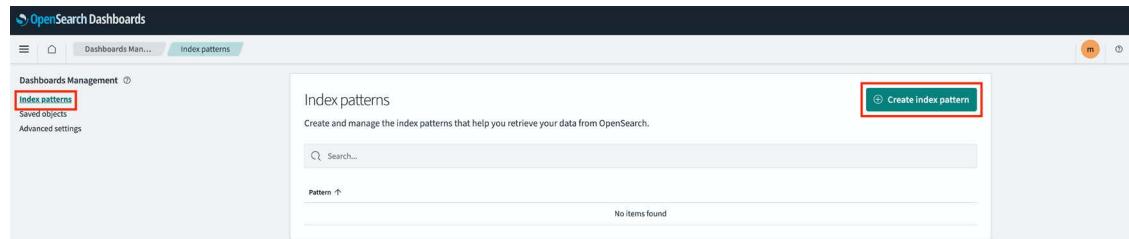


Figure 15.22: Index patterns

On the next screen, use `logstash-*` for the **Index pattern name** to load all our indices from FluentBit and click **Next step**.

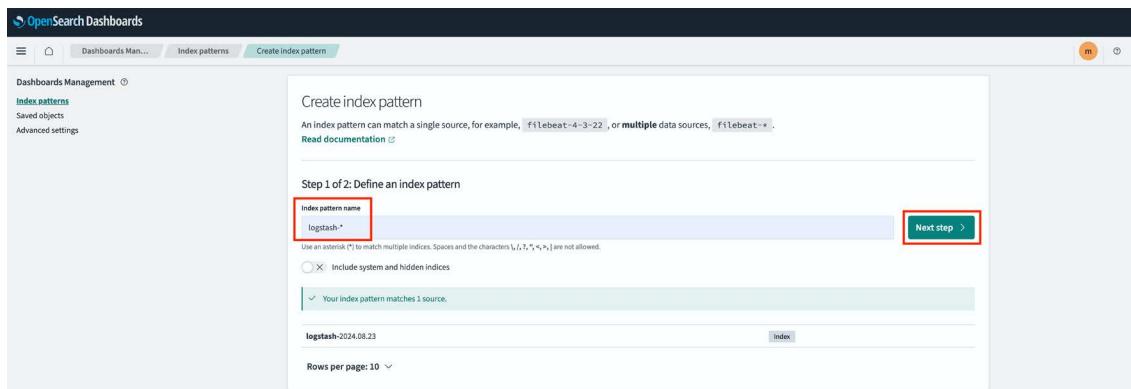


Figure 15.23: Create index pattern

On the next screen, choose `@timestamp` for the **Time field** and, finally, click **Create index pattern**:

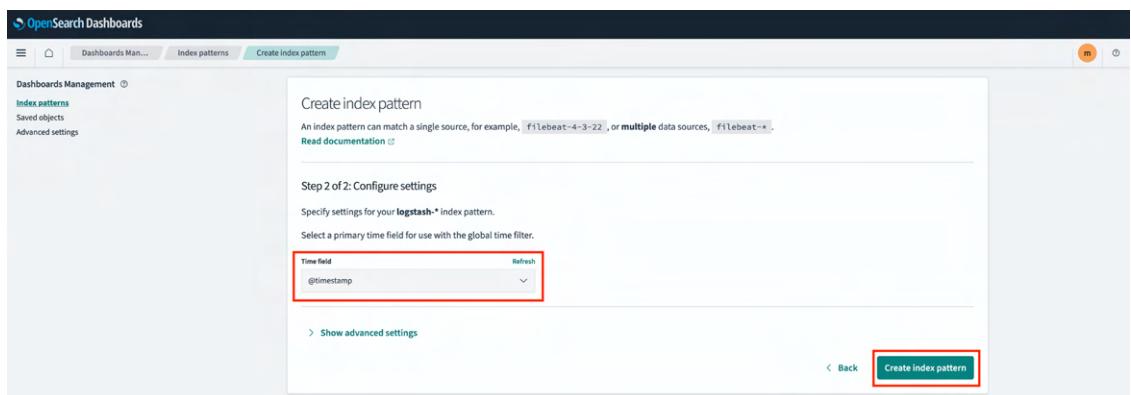
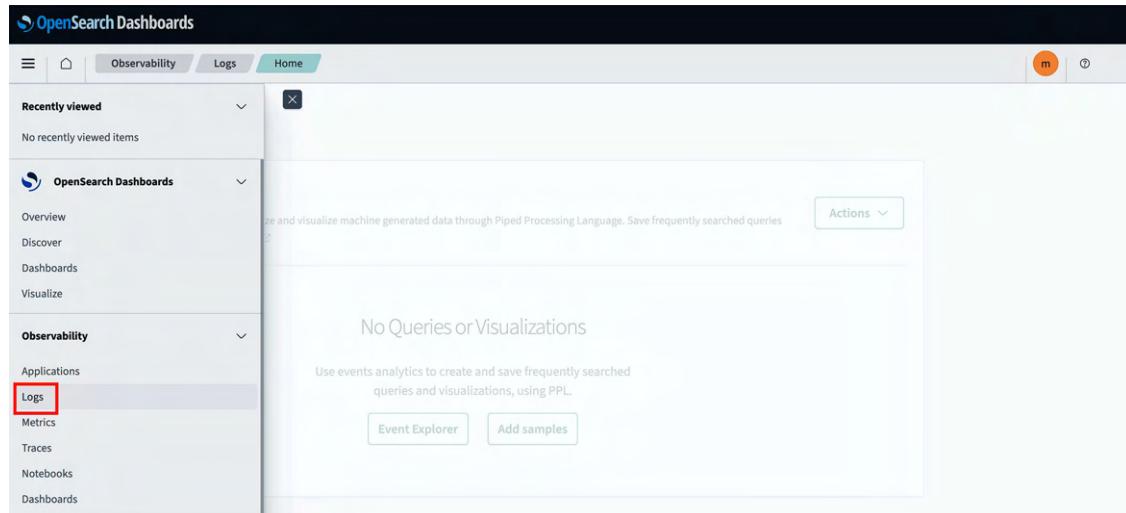


Figure 15.24: Index Time field

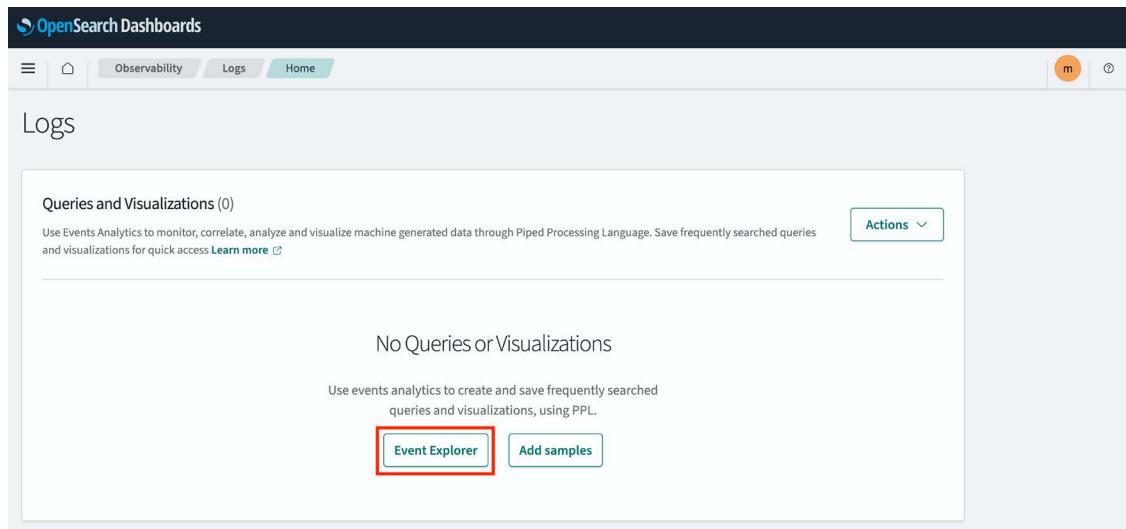
The next screen will show you the list of all the fields that can be searched. These fields are what are created by FluentBit and provide much easier searching of logs. They have all kinds of metadata from Kubernetes, including the namespace, labels, annotations, pod names, and so on. With our index pattern created, next, we'll want to query the log data to find which logs are coming from `ingress-nginx`. Next, click on the three horizontal bars in the upper-left corner again to reveal the menu and, under **Observability**, click **Logs**:



The screenshot shows the OpenSearch Dashboards interface. The top navigation bar has tabs for 'Observability' (selected), 'Logs', and 'Home'. Below the navigation is a sidebar with sections for 'Recently viewed' (empty), 'OpenSearch Dashboards' (Overview, Discover, Dashboards, Visualize), and 'Observability' (Applications, Logs, Metrics, Traces, Notebooks, Dashboards). The 'Logs' item in the 'Observability' section is highlighted with a red box. The main content area displays a message: 'No Queries or Visualizations' and 'Use events analytics to create and save frequently searched queries and visualizations, using PPL.' It includes 'Event Explorer' and 'Add samples' buttons.

Figure 15.25: Logs menu item

We haven't created any visualization, so there's nothing to see yet! Click on **Event Explorer**:



The screenshot shows the 'Logs' screen within the OpenSearch Dashboards interface. The top navigation bar has tabs for 'Observability' (selected), 'Logs', and 'Home'. The main content area displays a message: 'No Queries or Visualizations (0)' and 'Use Events Analytics to monitor, correlate, analyze and visualize machine generated data through Piped Processing Language. Save frequently searched queries and visualizations for quick access [Learn more](#)'. It includes 'Event Explorer' and 'Add samples' buttons. The 'Event Explorer' button is highlighted with a red box.

Figure 15.26: The Logs screen

On the next screen, set the **index pattern** to **logstash-\***, and the **timeframe** to the last 15 hours. Finally, click **Refresh**.

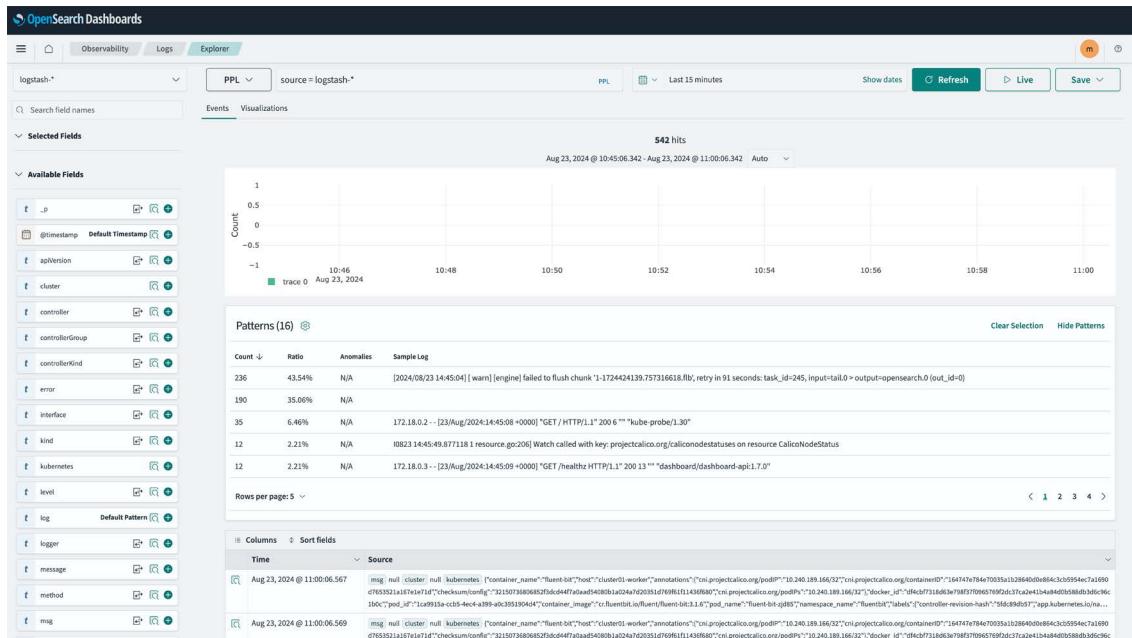


Figure 15.27: Logs Explorer

This will load quite a bit of data, most of which is meaningless to us. We only want data from the `ingress-nginx` namespace. So, we'll want to constrain the results to our `ingress-nginx` namespace. Next to **PPL**, paste in the following and click **Refresh**:

```
source = logstash-*
| where kubernetes.namespace_name=="ingress-nginx"
| fields log
```

Now, you'll see the access logs from NGINX:

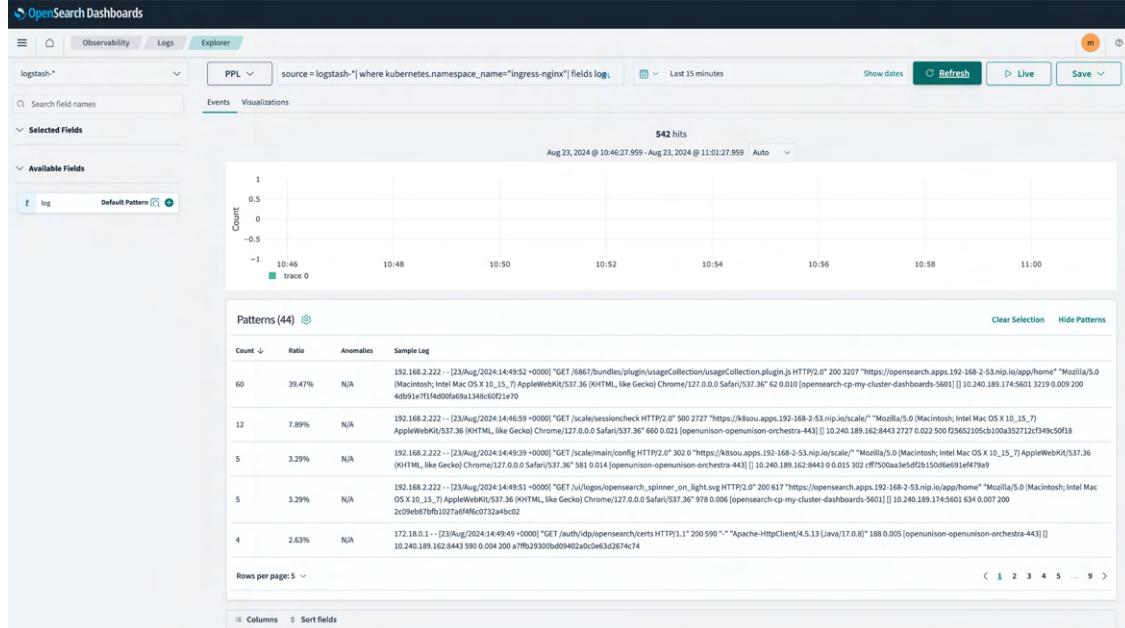


Figure 15.28: Searching for NGINX logs

While we're now able to search our logs from a central location, this only scratches the surface of what OpenSearch is capable of. As we said earlier in the chapter, there are books written on this topic alone, so we'll not be able to get too proficient in OpenSearch in this chapter but we have covered enough to demonstrate how logs move from your containers into a centralized system. Whether you're deploying an on-premises cluster, like our KinD cluster, or a cloud-based cluster, the same concepts will exist.

## Summary

Logging and monitoring are crucial to being able to track the health of your cluster, planning for ongoing maintenance and capacity, and also making sure to maintain compliance. In this chapter, we started with monitoring, walking through the Prometheus stack, and exploring each component and how they interact. After looking at the stack, we worked on how to monitor systems running on our cluster by integrating our OpenUnison into Prometheus. The last Prometheus topic we explored was integrating the stack into our enterprise authentication system using OpenUnison.

After working through Prometheus, we explored logging in Kubernetes by deploying an OpenSearch cluster to centralize our log aggregation. After deployment, we tracked logs from the container that generates them into OpenSearch's indexes and then how to access them securely using OpenSearch's Kibana dashboard.

In the next chapter, we're going to learn how service meshes work and deploy Istio.

## Questions

1. Prometheus' metrics are transferred using JSON.
  - a. True
  - b. False
2. Where can Alertmanager send alerts to?
  - a. Webhook
  - b. Slack
  - c. Email
  - d. All of the above
3. What label does a ConfigMap that stores a Grafana dashboard need?
  - a. `grafana_dashboard: 1`
  - b. `dashboard_type: grafana`
  - c. None needed
4. OpenSearch is compatible with Elasticsearch.
  - a. True
  - b. False
5. Logstash is required for log management.
  - a. True
  - b. False

## Answers

1. b: False: Prometheus has its own format for metrics.
2. d: Alertmanager can send notifications to all of these systems and more.
3. a: Grafana looks for all ConfigMaps across the cluster with `grafana_dashboard: 1` to load dashboards.
4. b: False: OpenSearch was forked from Elasticsearch 7.0; the two systems have since diverged.
5. b: False: Logstash is not required; systems such as FluentD and FluentBit are also compatible with OpenSearch and Elasticsearch.

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 16

## An Introduction to Istio



*"If it makes it easier for users to use on the frontend, it's probably complex on the backend."*

This chapter will introduce you to Istio, a Service mesh add-on for Kubernetes. A Service mesh is a tool for Kubernetes that improves the management, security, and visibility of microservices in a cluster. It simplifies complex networking tasks by handling service-to-service communication, load balancing, and traffic routing outside the application code. Istio also enhances security with features like encryption, authentication, and authorization. It provides detailed metrics and monitoring, allowing developers to understand how their services are performing.

Istio is a large, complex system that provides benefits to your workloads by offering enhanced security, discovery, observability, traffic management, and more – all without requiring application developers to write modules or applications to handle each task.

While Istio presents a steep learning curve, mastering it unlocks the potential to offer developers advanced functionalities, enabling intricate Service mesh deployments and a broad spectrum of capabilities, including the ability to do the following:

- Route traffic based on various requirements
- Secure service-to-service communication
- Traffic shaping
- Circuit breaking
- Service observability
- The future: Ambient mesh

Developers can use these tools with minimal or no changes to their code. When something is easy for users, it often means there's a lot of complexity behind the scenes, and Istio is a good example of this. This chapter will show you how to set up Istio and Kiali, a tool for monitoring. We'll also discuss Istio's key features that help manage traffic, enhance security, and reveal workloads.

To fully explain Istio, we'd need a whole other book focused just on its custom resources and ways to use them. Our aim in this chapter and the next is to give you the basic knowledge you need to start using Istio confidently. We can't go into every detail about each component, so we recommend visiting the Istio website at <https://istio.io> for additional information to build on what you will learn in this chapter.

This chapter will cover the following topics:

- Understanding the control plane and data plane
- Why should you care about a Service mesh?
- Introduction to Istio concepts
- Understanding Istio components
- Installing Istio
- Introducing Istio resources
- Deploying add-on components to provide observability
- Deploying an application into the Service mesh
- The future: Ambient mesh

Before we dive into the chapter, let's set the scene for what you're about to learn. This chapter is designed to introduce you to deploying Istio and the main features it provides. It gives you the key details to understand how Istio functions and what its various parts are. By the end of this chapter and the next chapter, where you'll add an application to the mesh, you should have a good grasp of how to set up and use a basic Istio Service mesh.

We will close out the chapter with a look at the future of the Service mesh, known as the ambient mesh. As of the publishing of this book, the ambient mesh is still in beta, and since a lot of things can change between beta and a GA release, we will only provide an overview of what the ambient mesh will bring when it's released.

To wrap up this introduction, here's a fun fact about Kubernetes: like many things in Kubernetes, Istio is named after something related to the sea. In Greek, "Istio" means "sail."

## Technical requirements

This chapter has the following technical requirements:

- A Docker host installed using the steps from *Chapter 1, Docker and Container Essentials*, with a minimum of 8 GB of RAM, although 16 GB is recommended
- A KinD cluster configured using the initial scripts from *Chapter 2, Deploying Kubernetes Using KinD*
- Installation scripts from this book's GitHub repository

You can access the code for this chapter by going to this book's GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter16>.

To use Istio to expose workloads, we will remove NGINX from the KinD cluster, which will allow Istio to utilize ports 80 and 443 on the host.

## Understanding the Control Plane and Data Plane

The Service mesh framework enhances how microservices communicate, making these interactions more secure, faster, and more reliable. It is separated into two main components: the control plane and the data plane, each playing a specific role in providing service-to-service communication in Kubernetes. These two layers are what make up the Service mesh as a whole and a basic understanding of each is key to understanding Istio.

### The Control Plane

Let's start with the Istio control plane.

The control plane in Istio is the central authority that controls and directs how services in the mesh communicate with each other. A common analogy is to think of a city's traffic control, managing roads and traffic signals to ensure safe and efficient traffic flow and order. It plays an important role in the orchestration of service-to-service communication, security, and observability across the entire mesh. We will discuss the different features that are controlled by the control plane when we discuss the main control plane daemon, `istiod`, in the *Understanding the Istio components* section.

### The Data Plane

The second component is the data plane, which is the worker layer that contains a set of proxies, known as Envoy proxies, deployed alongside your services. These proxies intercept and manage the network traffic between microservices without the need for you or your developers to do any additional work or recoding.

To build on the previous analogy we used for the control plane, think of the data plane as the roads in a city. Each service in the mesh receives a dedicated road where traffic is directed to the service by a traffic controller – in our case, the Istio control plane (`istiod`).

Up until now, you may have been using Kubernetes without a Service mesh. So, you may be wondering why you should care about the features that Istio brings to your cluster. In the next section, we will go over Istio's features so you will be able to explain to your developers and businesses why a Service mesh is an important add-on to clusters.

## Why should you care about a Service mesh?

Service meshes, like Istio, provide several features that developers would typically need to develop on their own, necessitating changes to their existing code. Without Istio, if developers require specific capabilities, such as secure communication between services coded in various programming languages like Java, Python, or Node.js, they would need to implement the necessary code, or libraries, for each language individually. This adds complexity to the code and can often lead to inefficient coding, causing performance issues with the application.

Adding security, like encryption, is just one example of what a developer may require to create an application. What about other features, like controlling data flow, testing how the application will react to failures, or how network delays may cause the application to behave in an unexpected way? These, and many other features, are all included with Istio – allowing developers to focus on their own business code, rather than writing additional code to control traffic or to simulate errors or delays.

Let's take a look at some of the advantages that Istio offers.

## **Workload observability**

Downtime or slowdowns in an application may impact your organization's reputation and potentially cause lost revenue.

Have you ever found it challenging to pinpoint the underlying issue in an application with many active services? Imagine the ability to quickly identify and resolve problems by monitoring the interactions and status of services in real time, or by replaying events from the past to discover where things went wrong hours or days before.

Managing complex applications and multiple services can feel overwhelming. The features offered by Istio make this task far less intimidating. If only there was a way to make this easier for developers! Thanks to Istio and its ecosystem, this isn't just wishful thinking. With powerful features like Prometheus for storing metrics, Kiali for deep insights, and Jaeger for detailed tracing, you're equipped with a powerful toolkit for troubleshooting.

In this chapter, we're going to set up all three add-ons, with a focus on using Kiali, which lets you observe communications between your services like never before.

## **Traffic management**

Istio provides you with powerful traffic management capabilities for your workloads, offering the flexibility to adopt any deployment model you need, without the hassle of altering your network infrastructure. This control is entirely in your and your developers' hands. Istio also includes tools that enable you to simulate various unpredictable scenarios that your application might encounter, such as HTTP errors, delays, timeouts, and retries.

We recognize that some of our readers might be new to the concept of deployment models. Grasping the different types available is crucial for understanding, and appreciating, the benefits that Istio brings to the table. With Istio, developers can effectively utilize deployment strategies like blue/green and canary deployments.

## **Blue/green deployments**

In this model, you deploy two versions to production, directing a certain percentage of traffic to each version of the application, usually sending a low amount of traffic to the “new” (green) release. As you verify that the new deployment is working as expected, you can cut over all of the traffic to the green deployment, or you use blue/green combined with a canary deployment to the new version until you are eventually sending 100% of the traffic to the new deployment.

## Canary deployments

This term comes from the mining days when miners would put a canary in the mining shaft to verify it was safe to work in the environment. In the case of a deployment, it allows you to deploy an early test version of the application before graduating the release to a new version. Essentially, this is similar to the blue/green deployment, but in a canary deployment, you would direct a very small percentage of traffic to the canary version of the application. Using a small percentage of traffic will minimize any impact that the canary deployment may introduce. As you become more confident that the “canary” version of the application is stable, you will move over additional traffic until you are 100% on the new version.

## Finding issues before they happen

We can go a step further from deployment models; Istio also provides tools for you to develop resilience and testing for your workloads before you deploy them and learn about issues from customers or end-users.

Have you ever worried about how an application will react to certain unseen events?

Developers need to worry about events that they have little control over, including:

- Application timeouts
- Delays in communication
- HTTP error codes
- Retries

Istio provides objects to assist in dealing with these by allowing you to create an issue with the workload before you move to production. This allows developers to capture and resolve issues in their applications before releasing them to production, creating a better user experience.

## Security

In today’s world, security is an issue we should all be concerned about. Many of the methods to secure a workload are complex and may require a skillset that many developers do not have. This is truly where Istio shines, providing the tools to easily deploy security and minimize its impact on development.

The first, and most popular, security feature in Istio is the ability to provide **mutual Transport Layer Security (mTLS)** between workloads. Using mTLS, Istio provides not only encryption for communication but workload identity too. When you visit a website that has an expired certificate or a self-signed certificate, your browser will warn you that the site can’t be trusted. That’s because your browser performs server authentication when establishing a TLS connection by verifying that the certificate presented by the server is trusted by the browser. mTLS verifies trust from the client to the server, but also from the server to the client. That’s the mutual part. The server validates that the certificate presented by the client is trusted as well as the client validating the server. When you first start a cluster and use the initial certificate created for you, you’re using mTLS. Istio makes this much easier because it will create all of the certificates and identities for you using its built-in sidecar.

You can configure mTLS as a requirement (STRICT), or as an option (PERMISSIVE), for the entire mesh or individual namespaces. If you set either option to STRICT, any communication to the service will require mTLS and if a request fails to provide an identity, the connection will be denied. However, if you set the PERMISSIVE option, traffic that has an identity and requests mTLS will be encrypted, while any request that does not provide an identity or encryption request will still be allowed to communicate.

Another feature provided will give you the ability to secure what communication is allowed to a workload, similar to a firewall, but in a much simpler implementation. Using Istio, you can decide to only allow HTTP GET requests, or only HTTP POST requests, or both – from only defined sources.

Finally, you can use **JSON Web Tokens (JWTs)** for initial user authentication to limit who is authorized to communicate with a workload. This allows you to secure the initial communication attempt by only accepting JWTs that come from an approved token provider.

Now that we have discussed some of the reasons you would want to deploy Istio, let's introduce you to some Istio concepts.

## Introduction to Istio concepts

The principles of Istio can be divided into four main areas: traffic management, security, observability, and extensibility. For each of these areas, we'll introduce the components and custom resources that developers can utilize to tap into the benefits of using Istio.

## Understanding the Istio components

Similar to a standard Kubernetes cluster, Istio refers to two separate planes, the control plane and the data plane. Historically, the data plane included four different services, Pilot, Galley, Citadel, and Mixer – all broken out in a true microservices design. This design was used for multiple reasons, including the flexibility to break out the responsibilities to multiple teams, the ability to use different programming languages, and the ability to scale each service independently of the others.

Istio has evolved quickly since its initial release. The team made the decision that breaking out the core services had little benefit and, in the end, made Istio more complex. This led the team to redesign Istio and starting with Istio 1.5, Istio includes the components that we will discuss in this section.

## Making the Control Plane simple with `istiod`

Just as Kubernetes bundles multiple controllers into a single executable, `kube-controller-manager`, the Istio team decided to bundle the control plane components into a single daemon called `istiod`. This single daemon grouped all of the control plane components into a single pod that can be easily scaled as performance is required.

The main advantages to the single daemon are listed in an Istio blog at <https://istio.io/latest/blog/2020/istiod/>. To summarize the team's reasoning, the single process provides:

- Easier and quicker control plane installations
- Easier configuration

- Integration of virtual machines into the Service mesh more easily, requiring a single agent and Istio's certificates
- Easier scaling
- Reduced control plane startup time
- A reduced amount of overall required resources

The control plane is responsible for controlling your Service mesh. It has a number of important features that are required to create and manage the components of Istio, and in the next section, we will explain the features that istiod provides.

## Breaking down the istiod pod

Moving to a single binary didn't reduce Istio's functionality or features; it still provides all of the features that the separate components provided, they are all just in a single binary now. Each piece provides a key feature to the Service mesh, and in this section, we will explain these features:

- **Service Discovery:** Ensures that Envoy proxies, deployed alongside a Service in the same pod, have up-to-date information about the network locations, including the IP address and ports of services in the mesh. Service Discovery provides efficient service-to-service communication across the mesh.

Services can frequently scale up or down, and pods may be terminated or launched as part of rolling updates or auto-scaling activities. Each change can potentially alter the endpoints. Service discovery automates the process of tracking these changes by watching for updates to services and their associated pods. When a change has occurred, the Service discovery component updates the internal registry of Service endpoints and pushes these updates to the Envoy sidecars.

Service Discovery is essential for maintaining the responsiveness and efficiency of the Service mesh, dynamically adapting to the ever-changing landscape of containerized application environments in real time.

- **Configuration Distribution:** Handles the configuration of traffic routing, security protocols, and policy enforcement across the data plane's sidecars. Configuration distribution centralizes functions that used to be performed by a component called Galley, including the authorization of configuration changes in the Service mesh.
- **Certificate Lifecycle Management:** Manages the issuing, renewal, and revocation of digital certificates, which are used for secure service-to-service communication using mutual Transport Layer Security (mTLS) guaranteeing that was previously handled by a component called Citadel, which provided identity verification and management of certificates, guaranteeing that all services within the mesh can trust connections, without requiring any additional components or configuration. mTLS reduces security threats by encrypting the transferring of data between services, providing the confidentiality and integrity of communication within the Service mesh.

- **Automated Envoy Proxy Deployment:** Streamlines the deployment process by automatically deploying Envoy sidecar proxies within Kubernetes pods. This seamless integration optimizes the management of both Egress and Ingress traffic through the pods, serving as an invisible mediator that oversees network traffic.

This automated process ensures that each pod in the Service mesh receives its own Envoy proxy, providing advanced traffic capabilities including routing, load distribution, and security measures. The automation of Envoy proxy deployment removes the complexities involved in establishing and upkeep the Service mesh, allowing developers and operators to spend time on their primary responsibilities.

- **Traffic Routing and Control:** Responsible for creating and sharing the rules for managing traffic to Envoy proxies, providing a pivotal role in executing advanced network operations, and providing a way for precise control over often complex network traffic flows.

The functionalities provided by traffic routing and control, include:

- Determining the path(s) for directing traffic
- Strategic retry mechanisms
- Failover schemes to ensure continuity
- The introduction of faults for the purpose of creating realistic testing environments

Traffic routing and controls provide a number of advantages, including streamlining the management of network traffic, testing the network's stability and response under certain conditions, and increasing workload resilience by simulating disruptions and how the application reacts before they occur in production.

- **Security Policy Enforcement:** Uses security rules to make sure only authorized users or services have access and can interact within the network securely.
- **Observability Data Collection:** In a Service mesh, it's essential to keep an eye on how things are running and to quickly identify and solve any problems. This is where observability data collection comes into play, gathering and analyzing telemetry information, including metrics, logs, and traces from the data plane, enhancing the mesh's monitoring and operational insight capabilities.

Now that we have discussed what istiod provides, we will move on to how incoming traffic is managed in the Service mesh using the `istio-ingressgateway` component.

## **Understanding `istio-ingressgateway`**

Moving on from the base istiod pod, we come to one of the most important components of Istio, `istio-ingressgateway`. This gateway facilitates external clients' and services' access to the Service mesh, acting as the entry point into the Kubernetes cluster. It's standard for every Istio-enabled cluster to be equipped with at least one instance of `istio-ingressgateway`. However, Istio's design does not confine you to just one; depending on your specific needs, it's possible to deploy multiple ingress gateways to serve various purposes or handle different traffic patterns.

The `istio-ingressgateway` provides access to applications using two methods:

1. Standard Kubernetes Ingress object support
2. Istio Gateway and VirtualService objects

Since we have already discussed and deployed NGINX as an Ingress controller, we will not cover using Envoy as a standard Ingress controller; instead, we will focus on the second method of using Gateways and Virtual Services for incoming requests.

Using Gateways to expose our services provides more flexibility, customization, and security over a standard Ingress object.

## Understanding `istio-egressgateway`

The `istio-egressgateway` is designed to direct traffic from sidecars to either a single pod or a collection of pods, thereby centralizing the exiting (egress) traffic from the Service mesh. Normally, Istio sidecars manage both incoming and outgoing traffic for services within the mesh. While `istio-ingressgateway` is utilized for managing incoming traffic to the mesh, implementing `istio-egressgateway` allows for the regulation of outgoing traffic as well. The functionalities and details of both `ingressgateway` and `egressgateway` will be explored thoroughly in the *Introducing Istio resources* section.

Now, let's jump into how you install Istio in a cluster.

## Installing Istio

There are multiple methods to deploy Istio. The most common methods today are to use either `istioctl` or Helm, but there are additional options depending on your organization. You may elect to use one of the alternative installation methods of creating manifests via `istioctl` or Helm.

A brief list of advantages and disadvantages for each method is detailed in *Table 16.1*:

Deployment method	Advantages	Disadvantages
<code>istioctl</code>	<p>Configuration validation and health checks</p> <p>Does not require any privileged pods, increasing cluster security</p> <p>Multiple configuration options</p>	Each Istio version requires a new binary
Istio operator	<p>Configuration validation and health</p> <p>Does not require multiple binaries for each Istio version</p> <p>Multiple configuration options</p>	Requires a privileged pod running in the cluster

Manifests (via <code>istioctl</code> )	Generates manifests that can be customized before deploying using <code>kubectl</code> Multiple configuration options	Not all checks are performed, which could lead to deployment errors Error checks and reporting are limited when compared to using <code>istioctl</code> or the Istio operator
Helm	Helm and charts are well known to most Kubernetes users Leverages Helm standards, which allow for easy management of deployments	Offers the least validation checks of all deployment options Most tasks will require additional work and complexity versus the other deployment models

Table 16.1: Istio deployment methods

For this chapter, we will focus on using the `istioctl` binary for installation, and in the next section, we will deploy Istio using `istioctl`.

## Downloading Istio

We have included a script that will deploy Istio, output the verification of the installation, remove NGINX Ingress, and expose `istio-ingressgateway` as the Ingress to our KinD cluster. The manual process is provided below if you prefer to install it manually using `istioctl`. The script, `install-istio.sh`, is provided for readers who may use it in automation for their own testing and is located in the `chapter16` directory.

The first thing that we need is to define the version of Istio we want to deploy. We can do this by setting an environment variable, and in our example, we want to deploy Istio 1.20.3. First, make sure you are in the `chapter16` directory where you cloned the repo and execute the following command:

```
curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.20.3 TARGET_  
ARCH=x86_64 sh -
```

This will download the installation script and execute it using the `ISTIO_VERSION` that we defined before executing the `curl` command. After executing, you will have an `istio-1.20.3` directory in your current working directory.

Finally, since we will be using executables from the `istio-1.12.3` directory, you should add it to your path statement. To make this easier, you should be in the `chapter16` directory from the book repository before setting the path variable:

```
export PATH="$PATH:$PWD/istio-1.20.3/bin"
```

## Installing Istio using a profile

To make deploying Istio easier, the team has included a number of pre-defined profiles. Each profile defines which components are deployed and the default configuration. There are seven profiles included, but only five profiles are used for most deployments.

Profile	Installed Components
Default	<code>istio-ingressgateway</code> and <code>istiod</code>
Demo	<code>istio-egressgateway</code> , <code>istio-ingressgateway</code> , and <code>istiod</code>
Minimal	<code>istiod</code>
Preview	<code>istio-ingressgateway</code> and <code>istiod</code>
Ambient	<code>istiod</code> , CNI, and Ztunnel Note: In the 1.20.3 Istio release, ambient mesh is an alpha feature

*Table 16.2: Istio profiles*

If none of the included profiles fit your deployment requirements, you can create a customized deployment. This is beyond the scope of this chapter since we will be using the included demo profile – however, you can read more about customizing the configuration on Istio’s site: <https://istio.io/latest/docs/setup/additional-setup/customize-installation/>.

To deploy Istio using the demo profile using `istioctl`, we simply need to execute a single command:

```
istioctl manifest install --set profile=demo
```

The installer will ask you to verify that you want to deploy Istio using the default profile, which will deploy all of the Istio components:

```
This will install the Istio 1.20.3 "demo" profile (with components: Istio core, Istiod, Ingress gateways, and Egress gateways) into the cluster. Proceed? (y/N)
```

Press the `y` key to say yes to proceed with the deployment. If you want to bypass the confirmation, you can add an option to the `istioctl` command line, `--skip-confirmation`, which tells `istioctl` to bypass the confirmation.

If everything went well, you should see a confirmation that each component was installed, and a completion message that thanks you for installing Istio.

```
✓ Istio core installed
✓ Istiod installed
✓ Egress gateways installed
✓ Ingress gateways installed
✓ Installation complete
Made this installation the default for injection and validation.
```

The `istioctl` executable can be used to verify the installation. To verify the installation, you require a manifest. Since we used `istioctl` to deploy Istio directly, we do not have a manifest, so we need to create one to check our installation.

```
istioctl manifest generate --set profile=demo > istio-kind.yaml
```

Then run the `istioctl verify-install` command.

```
istioctl verify-install -f istio-kind.yaml
```

This will verify each component, and once verified, it will provide a summary similar to the output below:

```
Checked 15 custom resource definitions
Checked 3 Istio Deployments
✓ Istio is installed and verified successfully
```

Now that we have verified the installation, let's look at what `istioctl` created:

- A new namespace called `istio-system`.
- Three deployments were created, and a corresponding service for each:
  - `istio-ingressgateway`
  - `istio-egressgateway`
  - `istiod`
- 15 **CustomResourceDefinitions (CRDs)** to provide the Istio resources, including:
  - `destinationrules.networking.istio.io`
  - `envoyfilters.networking.istio.io`
  - `gateways.networking.istio.io`
  - `istiooperators.install.istio.io`
  - `peerauthentications.security.istio.io`
  - `proxyconfigs.networking.istio.io`
  - `requestauthentications.security.istio.io`
  - `serviceentries.networking.istio.io`
  - `sidecars.networking.istio.io`
  - `telemetries.telemetry.istio.io`
  - `virtualservices.networking.istio.io`
  - `wasmplugins.extensions.istio.io`
  - `workloadentries.networking.istio.io`
  - `workloadgroups.networking.istio.io`

At this stage, there's no need to be concerned with the details of the Custom Resources (CRs). As we progress through this chapter, we'll delve into the specifics of the most commonly used resources. Following that, in the next chapter, we'll cover how to deploy an application into the mesh, which will make use of several of the CRs that have been deployed.



For any CRs that are not covered in this chapter or the next chapter, you can reference the documentation on the [istio.io site](https://istio.io/latest/docs), located here: [istio.io/latest/docs](https://istio.io/latest/docs)

## Exposing Istio in a KinD cluster

With Istio deployed, our next step is to expose it to our network so we can access the applications we'll build. Since we're running on KinD, this can be tricky. Docker is forwarding all traffic from port 80 (HTTP) and 443 (HTTPS) on our KinD server to the worker node. The worker node is in turn running the NGINX Ingress controller on ports 443 and 80 to receive that traffic. In a real-world scenario, we'd use an external load balancer, like MetalLB, to expose the individual services via a LoadBalancer. For our labs though, we're going to instead focus on simplicity.

When you executed the previous script to install Istio, the last step ran a separate script called `expose_istio.sh` that does two things. First, it will delete the `ingress-nginx` namespace, removing NGINX and freeing up ports 80 and 443 on the Docker host. Second, it will patch the `istio-ingressgateway` Deployment in the `istio-system` namespace so that it runs on ports 80 and 443 on the worker node.

Since the script was executed as part of the installation, you do not need to execute it again.

Now that we have Istio fully deployed in our cluster and we know the custom resources that Istio includes, let's move on to the next section, which will explain each resource and its use-cases.

## Introducing Istio resources

Istio's custom resources provide powerful features to your cluster and each one could take up a chapter by itself.

In this section, we want to provide enough details so you will have a strong understanding of each object. After the object overview, we will deploy a basic application that will demonstrate many of the objects in a real-world application example.

## Authorization policies

Authorization policies are optional; however, if you do not create any, all requests to resources will be allowed access to your cluster workloads. This may be the desired default action for some organizations, but most enterprises should deploy workloads based on the least required privileges. This means that you should only allow what access is required for accessing the application – nothing more and nothing less. Least privilege access is often overlooked by organizations since it adds some complexity to access and if not configured correctly, it may deny access to valid requests. While this is true, it is not a valid argument to leave your systems wide open to all access requests.

Istio's authorization policies offer detailed access management for services within your mesh, allowing you to define access rights based on the identities of the callers and their permissions. They provide developers with the ability to control access to workloads based on actions including deny, allow, and custom.

Before explaining policies in more depth, we need to start with a concept called **implicit enablement**. This means that when **any** authorization policy matches a request, Istio will change the default allow all to a deny for any request that doesn't have a match in the policy.

Let's use an example to explain this in more detail. We have an NGINX server running in a namespace where Istio has been enabled and we have a standard that access to port 80 must be denied.

At a quick glance, this looks like it should be an easy policy, we would simply create a deny policy that contains port 80. So, we create the policy and deploy it to our cluster – and to verify the policy, we try to access the website on port 80. We open a browser and, as expected, we cannot access the site. Great! Now let's verify that we can access the site on port 443. We change the URL to access the site using port 443, and to our surprise, it is also *denied*.

Wait, what?!? The deny policy only denies port 80 – why is port 443 also being denied?

This is **implicit enablement** in action and it can be confusing at first for anyone who is new to Istio. As discussed at the beginning of this section, when a policy is created and a request matches that policy, Istio will change from an *allow all* security posture, to a *deny all* security posture. Even though we intended to deny access to only port 80, without an allow policy for port 443, access will also be denied.

To complete the requirement, and allow access to our NGINX site on port 443, we would need to create an allow policy that allows all incoming traffic to port 443. We will explain this in more depth in a minute.

Understanding how a policy's actions are evaluated is very important, since a misconfigured policy may not provide the expected results. The high-level flow for policy evaluation is shown in *Figure 16.1*.

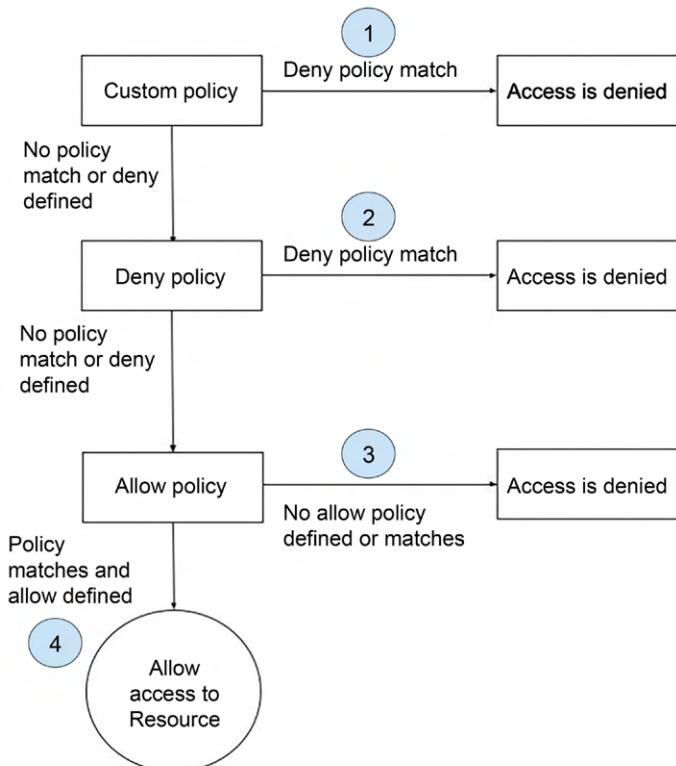


Figure 16.1: Istio policy evaluation flow

1. If the evaluation of a **CUSTOM** action defines a **DENY** to the request, the access is denied, and the evaluation process stops.
2. Next, if a **DENY** policy matches the request, the request is denied access to the resource, and the evaluation process stops.
3. If there are no **ALLOW** policies that match the request, access will be denied to the request.
4. If an **ALLOW** policy matches the request, access to the resource is granted.

Along with understanding the flow of policies, you need to understand how conflicting policies will be implemented. If a policy has any conflicting rules, like denying and allowing the same request, the deny policy will be evaluated first and the request will be denied since the deny policy is evaluated before the allow policy. It's also very important to note that if you allow a certain action, like an HTTP GET, the GET request would be allowed, but any other operation would be denied since it has not been allowed by the policy.

Authorization policies can get very complex. The Istio team has created a page with multiple examples on the Istio site at <https://istio.io/latest/docs/reference/config/security/authorization-policy/>.

Policies can be broken down into scope, action, and rules:

- **Scope:** The scope defines what object(s) will be enforced by the policy. You can scope a policy to the entire mesh, a namespace, or any Kubernetes object label like a pod.
- **Actions:** There can be one of three actions defined, CUSTOM, ALLOW, or DENIED – each either denying or allowing a request based on the defined rules. ALLOW and DENY are the most commonly used actions, but CUSTOM actions are beneficial when you require complex logic that ALLOW or DENY cannot handle, using an external system for additional decision making.
- **Rules:** Define what actions will be allowed or denied by the request. Rules can become very complex, allowing you to define actions based on source and destination, different operations, keys, and more.

To help understand the flow, let's look at a few example authorization policies and what access will be applied when the policy is evaluated.

We will deploy a larger application later in the chapter. If you want to see how the example policies work in this section, you can deploy an NGINX web server using the script in the `chapter16/testapp` directory called `deploy-testapp.sh`. This will create all of the required Kubernetes and Istio objects to test the policies in a real cluster.

Once you have executed the script and the objects have been created, test that NGINX is working by curling to the `nip.io` VirtualService that was created. On our server, it created `testapp.10.3.1.248.nip.io`.

```
curl -v testapp.10.3.1.248.nip.io
```

This should display the NGINX welcome page. Now, we can create some example policies to show how authorization policies work. Each of the three examples is in the chapter16/testapp directory named `exampleX-policy.yaml`, where X is equal to 1, 2, or 3 – each can be deployed using the `kubectl apply <policynname> -n testapp` command.

## Example 1: Denying and allowing all access

For our first example, we will create a policy that will deny all requests to the resources in the namespace `testapp`:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: testapp-policy-deny
  namespace: testapp
spec:
  {}
```

After deploying the policy, attempt to `curl` to the `nip.io` address again. You will notice that this time, you will be denied access and Istio will return an RBAC error:

```
RBAC: access denied
```

It can take a few seconds for Istio to enable new policies. If you did not receive the `RBAC: access denied` error, wait a few seconds and try again.

This is a very simple policy that does not include a `selector` and defines nothing in the `spec` section. By omitting the `selector` section, Istio will apply the policy to all workloads in the namespace, and by not including anything in the `spec` section, Istio will deny all traffic. If we refer back to the policy flow diagram in *Figure 16.1*, this would flow down the bottom and evaluate as circle #3 – there is a `selector` match, which is *all* workloads in the namespace, and there hasn't been an `ALLOW` policy defined. This will lead to the request being denied.

We won't deploy the next example; we are showing it to reinforce the example provided above. By adding a single entry to the policy, we can change it from denying all requests to allowing all requests.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: testapp-policy-deny
  namespace: testapp
spec:
  rules:
    - {}
```

When we add the `rules` section to the policy definition, with a `{}`, we are creating a rule that means all traffic. Similar to the previous example, we have not added a `selector`, which means the policy will apply to all deployments in the namespace. Since this rule is for all workloads and the rule includes all traffic, access would be allowed.

You might be starting to see why we mentioned how not understanding how policies are evaluated in the flow may lead to unexpected access results. This is a prime example of how a single entry, `rules`, changes the policy from denying all requests to allowing all requests.

Before moving on, delete the policy by executing:

```
kubectl delete -f example1-policy.yaml -n testapp
```

## Example 2: Allowing only GET methods to a workload

Policies can get very granular, allowing only certain operations like GET from an HTTP request. This example will allow GET requests while denying all other request types for pods that are labeled with `app=nginx-web` in the `marketing` namespace. For this example, we will use the same NGINX deployment from the first example. Create the policy using the manifest in the `chapter16/testapp` directory called `example2-policy.yaml` using `kubectl`:

```
kubectl create -f example2-policy.yaml -n testapp
```

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: nginx-get-allow
  namespace: marketing
spec:
  selector:
    matchLabels:
      app: nginx-web
  action: ALLOW
  rules:
  - to:
    - operation:
      methods: ["GET"]
```

If you attempt to `curl` to the same `nip.io` address as we used in example 1, you will see the NGINX welcome page. This is using an HTTP GET command. To prove that HTTP PUT commands will be blocked, we can use a `curl` command to send a request to NGINX:

```
curl -X PUT -d argument=value -d value1=dummy-data http://testapp.10.3.1.248.
nip.io/
```

This will cause Istio to deny the request with an RBAC error:

```
RBAC: access denied
```

In the example, the policy accepts the GET request from any source, since we have only defined an action without a specific `from` object. Since we have not added the PUT action in our policy, any attempt to send an HTTP PUT request will be denied by the policy.

Policies can get even more granular, accepting (or denying) a request based on the source of the request. In the next example, we will show another example of a policy but we will limit the source to a single IP address.

Before moving on, delete the example policy using `kubectl`:

```
kubectl delete -f example2-policy.yaml -n testapp
```

### Example 3: Allowing requests from a specific source

In our last policy example, we will limit what source will be allowed access to a workload using a GET or POST method.

This will increase security by denying any request from a source that is not in the policy source list. We will not create this policy since many readers may be limited in the number of machines they have to use for testing.

```
metadata:
  name: nginx-get-allow-source
  namespace: marketing
spec:
  selector:
    matchLabels:
      app: nginx
  action: ALLOW
  rules:
  - from:
    - source:
      ipBlocks:
      - 192.168.10.100
```

Unlike the previous examples, this policy has a `source:` section, which allows you to limit access based on different sources, like an IP address. This policy will allow the source IP `192.168.10.100` access to all operations on the NGINX server, and all other sources will be denied access.

Moving on from authorization policies, we will introduce our next custom resource, destination gateways.

## Gateways

Earlier, we mentioned that traffic will come into a central point, `istio-ingressgateway`. We didn't explain how the traffic flows from the `ingressgateway` to a namespace and workloads – this is where gateways come in.

A gateway can be configured at the namespace, so you can delegate the creation and configuration to a team. It is a load balancer that receives incoming and outgoing traffic that can be customized with options like accepted ciphers, TLS versions, certificate handling, and more.

Gateways work along with Virtual Services, which we will discuss in the next section, but until then, the following figure shows the interaction between the `Gateway` and `VirtualService` objects.

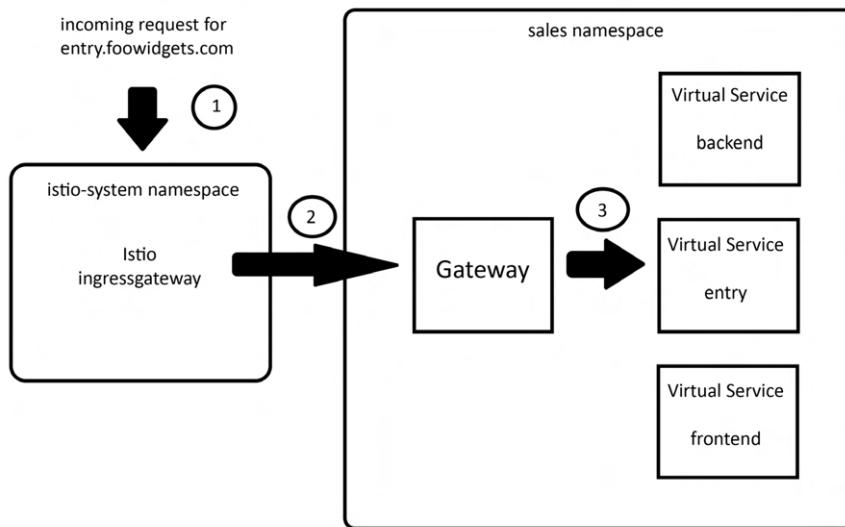


Figure 16.2: Gateway to Virtual Service communication flow

The list below explains the communication shown in Figure 16.2 in more detail:

1. An incoming request is sent to the Istio `ingress-gateway` controller, located in the `istio-system` namespace.
2. The `sales` namespace has a gateway configured that is set to use the `ingressgateway` with a host of `entry.foowidgets.com`. This tells the `ingressgateway` to send the request to the `gateway` object in the `sales` namespace.
3. Finally, the traffic is routed to the service using a `Virtual Service` object that has been created using the `gateway` in the `sales` namespace.

To show an example `Gateway` configuration, we have a namespace called `sales` that has Istio enabled, running an application that can be accessed using the URL `entry.foowidgets.com`, and we need to expose it for external access. To accomplish this, we would create a gateway using the example manifest below. (The example below is just for discussion; you do not need to deploy it on your KinD cluster.)

```
apiVersion: networking.istio.io/v1alpha3
```

```
kind: Gateway
metadata:
  name: sales-gateway
  namespace: sales
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 443
      name: http
      protocol: HTTP
    hosts:
    - sales.foowidgets.com
  tls:
    mode: SIMPLE
    serverCertificate: /etc/certs/servercert.pem
    privateKey: /etc/certs/privatekey.pem
```

This gateway configuration will tell the ingress gateway to listen on port 443 for requests that are incoming for `sales.foowidgets.com`. It also defines the certificates that will be used to secure the communication for incoming web requests.

You may be wondering, “How does it know to use the ingress gateway that we have running in our cluster?” If you look at the `spec` section, and then the `selector`, we have configured the `selector` to use an ingress gateway that has the label `istio=ingressgateway`. This `selector` and label tell the gateway object which ingress gateway will create our new gateway for incoming connections. When we deployed Istio earlier, the ingress gateways were labeled with the default label `istio=ingressgateway`, as shown highlighted below, from a `kubectl get pods --show-labels -n istio-system`.

```
app=istio-ingressgateway,chart=gateways,heritage=Tiller,install.operator.istio.
io/owning-resource=unknown,istio.io/rev=default,istio=ingressgateway,operator.
istio.io/component=IngressGateways,pod-template-hash=78c9969f6b,release=istio,s
ervice.istio.io/canonical-name=istio-ingressgateway,service.istio.io/canonical-
revision=latest,sidecar.istio.io/inject=false
```

You may be wondering how the gateway will be used to direct traffic to a particular workload since there are no configuration options in the gateway telling it where to direct traffic. That’s because the gateway just configures the ingress gateways to accept traffic for a destination URL and the required ports – it does not control how the traffic will flow to a service; that’s the job of the next object, the `Virtual Service` object.

## Virtual services

Gateways and virtual services combine to provide the correct traffic route to a service, or services. Once you have a gateway deployed, you need to create a virtual service object to tell the gateway how to route traffic to your service(s).

Building on the gateway example, we need to tell the gateway how to route traffic to our web server running on port 443. The server has been deployed using NGINX in the `marketing` namespace and it has a label of `app-nginx` and a service named `frontend`. To route traffic to the NGINX service, we would deploy the manifest below. (The example below is just for discussion; you do not need to deploy it on your KinD cluster.)

```
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: sales-entry-web-vs
  namespace: sales
spec:
  hosts:
    - entry.foowidgets.com
  gateways:
    - sales-gateway
  http:
    - route:
        - destination:
            port:
              number: 443
            host: entry
```

Breaking down the manifest, we specify the host(s) that this `VirtualService` object will route; in our example, we only have one host, `entry.foowidgets.com`. The next field defines which gateway will be used for the traffic; in the previous section, we defined a gateway called `marketing-gateway`, which was configured to listen on port 443.

Finally, the last section defines which service the traffic will be routed to. The `route`, `destination`, and `port` are all fairly straightforward to understand, but the `host` section can be misleading. This field actually defines the service that you will route the traffic to. In the example, we are going to route the traffic to a service called `entry`, so our field is defined with `host: entry`.

With this knowledge of using gateways and virtual services to route traffic in the Service mesh, we can move on to the next topic, destination rules.

## Destination rules

Virtual services provide a basic method to direct traffic to a service, but Istio offers an additional object to create complex traffic direction by using Destination rules. Destination rules are applied after Virtual Services. Traffic is initially routed using a Virtual Service and, if defined, a Destination rule can be used to route the request to its final destination.

This may be confusing at first, but it becomes easier when you see an example, so let's dive into an example that can route traffic to different versions of a deployment.

As we learned, incoming requests will use the Virtual Service initially, and then a destination rule, if defined, will route the request to the destination. In this example, we have already created a Virtual Service but we actually have two versions of the application labeled v1 and v2 and we want to direct traffic between both versions of the application using round-robin. To accomplish this, we would create a DestinationRule using the manifest below. (The example below is just for discussion; you do not need to deploy it on your KinD cluster.)

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: nginx
spec:
  host: nginx
  trafficPolicy:
    loadBalancer:
      simple: ROUND_ROBIN
  subsets:
    - name: v1
      labels:
        version: nginx-v1
    - name: v2
      labels:
        version: nginx-v2
```

Using this example, incoming requests to the NGINX server will be split between the two versions of the application equally since we defined the `loadBalancer` policy as `ROUND_ROBIN`. But what if we wanted to route traffic to the version that had the least number of connections? Destination rules have other options for `loadBalancer`, and to route connections to the version with the least connections, we would set the `LEAST_CONN` `loadBalancer` policy.

Next, we will discuss some of the security features Istio provides, starting with an object called Peer Authentication.

## Peer authentication

Istio's peer authentication object controls how the Service mesh controls the mutual TLS settings for workloads, either for the entire Service mesh or just a namespace. Each policy can be configured with a value that will either allow both encrypted communication and non-encrypted communication between pods or require encryption between pods.

mTLS mode	Pod communication	Description
STRICT	mTLS required	Any non-encrypted traffic sent to a pod will be denied
PERMISSIVE	mTLS optional	Both encrypted and non-encrypted traffic will be accepted by the pod

Table 16.3: PeerAuthentication options

If you wanted to set PeerAuthentication for the entire mesh, you would create a PeerAuthentication in the `istio-system` namespace. For example, to require mTLS between all pods, you would create the policy shown below:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: mtls-policy
  namespace: istio-system
spec:
  mtls:
    mode: STRICT
```

To allow both encrypted and non-encrypted traffic, the policy mode just needs to be set to PERMISSIVE, by changing the mode to `mode: PERMISSIVE`.

Rather than setting the mode for the entire mesh, many enterprises only set the mode to STRICT for namespaces that require additional security. In the example below, we set the mode to STRICT for the `sales` namespace.

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: mtls-policy
  namespace: sales
spec:
  mtls:
    mode: STRICT
```

Since this policy is configured for the `sales` namespace, rather than the `istio-system` namespace, Istio will only enforce a strict mTLS policy for the namespace rather than the entire Service mesh.

This is a great security feature provided by the mesh, but encryption won't stop a request from hitting our workload; it simply encrypts it. The next object we will discuss will add a level of security to a workload by requiring authentication before being allowed access.

## Request authentication and authorization policies

Security requires two pieces. First, the authentication piece, which is "who you are." The second piece is authorization, which is the actions that are allowed once authentication has been provided, or "what you can do."

`RequestAuthentication` objects are only one part required to secure a workload. To fully secure the workload, you need to create the `RequestAuthentication` object and an `AuthorizationPolicy`. The `RequestAuthentication` policy will determine what identities are allowed access to the workload, and the `AuthorizationPolicy` will determine what permissions are allowed.

A `RequestAuthorization` policy without an `AuthorizationPolicy` can lead to unintentionally allowing access to the resource. If you only create a `RequestAuthorization` policy, the access in *Table 16.4* shows who would be allowed access.

Token action	Access provided
Invalid token provided	Access will be denied
No token provided	Access will be granted
Valid token provided	Access will be granted

*Table 16.4: RequestAuthentication access*

As you can see, once we create a policy, any invalid JWT will be denied access to the workload, and any valid token will be allowed access to the workload. However, when no token is provided, many people think that access would be denied, but in reality, access would be allowed. A `RequestAuthentication` policy only verifies the tokens, and if no token is present, the `RequestAuthentication` rule will not deny the request.

An example manifest is shown below. We will use this manifest in the examples section of the chapter, but we wanted to show it in this section to explain the fields.

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: demo-requestauth
  namespace: demo
spec:
  selector:
    matchLabels:
      app: frontend
  jwtRules:
```

```
- issuer: testing@secure.istio.io
  jwksUri: https://raw.githubusercontent.com/istio/istio/release-1.11/
    security/tools/jwt/samples/jwks.json
```

This manifest will create a policy that configures a workload with the label matching `app=frontend` in the `demo` namespace to accept JWTs from the issuer `testing@secure.istio.io` with a URL to confirm the tokens at `https://raw.githubusercontent.com/istio/istio/release-1.11/security/tools/jwt/samples/jwks.json`.

This URL contains the key used to validate the tokens:

```
{ "keys": [ {"e": "AQAB", "kid": "DHFbpoIUqrY8t2zpA2qXfCmr5V05ZE4RzHU_-envvQ", "kt
y": "RSA", "n": "xAE7eB6qugXyCAG3yhh7pkDkT65pHmX-P7KfIupjf59vsdo91bSP9C8H07pSAGQ-
01MV_xFj9VswgsCg4R6otmg5PV2He951ZdHt0cU5DXIg_pbhLdKXbi66G1VeK6ABZOUW3WYtnNHD-
91gVuoeJT_DwtGGcp4ignkgXfk1Em4sw-4sfb4qdt5oLbyVpmW6x9cfa7vs2WTfURiCrBoUqgBo-
4WTiULmmHSGZH0jzwa8Wtrt0QGsAFjIbno85jp6MnGGGZPYZbDAa_b3y5u-YpW7ypZrvD8BgtKVjgtQ
gZhLAGezMt0ua3DRrWnKqTZ0BJ_EyxOGuHJrLsn00fnMQ" } ] }
```

When a token is presented, it will be verified that it came from the issuer defined in the `jwtRules` section of the `RequestAuthentication` object.

We'll walk through an example of how token authentication works in depth in the next chapter.

## Service entries

Once a workload is part of a Service mesh, its sidecar proxy will handle all outbound communication to services within the mesh. If the workload attempts to communicate with an external service that is not part of the mesh, this communication may fail if not properly configured. Fortunately, Istio provides mechanisms to define and manage external services, allowing workloads to communicate with services outside the mesh. One such mechanism is the `ServiceEntry` object, which allows you to define services that are external to the mesh and configure how these services should be accessed.

If you have a requirement for a workload in your Service mesh to communicate with a service outside of the Service mesh, you need to create an entry in the mesh for the external resource. This can be done in two ways, and the first method leads us to our next custom resource, the `ServiceEntry` object, which allows you to add external entries to the Service mesh. When you create a `ServiceEntry` for an external service, it will appear as if it was part of the actual Service mesh. This allows traffic to be routed to manually specified services from inside the Service mesh. Without a `ServiceEntry`, any attempt to communicate with the external resource would fail since Istio would attempt to look up the service in the Service mesh entries and it would fail to find the resource (since it is not part of the mesh).

To create a `ServiceEntry`, you need to create a new object that contains the hosts and ports for the external service. The example below will create a new entry that adds the host `api.foowidgets.com` on port `80`, using HTTP, to the Service mesh.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
```

```
name: api-server
namespace: sales
spec:
  hosts:
    - api.foowidgets.com
  ports:
    - number: 80
      name: http
      protocol: HTTP
```

ServiceEntries are a great resource to explicitly add external resources to a Service mesh. We mentioned that there are two ways to add external resources to the Service mesh, Service entries being one – and the other being the Sidecars object. The choice of which object to use is very specific to your own use-cases and organizational standards. Service entries are very specific and you must create an entry for each external resource you need to communicate with. Sidecars are different, and instead of defining what is external to the Service mesh, you define what is in the Service mesh.

## Sidecars

First, we know this can be confusing – this object is not the sidecar itself; it is an object that allows you to define what items your sidecar considers to be “in the mesh.” Depending on the size of your cluster, you may have thousands of services in the mesh and if you do not create a sidecar object, your Envoy sidecar will assume that your service needs to communicate with every other service.

Typically, you may only need your namespace to communicate with services in the same namespace or a small number of other namespaces. Since tracking every service in the mesh requires resources, it’s considered good practice to create a sidecar object to reduce the required memory in each Envoy sidecar.

To create a sidecar object that limits the services in your Envoy proxy, you would deploy the manifest shown below:

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: sales-sidecar
  namespace: sales
spec:
  egress:
    - hosts:
        - "./*"
        - "istio-system/*"
```

The spec in this manifest contains a list of hosts for the mesh, the ./\* references the namespace where the object was created, and all sidecars should contain the namespace where Istio was deployed, which would be `istio-system`, by default.

If we had three namespaces that needed to communicate across the mesh, we would simply need to add the additional namespaces to the hosts' entries:

```
apiVersion: networking.istio.io/v1beta1
kind: Sidecar
metadata:
  name: sales-sidecar
  namespace: sales
spec:
  egress:
    - hosts:
      - ./*
      - istio-system/*
      - sales2
      - sales3
```

Failing to limit the mesh objects may result in your Envoy sidecar crashloops due to resources. You may experience an **out of memory (OOM)** event, or simply crashloops that do not show any details of the root cause. If you experience these scenarios, deploying a sidecar object may resolve the issue.

## Envoy filters

Envoy filters provide you with the ability to create custom configurations that are generated by Istio. Remember that Pilot (part of `istiod`) is responsible for sidecar management. When any configuration is sent to Istio, Pilot will convert the configuration for Envoy to utilize. Since you are “limited” by the options in the Istio custom resource, you may not have all of the potential configuration options that are required for a workload, and that’s where Envoy filters come in.

Filters are very powerful, and potentially dangerous, configuration objects. They allow you to customize values that you cannot customize from a standard Istio object, allowing you to add filters, listeners, fields, and more. This brings a quote used in Spider-Man from the late Stan Lee to mind, “With great power comes great responsibility.” Envoy filters provide you with extended configuration options, but if a filter is misused, it could bring down the entire Service mesh.

Envoy filters are complex and, for the purposes of this book, are not a topic that needs deep understanding to understand Istio in general. You can read more about Envoy filters on the Istio site at <https://istio.io/latest/docs/reference/config/networking/envoy-filter/>.

## WASM plugins

Similar to Envoy filters, the `WasmPlugins` object is used to extend the capabilities of the Envoy sidecar. While they are similar in what they provide, they each provide different degrees of customization.

Typically, `WasmPlugins` are regarded as more straightforward to develop and implement, making them less complex and dangerous compared to Envoy filters. This simplicity, however, comes at the cost of reduced functionality relative to what Envoy filters can offer.

Envoy filters provide detailed, fine-grained control over the proxy settings, enabling the creation of more sophisticated operations than is possible with `WasmPlugins`. This level of control contributes to their complexity and the potential risk they pose if improperly configured or used, which could lead to disruptions within your Service mesh.

The choice between EnvoyFilters and `WasmPlugins` ultimately depends on your specific needs and preferences. It's important to consider various factors when deciding which option to adopt, weighing the trade-offs between ease of use, functionality, and the potential impact on your Service mesh.

## Deploying add-on components to provide observability

By now, you know how to deploy Istio and understand some of the most used objects, but you haven't seen one of the most useful features yet – observability. At the beginning of the chapter, we mentioned that observability is one of our favorite features provided by Istio, and in this chapter, we will explain how to deploy a popular Istio add-on called Kiali.

### Installing Istio add-ons

When you deploy Istio, you provide a Service mesh and all of the features to your developers. While this is powerful by itself, you need to add a few extra components to truly provide a complete solution. There are four add-ons that you should add to your Service mesh – while there are alternatives to some of the solutions, we are using the most commonly used add-ons, specifically:

- Prometheus
- Grafana
- Jaeger
- Kiali (which we will cover in the next section)

We have discussed Prometheus and Grafana in previous chapters, but Jaeger is a new component that we have not mentioned before.

Jaeger is an open-source offering that provides tracing between services in Istio. Tracing may be a new term to some readers. At a high level, traces are a representation of the execution path to a service. These allow us to view the actual path of the communication between services, providing an easy-to-understand view that provides metrics about performance and latency, allowing you to resolve issues quickly.

To deploy all of the add-ons, we have included a script to deploy Prometheus in the `chapter16/add-ons` directory, called `deploy-add-ons.sh`. Execute the script to deploy the add-ons in your cluster.

Many add-on example deployments do not maintain state, so for our deployment we have added persistency, leveraging the provisioner built into KinD, by adding persistent disks to each deployment.

The script executes the steps outlined below:

1. Deploys each add-on using standard Kubernetes manifests in the `istio-system` namespace.
  - a. Each deployment creates a PVC and mounts it as the data location to maintain persistency across reboots.

2. Finds your host's IP address to create new Gateway and VirtualService entries for each add-on.
3. Creates a shared Istio Gateway that will be used by each of the add-ons.
4. Creates VirtualServices that contain your `nip.io` URLs. The three URLs that will be created are:
  - a. `prom.<Host IP>.nip.io`
  - b. `grafana.<Host IP>.nip.io`
  - c. `kiali.<Host IP>.nip.io`

The final output from the script will contain the URLs that were created for you.

With the add-ons deployed, we can move on to the next section, which will cover the main tool for observability, Kiali.

## Installing Kiali

Kiali provides a powerful management console for our Service mesh. It provides graphical views of our services, pods, traffic security, and more. Since it's a very useful tool for both developers and operators, the remainder of this chapter will focus on deploying and using Kiali.

There are a few options to deploy Kiali, but we will use the most common installation method, using a Helm chart. To deploy the chart and create the required objects to access the Kiali UI, we have provided a script in the `chapter16/kiali` directory called `deploy-kiali.sh`. Execute the script to deploy Kiali.

The script will deploy Kiali into your cluster, in the `istio-system` namespace, pre-configured to integrate with the add-ons we deployed in the previous section. It will also expose Kiali's UI using a `nip.io` URL, which will be provided at the end of the script execution.

This deploys an anonymous access dashboard; however, Kiali can accept other authentication mechanisms to secure the dashboard. In the next chapter, we will modify the Kiali deployment to accept JWTs, using OpenUnison as the provider.

## Deploying an application into the Service mesh

We could define the components and objects of Istio all day, but if you are like us, you will find examples and use-cases more beneficial to understanding advanced concepts like the features provided by Istio. In this section, we will explain many of the custom resources in detail, providing examples that you can deploy in your KinD cluster.

## Deploying your first application into the mesh

Finally! We have Istio and the add-on components installed and we can move on to installing a real application in the Service mesh to verify everything is working.

For this section, we will deploy an example application from Google called the Boutique app. In the next chapter, we will deploy a different application and explain all of the details and communication between the services, but the Boutique app is a great application to test out the mesh before we get into that level of information.

In the chaper16/example-app directory, there is an installation script called `deploy-example.sh` that will deploy the application to the cluster. It will install the base application and the required Istio objects to make the application accessible to the outside world. The script execution is detailed below:

1. A new namespace named `demo` is created with a label containing `istio-injection=enabled`.
2. Using the `kubernetes-objects.yaml` manifest, the base application will be deployed.
3. The Istio objects will be created using templates to create the names in the `nip.io` domain for easy access to the application. The objects created in Istio include the `Gateway` and `VirtualService` objects.
4. The created `nip.io` domain will be output to the screen. On our server, it was `kiali.10.3.1.248.nip.io`.

Once executed, you will have a working `demo` application in the `demo` namespace. We will use this application to demonstrate the observability features of Istio and Kiali.

Quickly verify that the application and Istio objects have been deployed correctly by the script by using a browser to open the `nip.io` URL. You should see the Kiali home screen, which we will discuss in the next section.

## Using Kiali to observe mesh workloads

Kiali provides observability in your Service mesh. It provides a number of advantages to you and your developers, including a visual map of the traffic flow between objects, verifying mTLS between the services, logs, and detailed metrics.

### The Kiali overview screen

If you navigate to the homepage of Kiali, by using the URL provided when you executed the `create-ingress` script, this will open the Kiali overview page where you will see a list of namespaces in the cluster.

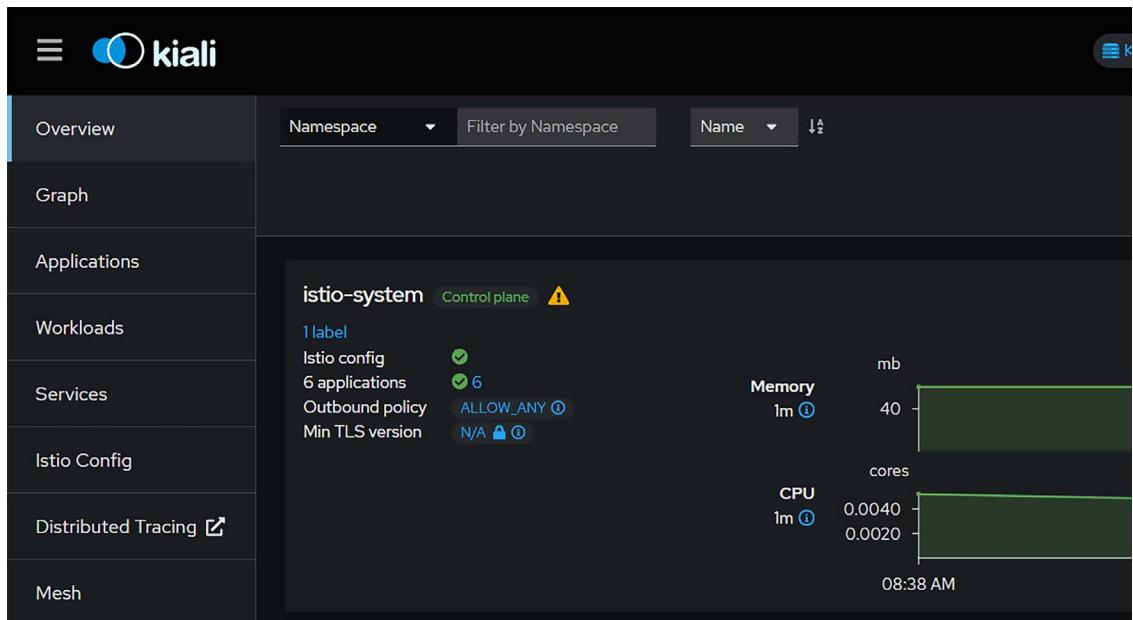


Figure 16.3: The Kiali homepage

Kiali will show all namespaces in the cluster, even if they do not have Istio enabled. In our current deployment, it will show all namespaces, regardless of any RBAC that has been implemented since it's running without any authentication. As mentioned in the *Installing Kiali* section, we will secure Kiali with JWTs in the next chapter.

## Using the Graph view

The first part of the dashboard that we will visit is the **Graph** view, which provides a graphical view of our application. Initially, it may look like a simple static graphical representation of the objects that make up the workload, but this is simply the default view when you open the **Graph** view; it isn't limited to a simple static view, as you will see in this section.

Since we deployed the example application into the `demo` namespace, scroll down a bit and look for the block that contains the `demo` namespace, click the three dots on the tile, and then select **Graph**:

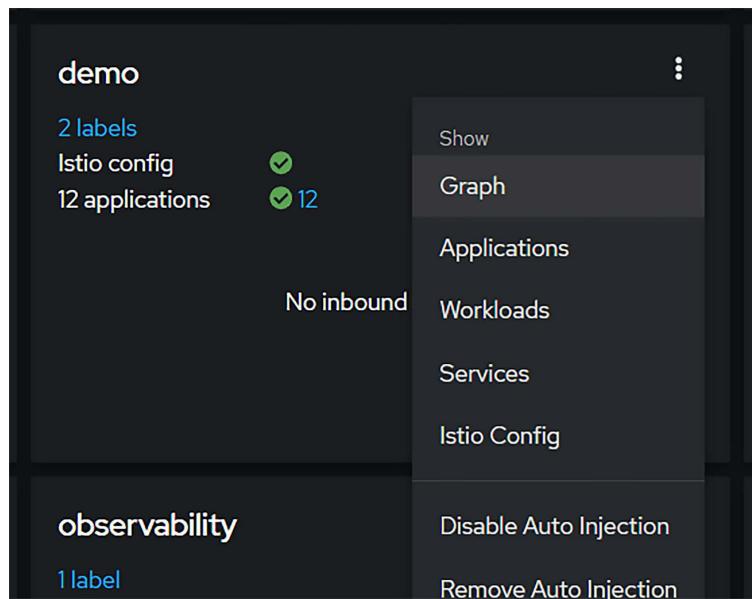


Figure 16.4: Using Kiali to show a graph of a namespace

This will take you to a new dashboard view that shows the demo application objects:

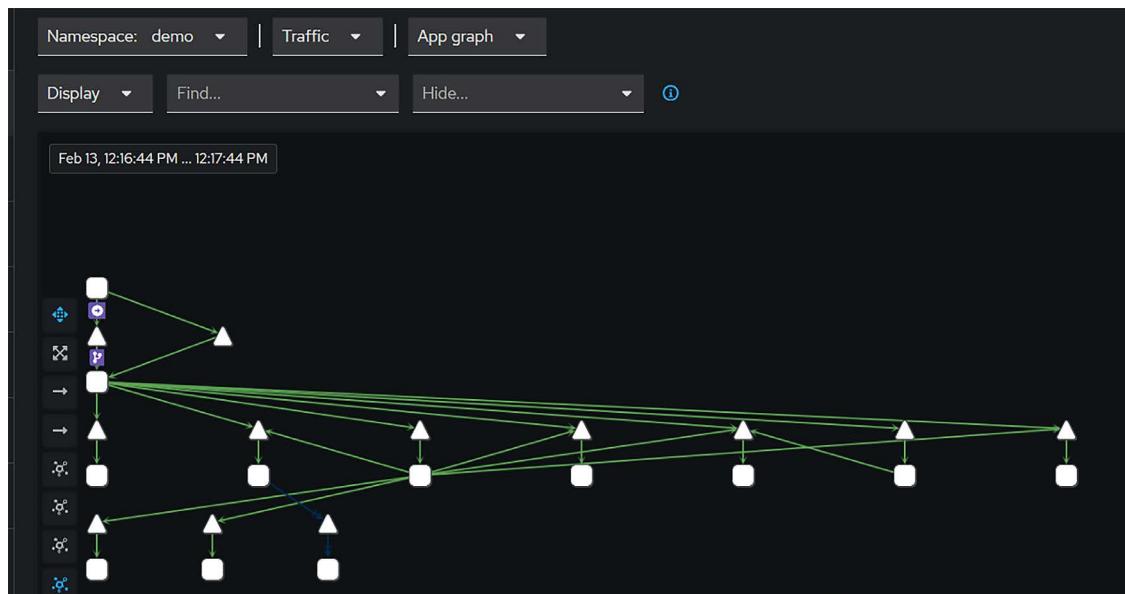


Figure 16.5: Kiali graph example

There are a lot of objects on the graph, and if you are new to Kiali, you may be wondering what each of the icons represents. Kiali provides a legend to help you identify what role each icon plays.

If you click on the icon in the lower left-hand section of the **graph** pane, you will see the legend icon. Click it to see an explanation of each icon – an abbreviated legend list is shown below:

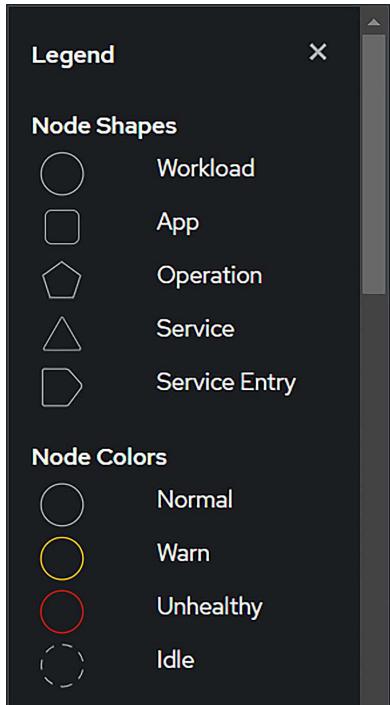


Figure 16.6: Kiali graph legend example

By default, this view only shows the paths between the application objects in a static view. However, you are not limited only to the static view – this is where Kiali starts to shine. We can actually enable a live traffic view, enabling us to watch the traffic flow for all requests.

To enable this option, click the **Display** option that is just above the Graph view, and in the list of options, enable traffic animation by checking the box, as shown in *Figure 16.7*.

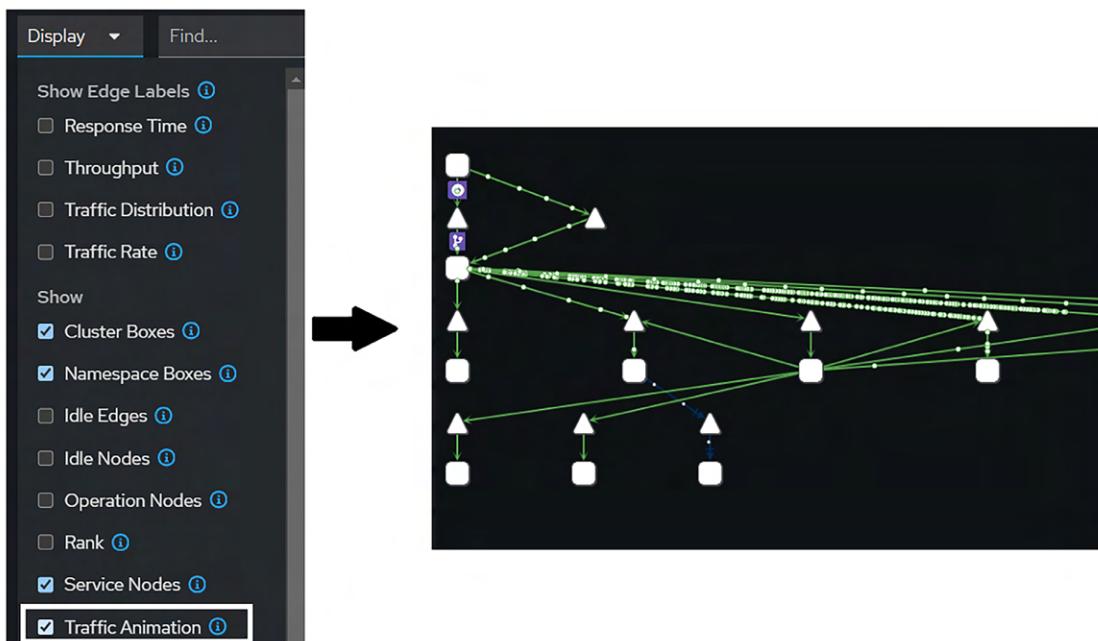


Figure 16.7: Enabling traffic animation

It's difficult to display in a static image, but once you have enabled the **Traffic Animation** option, you will see the flow of all requests in real time.

You are not limited to only traffic flow animations; you can use the **Display** option to enable a number of other options in the **Graph** view, including items like response time, throughput, traffic rate, and security.

In *Figure 16.8*, we have enabled throughput, traffic distribution, traffic rate, and security:

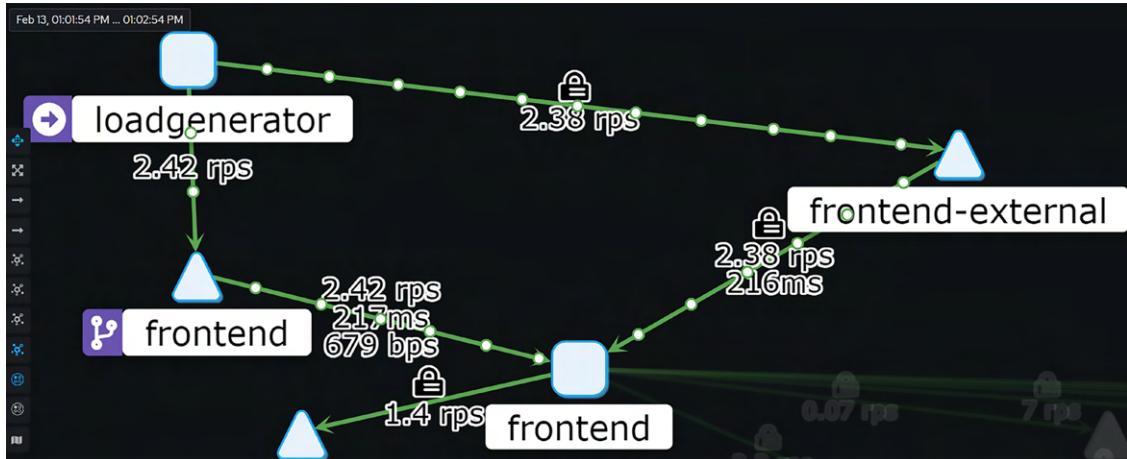


Figure 16.8: Kiali graph display options

As you can see in the image, the lines between objects now include additional information, including:

- A lock, which confirms that the communication is encrypted via the sidecar and mTLS
- RPS, which is the requests per second

As you can see, the Kiali **Graph** view is a powerful tool for observing the end-to-end communication for your workload. This is just one of the additional benefits of using a Service mesh. The observability that a mesh provides is an incredibly valuable tool for finding issues that would have been very difficult to uncover in the past.

We are not limited to only the **Graph** view; we also have three additional views that offer additional insight into the application. On the left-hand side of the Kiali dashboard, you will see the other three views, **Applications**, **Workloads**, and **Services**. You will also notice that there is one other option, **Istio Config**, which allows you to view the objects in the namespace that control the Istio features for the namespace.

## Using the Applications view

The Applications view shows you the details for the workloads that have the same labeling, allowing you to break down the view into smaller sections.

Using the Boutique Applications view that we have opened in Kiali, click on the **Applications** link in the left-hand options. This will take you to the overview page for the applications, broken down by labels.

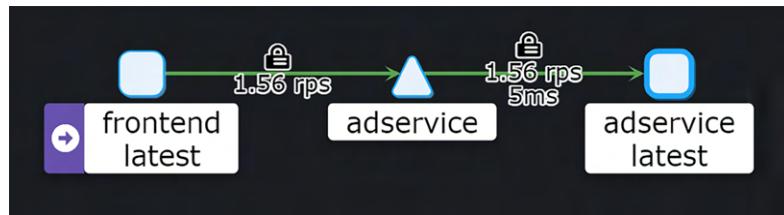
App Name	Health	Name	Namespace	Labels
	✓	A adservice	NS demo	app=adservice
	✓	A cartservice	NS demo	app=cartservice
	✓	A checkoutservice	NS demo	app=checkoutservice
	✓	A currencybservice	NS demo	app=currencybservice
	✓	A emailservice	NS demo	app=emailservice

Figure 16.9: Kiali Applications view

Each of the applications can provide additional information by clicking the name of the service. If we were to click the **adservice** application, Kiali would open a page providing an overview of what the **adservice** application interacts with. For each application, you can also look at the overview, traffic, inbound and outbound metrics, and traces.

The overview page presents you with a dedicated view of the objects that communicate with **adservice**. We saw a similar communications view in the **Graph** view, but we also saw every other object – including objects that have nothing to do with **adservice**.

The Applications view will streamline what we can see, making it easier to navigate the application.



*Figure 16.10: Simplified communication view using the Applications view*

As you can see, the Applications view contains the components from the Graph view. The communications path that involves **adservice** starts with the frontend pod, which targets the **adservice** service, which ultimately routes the traffic to the **adservice** pod.

We can see additional details in the application by clicking on one of the tabs at the top of the **Applications** view. The first tab next to the overview is the **Traffic** tab, which provides you with a view of the traffic for the application.

Inbound Traffic			
Status	Name	Rate	Percent Success
✓	A frontend	1.33rps	100.0%
No Outbound Traffic			

*Figure 16.11: Viewing application traffic*

The **Traffic** tab will show inbound and outbound traffic to the application. In the **adservice** example from the Boutique store, we can see that **adservice** has received inbound requests from the frontend. Below the inbound traffic, we can see the outbound traffic and, in our example, Kiali is telling us that there is no outbound traffic. As we can see in the overview in *Figure 16.10*, the **adservice** pod does not have any object that it connects to; therefore, we would not have any traffic to view. To get additional details on the traffic, you can click on the **View Metrics** link under **Actions** – this action is the same as if you were to click the **Inbound Metrics** tab.

The **Inbound Metrics** tab will provide you with additional details about the incoming traffic. *Figure 16.12* shows an abbreviated example for the **adservice** traffic.



Figure 16.12: Viewing inbound metrics

The inbound metrics will display a number of different metrics, including request volume, request duration, request and response size, request and response throughput, gRPC received and sent, TCP opened and closed, and TCP received and sent. This page will update in real time, allowing you to view the metrics as they are captured.

Finally, the last tab will allow you to look at the traces for the **adservice** application. This is why we deployed Jaeger in our cluster when we installed Istio. Tracing is a fairly complex topic and is outside the scope of this chapter. To learn more about tracing using Jaeger, head over to the Jaeger site at <https://www.jaegertracing.io/>.

## Using the Workloads view

The next view we will discuss is the **Workloads** view, which breaks down the views to the workload type, like deployments. If you click on the **Workloads** link in Kiali, you will be taken to a breakdown of the Boutique workloads.

Workload Name	Filter by Workload Na...				
H...	Name	Na...	Type	Labels	Details
✓	W adservice	NS demo	Deployment	app=adservice	🔴 Missing Version ⓘ
✓	W cartservice	NS demo	Deployment	app=cartservice	🔴 Missing Version ⓘ
✓	W checkoutservice	NS demo	Deployment	app=checkoutservice	🔴 Missing Version ⓘ
✓	W currencyservice	NS demo	Deployment	app=currencyservice	🔴 Missing Version ⓘ

Figure 16.13: The Workloads view

You may notice that there is a warning under the **Details** column that tells us we are missing a version of the deployments. This is one of the features of this view. It will offer details like a workload not being assigned a version, which is not an issue for standard functionality in the mesh, but it will limit the use of certain features, like routing and some telemetry. It's a best practice to always version your application, but for the example, Boutique from Google, they do not include a version in the deployments.

The **Workloads** view offers some of the same details as the **Applications** view, including traffic, inbound metrics, outbound metrics, and tracing – however, in addition to these details, we can now view the logs and details about Envoy.

If you click on the **Logs** tab, you will see the logs for the **adservice** container.

```

P adservice-cf48dc6df-6x9xt ▾ Show... Hide... ⓘ spans 3000 lines ▾
C ☐ istio-proxy ☐ server Copy Expand ⋮

① [2024-02-13T18:40:26.948Z] "POST /hipstershop.AdService/GetAds HTTP/2" 200 - via_upstream "-" 14 135 0 0 "-" "grpc-go/1.59.0" "56d"
{"instant":{"epochSecond":1707849626,"nanoOfSecond":948935860},"thread":"grpc-default-executor-4","level":"INFO","loggerName":"hipster"
① [2024-02-13T18:40:27.368Z] "POST /hipstershop.AdService/GetAds HTTP/2" 200 - via_upstream "-" 15 80 1 0 "-" "grpc-go/1.59.0" "e60"
{"instant":{"epochSecond":1707849627,"nanoOfSecond":369461730},"thread":"grpc-default-executor-4","level":"INFO","loggerName":"hipster"
① [2024-02-13T18:40:28.380Z] "POST /hipstershop.AdService/GetAds HTTP/2" 200 - via_upstream "-" 18 78 1 0 "-" "grpc-go/1.59.0" "59d"
 {"instant":{"epochSecond":1707849628,"nanoOfSecond":380779688}),"thread":"grpc-default-executor-4","level":"INFO","loggerName":"hipster"
① [2024-02-13T18:40:28.584Z] "POST /hipstershop.AdService/GetAds HTTP/2" 200 - via_upstream "-" 18 78 1 0 "-" "grpc-go/1.59.0" "72f"
 {"instant":{"epochSecond":1707849628,"nanoOfSecond":585049188}),"thread":"grpc-default-executor-4","level":"INFO","loggerName":"hipster"

```

Figure 16.14: Viewing the container logs

This is a real-time view of the logs that are being generated by the **adservice** container. In this view, you can create a filter to show or hide certain keywords, scroll back to previous events, change the default buffer size from 100 lines, copy the logs to your clipboard, or enter a fullscreen log view. Many users find this tab very useful since it doesn't require them to use `kubectl` to look at the logs; they can simply open up Kiali in a browser and quickly view the logs in the GUI.

The last tab we will discuss is the **Envoy** tab, which provides additional details about the Envoy sidecar. The details in this tab are extensive – it contains all of the mesh objects that you have included in the namespace (recall that we created a sidecar object to limit the objects to only the namespace and the `istio-system` namespace), all of the listeners, routes, the bootstrap configuration, config, and metrics.

By this point in the chapter, you can probably see how Istio would require its own book to cover all of the base components. All of the tabs in the **Envoy** tab provide a wealth of information, but it gets very detailed and we can't fit them all in this chapter, so for the purposes of this chapter, we will only discuss the **Metrics** tab.

Clicking on the **Metrics** tab, you will see metrics pertaining to the uptime of Envoy, the allocated memory, heap size, active upstream connections, upstream total requests, downstream active connections, and downstream HTTP requests.

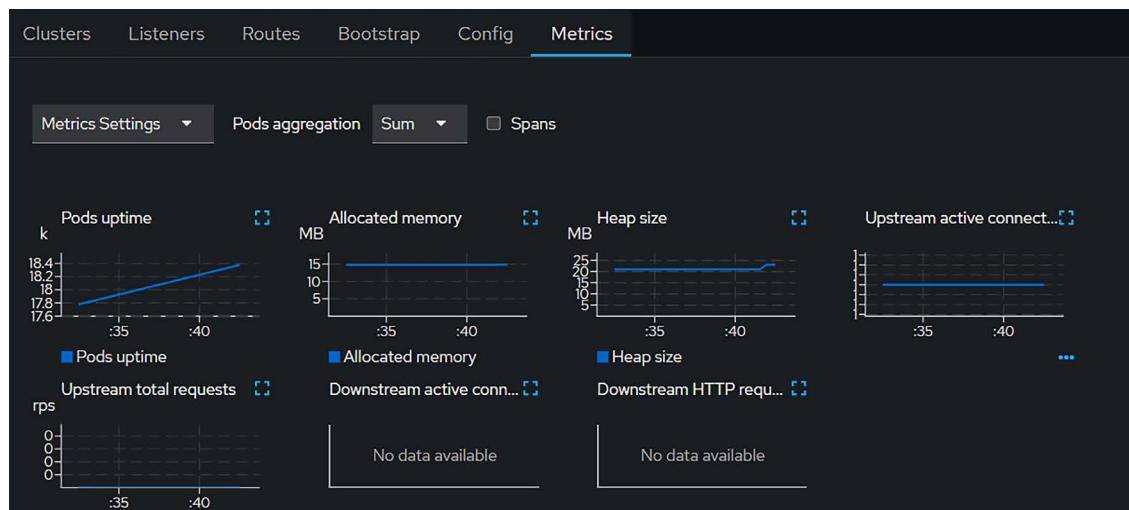


Figure 16.15: Envoy metrics

Like most metrics, these will be beneficial if you experience issues with the Envoy proxy container. The uptime will let you know how long the pod has been running, the allocated memory tells you how much memory has been allocated to the pod, which may help to identify why an OOM condition occurred, and active connections will identify if the service has issues if the connection count is lower than expected, or at zero.

## Using the Services view

Finally, we will discuss the last view for the application, the Services view. Just as the name implies, this will provide a view of the services that are part of the workload. You can open the Services view by clicking on the **Services** option in Kiali.

Service Name	Name	Namesp...	Labels	Config...	Details
adservice	adservice	NS demo		✓	
cartservice	cartservice	NS demo		✓	
checkoutservice	checkoutservice	NS demo		✓	
currencyservice	currencyservice	NS demo		✓	
emailservice	emailservice	NS demo		✓	

Figure 16.16: The Services view

Similar to the other views, this will provide the names of the services and the health of each of the services. If you click on any individual service, you will be taken to the details of the service. If you were to click **adservice**, you would be taken to the overview for the service.

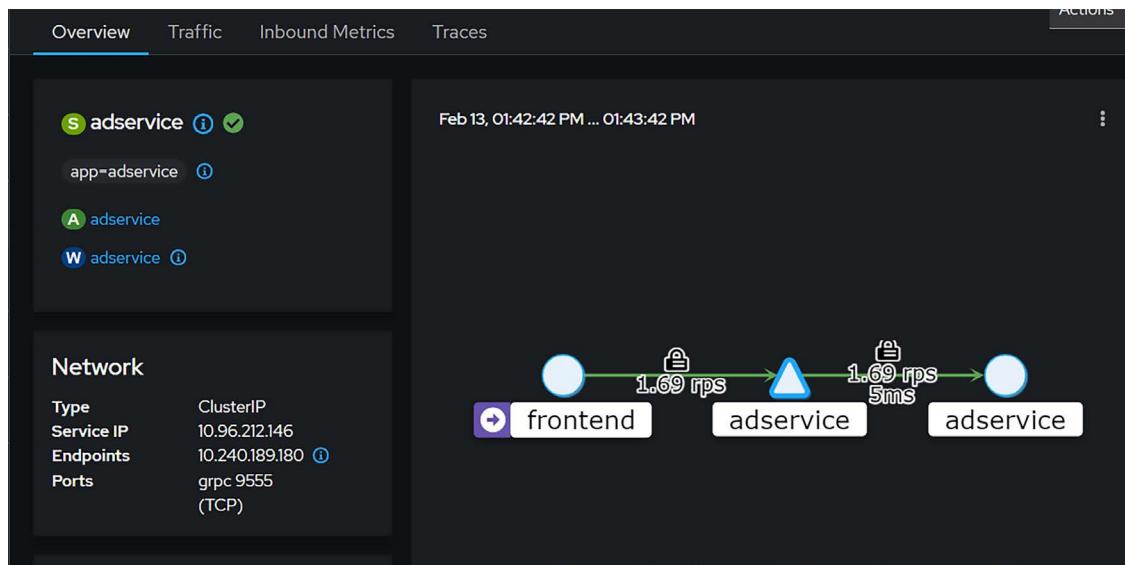


Figure 16.17: The services overview

The **Overview** page should have some objects familiar to you. Just like the other views, it provides a view of just the objects that communicate with **adservice**, and it has tabs for traffic, inbound metrics, and traces – however, in addition to these, it also shows the network information for the service. In our example, the service has been configured to use a **ClusterIP** type, the service IP assigned is **10.110.47.79**, it has an endpoint of **10.240.189.149**, and it has the gRPC TCP port exposed on port **9555**.

This is information you could retrieve using `kubectl`, but for many people, it's quicker to grab the details from the Kiali dashboard.

## The Istio Config view

The last view we have is not related to the workload in particular. Instead, it's a view for the Istio config for the namespace. This view will contain the Istio objects you have created. In our example, we have two objects, the gateway and the virtual service.

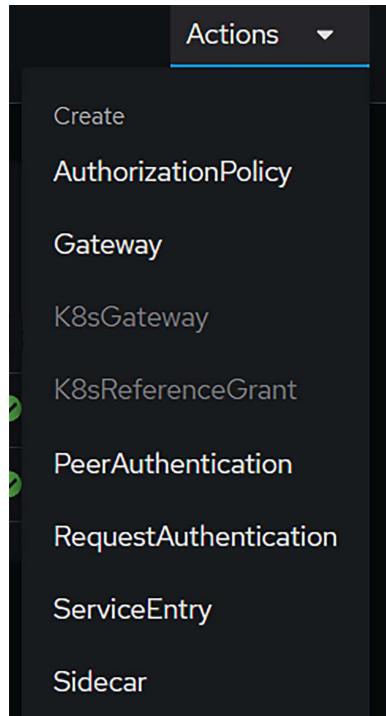
The screenshot shows the **Istio Config** view for the **demo** namespace. It includes a **Namespace** dropdown set to **demo** and an **Actions** dropdown. The main area displays a table of Istio objects:

Type	Name	Namespace	Type	Configuration
Gateway	<a href="#">G frontend-gateway</a>	NS demo	Gateway	✓
VirtualService	<a href="#">VS frontend-vs</a>	NS demo	VirtualService	✓

Figure 16.18: The Istio Config view

You can view the YAML for each object by clicking the name. This allows you to directly edit the object in the Kiali dashboard. Any changes that are saved will edit the object in the cluster, so be careful if you are using this method to modify the object.

This view offers one addition that the other views do not – the ability to create a new Istio object using a wizard. To create a new object, click the **Actions** dropdown in the upper right-hand corner of the Istio config view. This will bring up a list of objects that you can create, as shown in *Figure 16.19*.



*Figure 16.19: Istio object creation wizard*

As you can see in the figure, Kiali provides a wizard to create 6 Istio objects including **AuthorizationPolicies**, **Gateways**, **PeerAuthentication**, **RequestAuthentication**, **ServiceEntries**, and **Sidecars**.

Each option has a wizard to guide you through the specific requirements for that object. For example, we could create a sidecar using the wizard, as shown in *Figure 16.20*.

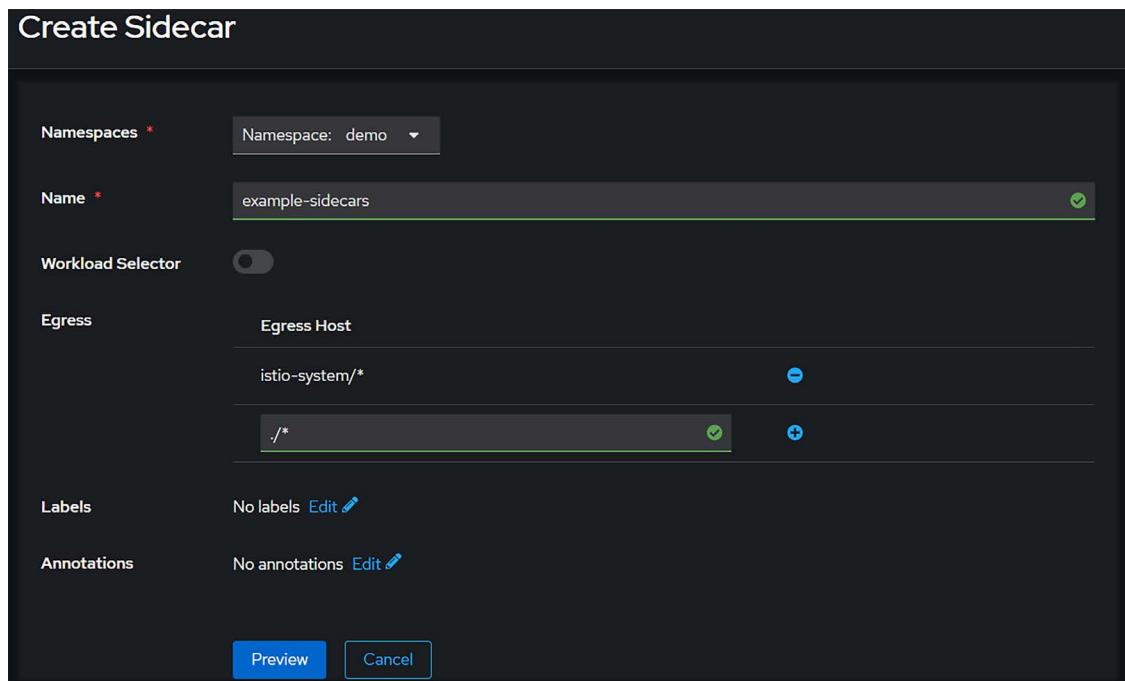


Figure 16.20: Using the Istio object wizard

Once all fields have been entered correctly, you can click **Preview**, which will take you to the next screen, where you will see the object YAML source, as shown in *Figure 16.21*.

The screenshot shows the 'Preview new istio objects' screen with the title 'Sidebar'. It includes 'Copy' and 'Download' buttons. The main area displays the generated YAML code for the Sidecar:

```
example-sidecars
1 apiVersion: networking.istio.io/v1beta1
2 kind: Sidecar
3 metadata:
4   name: example-sidecars
5   namespace: demo
6   labels: {}
7   annotations: {}
8 spec:
9   egress:
10     - hosts:
11       - istio-system/*
12 
```

At the bottom are 'Create' and 'Cancel' buttons.

Figure 16.21: Wizard source YAML

If it looks good, click **Create** to create the new object.

The wizards are a good tool for people who are new to Istio, but be careful not to rely on them too much. You should always understand how to create a manifest for all of your objects. Creating objects using wizards like these can lead to problems down the road without the knowledge of how the object works or is created.

In the next section, we will introduce where Istio is heading in the future. As powerful as the sidecar is, it has its limitations, and it requires additional resources for each pod in the mesh. In 2023, Istio introduced a new concept called ambient mesh as an early access feature, which removed the requirement for the Istio sidecar.

## The future: Ambient mesh

Today, Service meshes, such as Istio, depend on sidecar proxies connected to every service instance to handle traffic, security measures, and metric collection. Although this approach works well, it leads to extra resource usage and complexity, particularly in deployments in larger clusters.

In this chapter, we mentioned sidecars a lot – they're the heart of the mesh, providing the layer that removes all of the complexities of using mesh features without requiring code changes to our applications.

Ambient mesh marks a significant change in the Service mesh design, attempting to make it easier to add Service mesh features to an already complicated system without the need for sidecar proxies for every service. Its goal is to cut down on the extra work and complexity while keeping the main advantages of a Service mesh, including monitoring, security, and traffic management.

As of Istio 1.20, the ambient mesh has the **Alpha** status. The main reason that we haven't added a chapter on using an ambient mesh is due to potential changes that will likely occur between the current alpha stage and when it goes to **general availability (GA)**. However, since it is a giant leap forward and a major change in design, we wanted to bring it to your attention. You can read more about getting started with ambient mesh on Istio's site here: <https://istio.io/latest/docs/ops/ambient/getting-started/>.

Reading the docs on Istio's site will provide some great examples of how the ambient mesh deploys and works. Since many readers will be new to Istio, jumping into the docs for an overview may be a little much, so we wanted to provide our view on the key points of what the ambient mesh means to us.

As we mentioned, ambient mesh addresses a number of issues by removing the tasks that are usually managed by sidecar proxies. Rather than attaching a proxy to each service instance, the ambient mesh will integrate these functions directly into the network or in a common proxy layer. This design seeks to simplify processes and lessen the resources needed compared to the traditional sidecar method. This provides a number of advantages, including:

- **Preservation of Essential Features:** Despite its unique architecture, the ambient mesh will continue to deliver the base capabilities found in current Service meshes, including secure communications between services, traffic control, and the ability to monitor components.
- **Streamlined, Efficient Deployments:** The ambient mesh eliminates the necessity for individual sidecar proxies, streamlining the setup and management of Service meshes. This will facilitate easier adoption and maintenance, especially for organizations with complex microservices structures.

- **Improved Resource Utilization:** By minimizing the CPU and memory demands for service-to-service interactions, the ambient mesh allows the efficient use of resources.
- **Performance Enhancement:** The ambient mesh enhances system performance by optimizing the routes used for service communication, reducing delays and boosting efficiency.

Imagine the resource and complexity savings in a large cluster where you may have thousands, or tens of thousands, of services running. Currently, that would require as many proxy instances running, each adding an extra layer in the communication and using its own resources – using extra CPU and RAM that are not actually “required” for the base applications. The ambient mesh will save you money by reducing the required resources, and by simplifying the architecture, it should be easier to find issues when an application is not behaving as expected.

We hope that this chapter has provided a useful introduction to Istio and what’s in store for Istio’s future. In the next chapter, we will dive deeper into running applications in an enterprise using Istio.

## Summary

In this chapter, we introduced you to the Service mesh world, using the popular open-source project Istio. In the first section of the chapter, we explained some of the advantages of using a Service mesh, which included security and observability for mesh services.

The second section of the chapter detailed the installation of Istio and the different installation profiles that are available. We deployed Istio into our KinD clusters and we also removed NGNIX to free up ports 80 and 443 to be used by Istio’s ingress gateway. This section also included the objects that are added to a cluster once you deploy Istio. We covered the most common objects using example manifests that reinforce how to use each object in your own deployments.

To close out the chapter, we detailed how to install Kiali, Prometheus, and Jaeger to provide powerful observability in our Service mesh. We also explained how to use Kiali to look into an application in the mesh to view the application metrics and logs.

In the next chapter, we will deploy a new application and bind it to the Service mesh, building on many of the concepts that were presented in this chapter.

## Questions

1. What Istio object(s) is used to route traffic between multiple versions of an application?
  - a. Ingress rule
  - b. VirtualService
  - c. DestinationRule
  - d. You can’t route traffic between multiple versions, only a single instance
2. What tool(s) are required to provide observability in the Service mesh?
  - a. Prometheus
  - b. Jaeger

- c. Kiali
  - d. Kubernetes Dashboard
3. True or false: Istio features require developers to change their code to leverage features like mutual TLS and authorization.
- a. True
  - b. False
4. Istio made the control plane easier to deploy and configure by merging multiple components into a single executable called:
- a. Istio
  - b. IstioC
  - c. istiod
  - d. Pilot

## Answers

1. b - VirtualService and c - DestinationRule
2. a Prometheus and c - Kiali
3. b - False
4. c - Pilot



# 17

## Building and Deploying Applications on Istio

In the previous chapter, we deployed Istio and Kiali into our cluster. We also deployed an example application to see how the pieces fit together. In this chapter, we’re going to look at what it takes to build applications that will run on Istio. We’ll start by examining the differences between microservices and monolithic applications. Then, we’ll deploy a monolithic application on Istio and move on to building microservices that will run on Istio. This chapter will cover the following main topics:

- Comparing microservices and monoliths
- Deploying a monolith
- Building a microservice
- Do I need an API gateway?

Once you have completed this chapter, you’ll have a practical understanding of the difference between a monolith and a microservice, along with the information you’ll need to determine which one is best for you, and you will also have deployed a secured microservice in Istio.

### Technical requirements

To run the examples in this chapter, you’ll need:

- A running cluster with Istio deployed, as outlined in *Chapter 16, An Introduction to Istio*.
- Scripts from this book’s GitHub repository.

You can access the code for this chapter by going to this book’s GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter17>.

## Comparing microservices and monoliths

Before we dive too deeply into code, we should spend some time discussing the differences between microservices and monolithic architecture. The microservices versus monolithic architecture debate is as old as computing itself (and the theory is probably even older). Understanding how these two approaches relate to each other and your problem set will help you decide which one to use.

## My history with microservices versus monolithic architecture

Before we get into the microservices versus monoliths discussion, I wanted to share my own history. I doubt it's unique, but it does frame my outlook on the discussion and adds some context to the recommendations in this chapter.

My introduction to this discussion was when I was a computer science student in college and had started using Linux and open source. One of my favorite books, *Open Sources: Voices from the Open Source Revolution*, had an appendix on the debate between Andrew Tanenbaum and Linus Torvalds on microkernels versus monolithic kernels. Tanenbaum was the inventor of Minix, and a proponent of a minimalist kernel, with most of the functionality in user space. Linux, instead, uses a monolithic kernel design, where much more is done in the kernel. If you've ever run `modprobe` to load a driver, you're interacting with the kernel! The entire thread is available at <https://www.oreilly.com/openbook/opensources/book/appa.html>.

Linus' core argument was that a well-managed monolith was much easier to maintain than a microkernel.

Tanenbaum instead pointed to the idea that microkernels were easier to port and that most "modern" kernels were microkernels. Windows (at the time, Windows NT) is probably the most prevalent microkernel today. As a software developer, I'm constantly trying to find the smallest unit I can build. The microkernel architecture really appealed to that aspect of my talents.

At the same time, I was starting my career in IT, primarily as a Windows developer in the data management and analysis space. I spent most of my time working with **ASP (Active Server Pages**, Microsoft's version of PHP), Visual Basic, and SQL Server. I tried to convince my bosses that we should move off of a monolithic application design to use **MTS (Microsoft Transaction Server)**. MTS was my first exposure to what we would call today a distributed application. My bosses and mentors all pointed out that our costs, and so our customers' costs, would go through the roof if we injected the additional infrastructure for no benefit other than a cleaner code base. There was nothing we were working on that couldn't be accomplished with our tightly bound trio of ASP, Visual Basic, and SQL Server at a much lower cost.

I later moved from data management to identity management. I also switched from Microsoft to Java. One of my first projects was to deploy an identity management vendor's product that was built using a distributed architecture. At the time, I thought it was great, until I started trying to debug issues and trace down problems across dozens of log files. I quickly started using another vendor's product that was built as a monolith. Deployments were slow, as they required a full recompile, but otherwise, management was much easier, and it scaled every bit as well. We found that a distributed architecture didn't help because identity management was done by such a centralized team that having a monolith didn't impact productivity or management. The benefits of distributing implementation just didn't outweigh the additional complexity.

Fast forward to the founding of Tremolo Security. This was in 2010, so it was before Kubernetes and Istio came along. At the time, virtual appliances were all the rage! We decided OpenUnison would take the monolithic approach because we wanted to make it easier to deploy and upgrade. In *Chapter 6, Integrating Authentication into Your Cluster*, we deployed OpenUnison with some Helm charts to layer on different configurations. How much harder would it have been had there been an authentication service to install, a directory service, a just-in-time provisioning service, etc.? It made for a much simpler deployment having one system.

With all that said, it's not that I'm anti-microservice—I'm not! When used correctly, it's an incredibly powerful architecture used by many of the world's largest companies. I've learned through the years that if it's not the right architecture for your system, it will considerably impact your ability to deliver. Now that I've filled you in on my own journey through architectures, let's take a deeper look at the differences between microservices and monoliths.

## Comparing architectures in an application

First, let's talk about what these two architecture approaches each do in a common example application, a storefront.

### Monolithic application design

Let's say you have an online store. Your store will likely need a product lookup service, a shopping cart, a payment system, and a shipping system. This is a vast oversimplification of a storefront application, but the point of the discussion is how to break up development and not how to build a storefront. There are two ways you could approach building this application. The first is you could build a monolithic application where all the code for each service is stored and managed in the same tree. Your application infrastructure would probably look something like this:

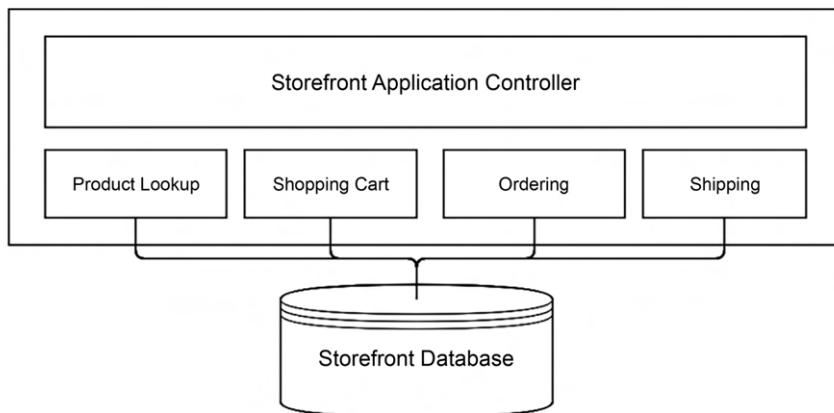


Figure 17.1: Monolithic application architecture

In our application, we have a single system with multiple modules. Depending on your programming language of choice, these could be classes, structs, or other forms of code module. A central application manages the user's interaction with this code. This would likely be a web frontend with the modules being server-side code, written up as web services or a post/response-style app.

Yes, web services can be used in a monolith! These modules likely need to store data, usually in some kind of a database. Whether it's a relational database, a document database, or a series of databases isn't really important.

The biggest advantage to this monolithic architecture is it's relatively simple to manage and have the systems interact with each other. If the user wants to do a product search, the storefront will likely just execute some code like the following:

```
list_of_products = products.search(search_criteria);
display(list_of_products);
```

The application code only needs to know the interface of the services it's going to call. There's no need to "authenticate" that call from the application controller to the product directory module. There's no concern with creating rate-limiting systems or trying to work out which version of the service to use. Everything is tightly bound. If you make an update to any system, you know pretty quickly if you broke an interface, since you're likely using a development tool that will tell you when module interfaces break. Finally, deployment is usually pretty simple. You upload your code to a deployment service (or create a container... this is a Kubernetes book!).

What happens if you need to have one developer update your ordering system while another developer updates your payment system? They each have their own copies of the code that need to be merged. After merging, the changes from both branches need to be reconciled before deployment. This may be fine for a small system, but as your storefront grows, this can become cumbersome to the point of being unmanageable.

Another potential issue is, what if there's a better language or system to build one of these services in than the overall application? I've been on multiple projects over the years where Java was a great choice for certain components, but C# had better APIs for others. Maybe one service team was built around Python and another on Ruby. Standardization is all well and good, but you wouldn't use the butt end of a screwdriver to drive in a nail for the sake of standardization, would you?

This argument doesn't pertain to frontend versus backend. An application with a JavaScript frontend and a Golang backend can still be a monolithic application. Both the Kubernetes Dashboard and Kiali are examples of monolithic applications built on service APIs across different languages. Both have HTML and JavaScript frontends, while their backend APIs are written in Golang.

## Microservices design

What if we broke these modules up into services? Instead of having one single source tree, we would break our application up into individual services like the following:

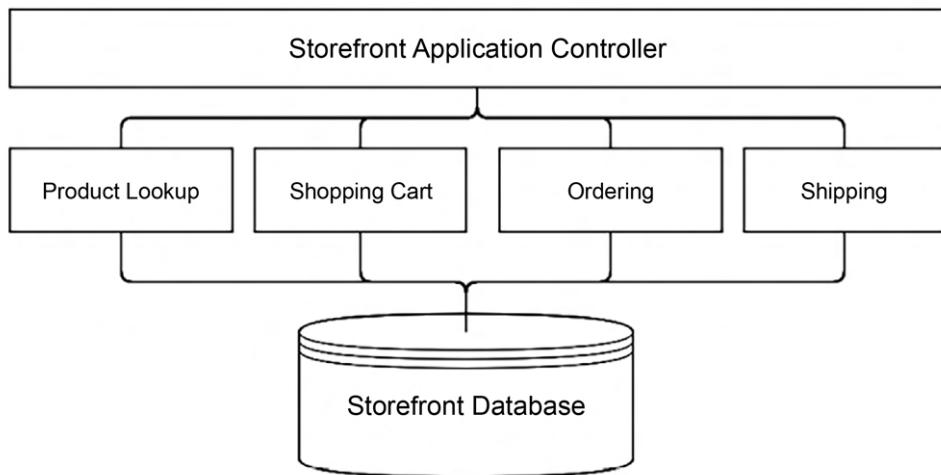


Figure 17.2: Simple microservices architecture

This doesn't look that much more complex. Instead of a big box, there's a bunch of lines. Let's zoom in on the call from our frontend to our product lookup service:

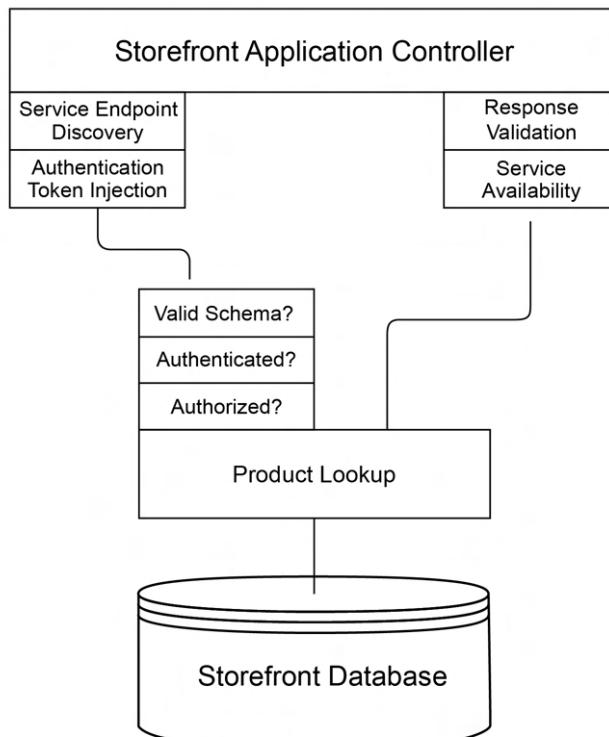


Figure 17.3: Service call architecture

It's no longer a simple function or method call. Now, our storefront controller needs to determine where to send the service call, and this will likely change in each environment. It also needs to inject some kind of authentication token, since you wouldn't want just anyone calling your services. Since the remote service no longer has a local code representation, you'll either need to build the call manually or use a schema language to describe your product listing service, combining it with a client binding. Once the call is made, the service needs to validate the call's schema and apply security rules for authentication and authorization. Once the response is packaged and sent back to our storefront controller, the controller needs to validate the schema of the response. If there's a failure, it needs to decide if it's going to retry or not.

Combine all this additional complexity with version management. Which version of the product look-up service should our storefront use? Are other services tightly coupled together? There are several benefits to the microservices approach, as we discussed earlier, in terms of version and deployment management. These advantages come with the cost of additional complexity.

## **Choosing between monoliths and microservices**

Which of these two approaches is right for you? That really depends. What does your team look like? What are your management needs? Do you need the flexibility that comes from microservices or will a monolith's simpler design make for an easier-to-manage system?

One of the major benefits of a microservice architecture is that you can have multiple teams working on their own code without having to share the same source repository. Before assuming that breaking the services into their own source repositories will benefit your team, how closely tied are the services? If there are numerous interdependencies, then your microservices are really just a distributed monolith, and you may not get the benefits of different repositories. It may be easier to manage branches and merge them.

Also, will your services need to be called by other systems? Look at the cluster we built in the last chapter. Kiali has its own services, but they're not likely to be used by other applications. Jaeger and Prometheus, however, do have services that are used by Kiali, even if those systems have their own frontends too. In addition to these services, Kiali uses the Kubernetes API. All these components are deployed separately and are managed separately. They need to be upgraded on their own, monitored, and so on. This can be a management headache because each system is independently managed and maintained. That said, it wouldn't make any sense for the Kiali team to re-implement Prometheus and Jaeger in their own project. It also wouldn't make sense to just import the entire source tree for these projects and be forced to keep them up to date.

## **Using Istio to help manage microservices**

We've spent quite a bit of time talking about microservices and monoliths without talking about Istio. Earlier in this chapter, *Figure 17.3* pointed out decisions that were needed by our microservice before we could get to calling our code.

These should look familiar because we covered objects from Istio that service most of these needs in the last chapter! Istio can remove our need to write code to authenticate and authorize clients, discover where services are running, and manage traffic routing. Throughout the rest of this chapter, we're going to walk through building a small application off of a microservice, using Istio to leverage these common services without having to build them into our code.

So far, we've looked at the differences between monoliths and microservices, and how those differences interact with Istio at a conceptual level. Next, we'll see how a monolith is deployed into Istio.

## Deploying a monolith

This chapter is about microservices, so why are we starting with deploying monoliths in Istio? The first answer is, because we can! There's no reason to not get the benefits of Istio's built-in capabilities when working with monoliths in your cluster. Even though it's not a "microservice," it's still good to be able to trace through application requests, manage deployments, and so on. The second answer is, because we need to. Our microservice will need to know which user in our enterprise is calling it. To do that, Istio will need a JWT to validate. We'll use OpenUnison to generate JWTs first so that we can call our service manually, and then so we can authenticate users from a frontend and allow it to call our service securely.

Starting with your cluster from *Chapter 16*, we're now going to deploy OpenUnison. Go to the `chapter17/openunison-istio` directory and run `deploy_openunison_istio.sh`:

```
cd chapter17/openunison-istio  
./deploy_openunison_istio.sh
```

This is going to take a while to run. This script does a few things:

1. Deploys cert-manager with our enterprise CA.
2. Deploys all of the OpenUnison components (including our testing Active Directory) for impersonation, so we don't need to worry about updating the API server for SSO to work.
3. Labels the `openunison` namespace with `istio-injection: enabled`. This tells Istio to enable sidecar injection for all pods. You can do this manually by running `kubectl label ns openunison istio-injection=enabled`.
4. Creates all of our Istio objects for us (we'll go into the details of these next).
5. Creates an `ou-tls-certificate` Certificate in the the `istio-system` namespace. Again, we'll dive into the details as to why in the next section.

Once the script is run, we're able to now log in to our monolith! Just like in *Chapter 6, Integrating Authentication into Your Cluster*, go to <https://k8sou.apps.XX-XX-XX-XX.nip.io/> to log in, where `XX-XX-XX-XX` is your host's IP address.

For instance, my host runs on `192.168.2.114`, so my URL is <https://k8sou.apps.192-168-2-114.nip.io/>. Again, as in *Chapter 6*, the username is `mmosley` and the password is `start123`.

Now that our monolith is deployed, let's walk through the Istio-specific configuration as it relates to our deployment.

## Exposing our monolith outside our cluster

Our OpenUnison is running, so let's look at the objects that expose it to our network. There are two main objects that do this work: `Gateway` and `VirtualService`. These objects were created when we installed OpenUnison. How these objects are configured was described in *Chapter 16, An Introduction to Istio*. Then, we'll look at running instances to show how they grant access. First, let's look at the important parts of our gateways. There are two. The first one, `openunison-gateway-orchestra`, handles access to the OpenUnison portal and the Kubernetes Dashboard:

```
apiVersion: networking.istio.io/v1beta1
kind: Gateway
metadata:
spec:
  selector:
    istio: ingressgateway
  servers:
    - hosts:
        - k8sou.apps.192-168-2-114.nip.io
        - k8sdb.apps.192-168-2-114.nip.io
      port:
        name: http
        number: 80
        protocol: HTTP
      tls:
        httpsRedirect: true
    - hosts:
        - k8sou.apps.192-168-2-114.nip.io
        - k8sdb.apps.192-168-2-114.nip.io
      port:
        name: https-443
        number: 443
        protocol: HTTPS
      tls:
        credentialName: ou-tls-certificate
        mode: SIMPLE
```

The `selector` tells Istio which `ingress-ingressgateway` pod to work with. The default gateway deployed to `istio-system` has the label `istio: ingressgateway`, which will match this one. You could run multiple gateways, using this section to determine which one you want to expose your service to. This is useful if you have multiple networks with different traffic or if you want to separate traffic between applications on a cluster.

The first entry in the servers list tells Istio that if a request comes on HTTP to port 80 for either of our hosts, then we want Istio to send a redirect to the HTTPS port. This is a good security practice, so folks, don't try to bypass HTTPS. The second entry in servers tells Istio to accept HTTPS connections on port 443, using the certificate in the Secret named ou-tls-certificate. This Secret must be a TLS Secret and be in the same namespace as the pod running the ingress gateway. For our cluster, this means that ou-tls-certificate *MUST* be in the istio-system namespace. That's why our deployment script created the wild card certificate in the istio-system namespace. This is different from using an Ingress object with NGINX, where you keep the TLS Secret in the same namespace as your Ingress object.

If you don't include your Secret in the correct namespace, it can be difficult to debug. The first thing you'll notice is that when you try to connect to your host, your browser will report that the connection has been reset. This is because Istio doesn't have a certificate to serve. Kiali won't tell you there's a configuration issue, but looking at the istiod pod in istio-system's logs, you'll find failed to fetch key and certificate for kubernetes://secret-name, where secret-name is the name of your Secret. Once you copy your Secret into the correct namespace, your app will start working on HTTPS.

The second Gateway, openunison-api-gateway-orchestra, is used to expose OpenUnison directly via HTTPS for the API server host. This bypasses most of Istio's built-in functionality, so it's not something we'll want to do unless needed. The important difference in this Gateway versus our other Gateway is how we configure TLS:

```
- hosts:  
  - k8sapi.192-168-2-114.nip.io  
port:  
  name: https-443  
  number: 443  
  protocol: HTTPS  
tls:  
  mode: PASSTHROUGH
```

We use PASSTHROUGH as the mode instead of SIMPLE. This tells Istio to not bother trying to decrypt the HTTPS request and, instead, send it downstream. We have to do this for the Kubernetes API calls because Envoy doesn't support the SPDY protocol used by kubectl for exec, cp, and port-forward, so we need to bypass it. This, of course, means that we lose much of Istio's capabilities, so it's not something we want to do if we can avoid it.

While the Gateway objects tell Istio how to listen for connections, the VirtualService objects tell Istio where to send the traffic to. Just like with the Gateway objects, there are two VirtualService objects. The first object handles traffic for both the OpenUnison portal and the Kubernetes Dashboard. Here are the important parts:

```
spec:  
  gateways:  
    - openunison-gateway-orchestra
```

```
hosts:
  - k8sou.apps.192-168-2-114.nip.io
  - k8sdb.apps.192-168-2-114.nip.io
http:
  - match:
    - uri:
      prefix: /
route:
  - destination:
    host: openunison-orchestra
    port:
      number: 80
```

The `gateways` section tells Istio which Gateway objects to link this to. You could, in theory, have multiple Gateways as sources for traffic. The `hosts` section tells Istio which hostnames to apply this configuration to, with the `match` section telling Istio what conditions to match requests on. This section can provide quite a bit of power for routing microservices, but for monoliths, just `/` is usually good enough.

Finally, the `route` section tells Istio where to send the traffic. `destination.host` is the name of the Service you want to send the traffic to. We're sending all traffic to port 80 (sort of).

The NGINX Ingress version of this configuration sent all traffic to OpenUnison's HTTPS port (8443). This meant that all data was encrypted over the wire from the user's browser, all the way to the OpenUnison pod. We're not doing that here because we're going to rely on mTLS from Istio's sidecar.

Even though we're sending traffic to port 80 using HTTP, the traffic will be encrypted from when it leaves the `ingressgateway` pod until it arrives at the sidecar on our OpenUnison pod that intercepts all of OpenUnison's inbound network connections. There's no need to configure TLS explicitly!

Now that we're routing traffic from our network to OpenUnison, let's tackle a common requirement of monolithic applications: sticky sessions.

## Configuring sticky sessions

Most monolithic applications require sticky sessions. Enabling sticky sessions means that every request in a session is sent to the same pod. This is generally not needed in microservices because each API call is distinct. Web applications that users interact with generally need to manage state, usually via cookies. However, those cookies don't generally store all of the session's state because they would get too big and would likely have sensitive information. Instead, most web applications use a cookie that points to a session that's saved on the server, usually in memory. While there are ways to make sure that this session is available to any instance of the application in a highly available way, it's not very common to do so. These systems are expensive to maintain and are generally not worth the work.

OpenUnison is no different than most other web applications and needs to make sure that sessions are sticky to the pod they originated from. To tell Istio how we want sessions to be managed, we use `DestinationRule`. The `DestinationRule` objects tell Istio what to do about traffic routed to a host by a `VirtualService`. Here are the important parts of ours:

```
spec:  
  host: openunison-orchestra  
  trafficPolicy:  
    loadBalancer:  
      consistentHash:  
        httpCookie:  
          name: openunison-orchestra  
          path: /  
          ttl: 0s  
    tls:  
      mode: ISTIO_MUTUAL
```

The `host` in the rule refers to the target (`Service`) of the traffic, not the hostname in the original URL. `spec.trafficPolicy.loadBalancer.consistentHash` tells Istio how we want to manage stickiness. Most monolithic applications will want to use cookies. `ttl` is set to `0s`, so the cookie is considered a “session cookie.” This means that when the browser is closed, the cookie disappears from its cookie jar.

You should avoid cookies with specific times to live. These cookies are persisted by the browser and can be treated as a security risk by your enterprise.

With OpenUnison up and running and understanding how Istio is integrated, let’s take a look at what Kiali will tell us about our monolith.

## Integrating Kiali and OpenUnison

First, let’s integrate OpenUnison and Kiali. Kiali, like any other cluster management system, should be configured to require access. Kiali, just like the Kubernetes Dashboard, can integrate with Impersonation so that Kiali will interact with the API server, using the user’s own permissions. Doing this is pretty straight forward. We created a script in the `chapter17/kiali` folder called `integrate-kiali-openunison.sh` that:

1. Deletes the old Gateways and VirtualServices for Kiali, Prometheus, Jaeger, and Grafana.
2. Updates Grafana to accept a header for SSO from OpenUnison.
3. Updates the Kiali Helm chart to use header for `auth.strategy` and restarts Kiali to pick up the changes.
4. Redeploys OpenUnison with Kiali, Prometheus, Jaeger, and Grafana integrated for SSO.

The integration works the same way as the dashboard, but if you’re interested in the details, you can read about them at <https://openunison.github.io/applications/kiali/>.

With the integration completed, let's see what Kiali can tell us about our monolith. First, log in to OpenUnison. You'll see new badges on the portal screen:

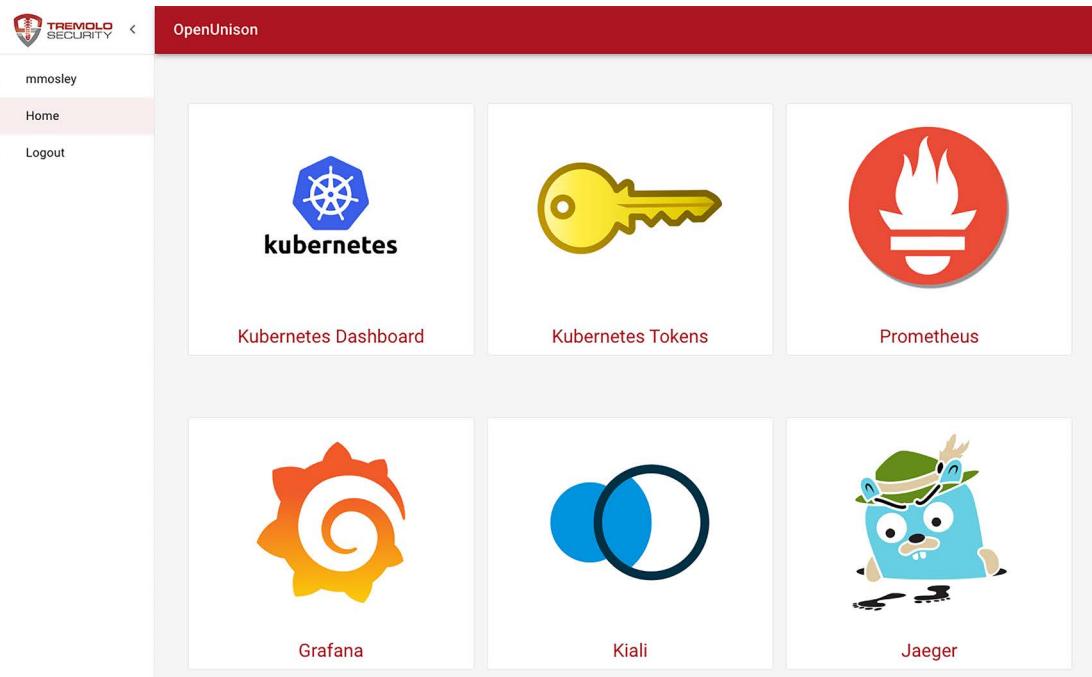


Figure 17.4: OpenUnison portal with the Kiali, Prometheus, Grafana, and Jaeger badges

Next, click on the Kiali badge to open Kiali, then click on **Graphs**, and choose the `openunison` namespace. You'll see a graph similar to the following:

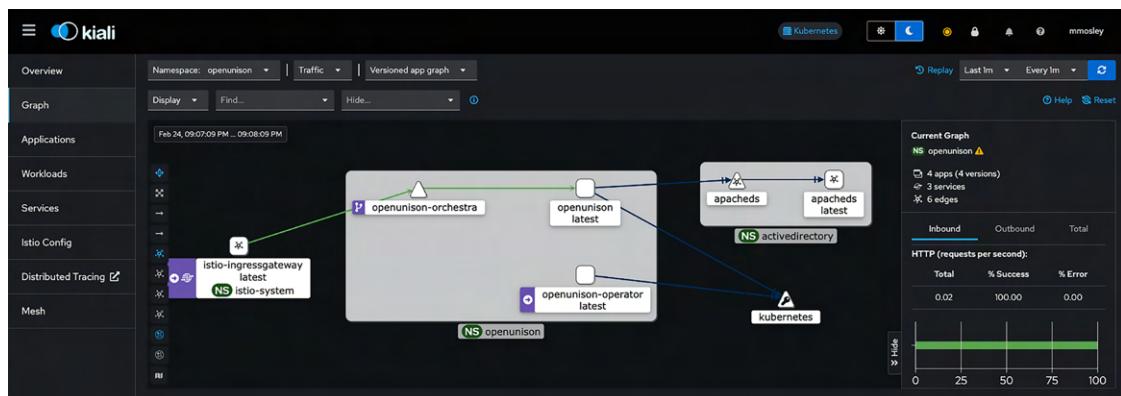


Figure 17.5: OpenUnison graph in Kiali

You can now view the connections between OpenUnison, apacheds, and other containers the same way you would with a microservice! Speaking of which, now that we've learned how to integrate a monolith into Istio, let's build a microservice and learn how it integrates with Istio.

## Building a microservice

We spent quite a bit of time talking about monoliths. First, we discussed which is the best approach for you, then we spent some time showing how to deploy a monolith into Istio to get from it many of the benefits that microservices do. Now, let's dive into building and deploying a microservice. Our microservice will be pretty simple. The goal is to show how a microservice is built and integrated into an application, rather than how to build a full-fledged application based on microservices. Our book is focused on enterprise, so we're going to focus on a service that:

- Requires authentication from a specific user
- Requires authorization for a specific user based on a group membership or attribute
- Does something very *important*
- Generates some log data about what happened

This is common in enterprise applications and the services they're built on. Most enterprises need to be able to associate actions, or decisions, with a particular person in an organization. If an order is placed, who placed it? If a case is closed, who closed it? If a check is cut, who cut it? There are of course many instances where a user isn't responsible for an action. Sometimes, it's another service that is automated. A batch service that pulls in data to create a warehouse isn't associated with a particular person. That is an *interactive* service, meaning that an end user is expected to interact with it, so we're going to assume that the user is a person in the enterprise.

Once you know who is going to use the service, you'll then need to know if the user is authorized to do so. In the previous paragraph, we identified that you need to know "who cut the check." Another important question is, "Are they allowed to cut the check?" You really don't want just anybody in your organization sending out checks, do you? Identifying who is authorized to perform an action could be the subject of multiple books, so to keep things simple, we'll make our authorization decisions based on group membership, at least at a high level.

Having identified the user and authorized them, the next step is to do something *important*. It's an enterprise, filled with important things that need doing! Since writing a check is something that we can all relate to and represents many of the challenges enterprise services face, we're going to stick with this as our example. We're going to write a check service that will let us send out checks.

Finally, having done something *important*, we need to make a record of it. We need to track who called our service, and once the service does the important parts, we need to make sure we record it somewhere. This can be recorded in a database or another service, or even sent to standard-out so that it can be collected by a log aggregator, like the OpenSearch we deployed in *Chapter 15*.

Having identified all the things that our service will do, the next step is to identify which part of our infrastructure will be responsible for each decision and action. For our service:

Action	Component	Description
User Authentication	OpenUnison	Our OpenUnison instance will authenticate users to our “Active Directory”
Service Routing	Istio	How we will expose our service to the world
Service Authentication	Istio	The RequestAuthentication object will describe how to validate the user for our service
Service Coarse Grained Authorization	Istio	AuthorizationPolicy will make sure users are members of a specific group to call our service
Fine-Grained Authorization, or Entitlements	Service	Our service will determine which payees you’re able to write checks for
Writing a Check	Service	The point of writing this service!
Log who Wrote the Check and to whom it was Sent	Service	Write this data to standard-out
Log Aggregation	Kubernetes	In production – a tool like OpenSearch

Table 17.1: Service responsibilities

We’ll build each of these components, layer by layer, in the following sections. Before we get into the service itself, we need to say hello to the world.

## Deploying Hello World

Our first service will be a simple Hello World service that will serve as the starting point for our check-writing service. Our service is built on Python using Flask. We’re using this because it’s pretty simple to use and deploy. Go to chapter17/hello-world and run the `deploy_helloworld.sh` script. This will create our Namespace, Deployment, Service, and Istio objects. Look at the code in the `service-source ConfigMap`. This is the main body of our code and the framework on which we will build our check service. The code itself doesn’t do much:

```
@app.route('/')
def hello():
    retVal = {
        "msg": "hello world!",
        "host": "%s" % socket.gethostname()
    }
    return json.dumps(retVal)
```

This code accepts all requests to / and runs our function called `hello()`, which sends a simple response. We’re embedding our code as a `ConfigMap` for the sake of simplicity.

If you've read all the previous chapters, you'll notice that we're violating some cardinal rules with this container from a security standpoint. It's a Docker Hub container running as root. That's OK for now. We didn't want to get bogged down in the build processes for this chapter. In *Chapter 19, Building a Developer Portal*, we'll walk through using GitLab workflows to build out a more secure version of the container for this service.

Once our service is deployed, we can test it out by using curl:

```
$ curl http://service.192-168-2-114.nip.io/
{"msg": "hello world!", "host": "run-service-785775bf98-fln49"}%
```

This code isn't terribly exciting, but next, we'll add some security to our service.

## Integrating authentication into our service

In *Chapter 16, An Introduction to Istio*, we introduced the RequestAuthentication object. Now, we will use this object to enforce authentication. We want to make sure that in order to access our service, you must have a valid JWT. In the previous example, we just called our service directly. Now, we want to only get a response if a valid JWT is embedded in the request. We need to make sure to pair our RequestAuthentication with an AuthorizationPolicy that forces Istio to require a JWT; otherwise, Istio will only reject JWTs that don't conform to our RequestAuthentication but allow requests that have no JWT at all.

Even before we configure our objects, we need to get a JWT from somewhere. We're going to use OpenUnison. To work with our API, let's deploy the pipeline token generation chart we deployed in *Chapter 6, Integrating Authentication into Your Cluster*. Go to the chapter6/pipelines directory and run the Helm chart:

```
$ helm install orchestra-token-api token-login -n openunison -f /tmp/openuni-
son-values.yaml
NAME: orchestra-token-api
LAST DEPLOYED: Tue Aug 31 19:41:30 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

This will give us a way to easily generate a JWT from our internal Active Directory. Next, we'll deploy the actual policy objects. Go into the chapter17/authentication directory and run `deploy-auth.sh`. It will look like this:

```
$ ./deploy-auth.sh
secret/cacerts configured
pod "istiod-75d8d56b68-tm6b7" deleted
requestauthentication.security.istio.io/hello-world-auth created
authorizationpolicy.security.istio.io/simple-hellow-world created
```

First, we created a Secret called cacerts to store our enterprise CA certificate and restart `istiod`. This will allow `istiod` to communicate with OpenUnison to pull jwks signature verification keys. Next, two objects are created. The first is the `RequestAuthentication` object and then a simple `AuthorizationPolicy`. First, we will walk through `RequestAuthentication`:

```
apiVersion: security.istio.io/v1
kind: RequestAuthentication
metadata:
  name: hello-world-auth
  namespace: istio-hello-world
spec:
  jwtRules:
    - audiences:
        - kubernetes
      issuer: https://k8sou.192-168-2-119.nip.io/auth/idp/k8sIdp
      jwksUri: https://k8sou.192-168-2-119.nip.io/auth/idp/k8sIdp/certs
      outputPayloadToHeader: User-Info
    selector:
      matchLabels:
        app: run-service
```

This object first specifies how the JWT needs to be formatted in order to be accepted. We're cheating here a bit by just leveraging our Kubernetes JWT. Let's compare this object to our JWT:

```
{
  "iss": "https://k8sou.192-168-2-119.nip.io/auth/idp/k8sIdp",
  "aud": "kubernetes",
  "exp": 1630421193,
  "jti": "JGnX1j0I5obI3Vcmb1MCXA",
  "iat": 1630421133,
  "nbf": 1630421013,
  "sub": "mmosley",
  "name": " Mosley",
  "groups": [
    "cn=group2,ou=Groups,DC=domain,DC=com",
    "cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"
  ],
  "preferred_username": "mmosley",
  "email": "mmosley@tremolo.dev"
}
```

The `aud` claim in our JWT lines up with the audiences in our `RequestAuthentication`. The `iss` claim lines up with `issuer` in our `RequestAuthentication`. If either of these claims doesn't match, then Istio will return a 401 HTTP error code to tell you that the request is unauthorized.

We also specify `outputPayloadToHeader: User-Info` to tell Istio to pass the user info to the downstream service as a base64-encoded JSON header, with the name `User-Info`. This header can be used by our service to identify who called it. We'll get into the details of this when we get into entitlement authorization.

Additionally, the `jwksUri` section specifies the URL that contains the RSA public keys used to verify the JWT. This can be obtained by first going to the issuer's OIDC discovery URL and getting the URL from the `jwks` claim.

It's important to note that the `RequestAuthentication` object will tell Istio what form the JWT needs to take, but not what data about the user needs to be present. We'll cover that next, in the authorization section.

Speaking of authorization, we want to make sure to enforce the requirement for a JWT, so we will create this very simple `AuthorizationPolicy`:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: simple-hello-world
  namespace: istio-hello-world
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            requestPrincipals:
              - '*'
  selector:
    matchLabels:
      app: run-service
```

The `from` section says that there must be a `requestPrincipal`. This tells Istio there must be a user (and in this case, anonymous is not a user). `requestPrincipal` comes from JWTs and represents users. There is also a `principal` configuration, but this represents the service calling our URL, which in this case would be `ingressgateway`. This tells Istio that a user must be authenticated via a JWT.

With our policy in place, we can now test it. First, with no user:

```
curl -v http://service.192-168-2-119.nip.io/
*   Trying 192.168.2.119:80...
* TCP_NODELAY set
* Connected to service.192-168-2-119.nip.io (192.168.2.119) port 80 (#0)
> GET / HTTP/1.1
> Host: service.192-168-2-119.nip.io
> User-Agent: curl/7.68.0
> Accept: */*
```

```
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 403 Forbidden
< content-length: 19
< content-type: text/plain
< date: Tue, 31 Aug 2021 20:23:14 GMT
< server: istio-envoy
< x-envoy-upstream-service-time: 2
<
* Connection #0 to host service.192-168-2-119.nip.io left intact
```

We can see that the request was denied with a 403 HTTP code. We received 403 because Istio was expecting a JWT but there wasn't one. Next, let's generate a valid token the same way we did in *Chapter 6, Integrating Authentication into Your Cluster*:

```
curl -H "Authorization: Bearer $(curl --insecure -u 'mmosley:start123' https://k8sou.apps.192-168-2-119.nip.io/k8s-api-token/token/user 2>/dev/null| jq -r '.token.id_token')" http://service.192-168-2-119.nip.io/
>{"msg": "hello world!", "host": "run-service-785775bf98-6bbwt"}
```

Now, we have success! Our hello world service now requires proper authentication. Next, we'll update our authorization to require a specific group from Active Directory.

## Authorizing access to our service

So far, we've built a service and made sure users have a valid JWT from our identity provider before they can access it.

Now, we want to apply what's often referred to as "coarse-grained" authorization. This is application- or service-level access. It says, "You are generally able to use this service," but it doesn't say you're able to perform the action you wish to take. For our check-writing service, you may be authorized to write a check, but there are likely more controls that limit who you can write a check for. If you're responsible for the **Enterprise Resource Planning (ERP)** system in your enterprise, you probably shouldn't be able to write checks for the facility vendors. We'll get into how your service can manage these business-level decisions in the next section, but for now, we'll focus on the service-level authorization.

It turns out we have everything we need. Earlier, we looked at our `mmosley` user's JWT, which had multiple claims. One such claim was the `groups` claim. We used this claim in *Chapter 6, Integrating Authentication into Your Cluster*, and *Chapter 7, RBAC Policies and Auditing*, to manage access to our cluster. In a similar fashion, we'll manage who can access our service based on our membership of a particular group. First, we'll delete our existing policy:

```
kubectl delete authorizationpolicy simple-hellow-world -n istio-hello-world
authorizationpolicy.security.istio.io "simple-hellow-world" deleted
```

With the policy disabled, you can now access your service without a JWT. Next, we'll create a policy that requires you to be a member of the group `cn=group2,ou=Groups,DC=domain,DC=com` in our Active Directory.

Deploy the below policy (in `chapter17/coursed-grained-authorization/coursed-grained-az.yaml`):

```
---
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: service-level-az
  namespace: istio-hello-world
spec:
  action: ALLOW
  selector:
    matchLabels:
      app: run-service
  rules:
  - when:
    - key: request.auth.claims[groups]
      values: ["cn=group2,ou=Groups,DC=domain,DC=com"]
```

This policy tells Istio that only users with a claim called `groups` that have the value `cn=group2,ou=Groups,DC=domain,DC=com` are able to access this service. With this policy deployed, you'll see that you can still access the service as `mosley`, and trying to access the service anonymously still fails. Next, try accessing the service as `jackson`, with the same password:

```
curl -H "Authorization: Bearer $(curl --insecure -u 'jackson:start123' https://k8sou.192-168-2-119.nip.io/k8s-api-token/token/user 2>/dev/null| jq -r '.token.id_token')" http://service.192-168-2-119.nip.io/
RBAC: access denied
```

We're not able to access this service as `jackson`. If we look at `jackson's id_token`, we can see why:

```
{
  "iss": "https://k8sou.192-168-2-119.nip.io/auth/idp/k8sIdp",
  "aud": "kubernetes",
  "exp": 1630455027,
  "jti": "Ae4Nv22HHYChUNJx78010A",
  "iat": 1630454967,
  "nbf": 1630454847,
  "sub": "jackson",
  "name": " Jackson",
  "groups": "cn=k8s-create-ns,ou=Groups,DC=domain,DC=com",
```

```
"preferred_username": "jjackson",
"email": "jjackson@tremolo.dev"
}
```

Looking at the claims, jjackson isn't a member of the group `cn=group2,ou=Groups,DC=domain,DC=com`.

Now that we're able to tell Istio how to limit access to our service to valid users, the next step is to tell our service who the user is. We'll then use this information to look up authorization data, log actions, and act on the user's behalf.

## Telling your service who's using it

When writing a service that does anything involving a user, the first thing you need to determine is, “Who is trying to use my service?” So far, we have told Istio how to determine who the user is, but how do we propagate that information down to our service? Our `RequestAuthentication` included the configuration option `outputPayloadToHeader: User-Info`, which injects the claims from our user’s authentication token as base64-encoded JSON into the HTTP request’s headers. This information can be pulled from that header and used by your service to look up additional authorization data.

We can view this header with a service we built, called `/headers`. This service will just give us back all the headers that are passed to our service. Let’s take a look:

```
curl -H "Authorization: Bearer $(curl --insecure -u 'mmosley:start123'
https://k8sou.192-168-2-119.nip.io/k8s-api-token/token/user 2>/dev/null| jq -r
'.token.id_token')"
http://service.192-168-2-119.nip.io/headers 2>/dev/null |
jq -r '.headers'

Host: service.192-168-2-119.nip.io
User-Agent: curl/7.75.0
Accept: */*
X-Forwarded-For: 192.168.2.112
X-Forwarded-Proto: http
X-Request-Id: 6397d068-537e-94b7-bf6b-a7c649db5b3d
X-Envoy-Attempt-Count: 1
X-Envoy-Internal: true
X-Forwarded-Client-Cert: By=spiffe://cluster.local/ns/istio-hello-world/sa/
default;Hash=1a58a7d0abf62d32811c084a84f0a0f42b28616ffde7b6b840c595149d99b2eb;-
Subject="";URI=spiffe://cluster.local/ns/istio-system/sa/istio-ingressgate-
way-service-account
User-Info: eyJpc3MiOiJodHRwczovL2s4c291LjE5Mi0xNjgtMi0xMTkubmlwLmlvL2F1dGgvaWR-
wL2
s4c01kcCIsImF1ZCI6Imt1YmVybmv0ZXMiLCJleHAiOjE2MzA1MTY4MjQsImp0aSI6InY0e
kpCNzdfRktpOXJ0QU5jWDVwS1EiLCJpYXQiOjE2MzA1MTY3NjQsIm5iZiI6MTYzMDUxNj
Y0NCwic3ViIjoibW1vc2xleSIsIm5hbWUiOiIgTW9zbGV5IiwiZ3JvdXBzIjpbiMNuPWdy
b3VwMixvdT1Hcm91cHMvREM9ZG9tYWluLERDPWNvbSISImNuPWs4cy1jbHVzdGVyLWFkbW
lucyxdvdt1Hcm91cHMvREM9ZG9tYWluLERDPWNvbSJdLCJwcmVmZXJyZWRfdXNlc5hbWUi
```

```
OiJtbW9zbGV5IiwiZW1haWwiOiJtbW9zbGV5QHRyZW1vbG8uZGV2In0=
X-B3-Traceid: 28fb185aa113ad089cfac2d6884ce9ac
X-B3-Spanid: d40f1784a6685886
X-B3-Parentspanid: 9cfac2d6884ce9ac
X-B3-Sampled: 1
```

There are several headers here. The one we care about is `User-Info`. This is the name of the header we specified in our `RequestAuthentication` object. If we decode from base64, we'll get some JSON:

```
{
  "iss": "https://k8sou.192-168-2-119.nip.io/auth/idp/k8sIdp",
  "aud": "kubernetes",
  "exp": 1630508679,
  "jti": "5VoEAAgv1rkpf1vOJ9uo-g",
  "iat": 1630508619,
  "nbf": 1630508499,
  "sub": "mmosley",
  "name": " Mosley",
  "groups": [
    "cn=group2,ou=Groups,DC=domain,DC=com",
    "cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"
  ],
  "preferred_username": "mmosley",
  "email": "mmosley@tremolo.dev"
}
```

We have all the same claims as if we had decoded the token ourselves. What we don't have is the JWT. This is important from a security standpoint. Our service can't leak a token it doesn't possess.

Now that we know how to determine who the user is, let's integrate that into a simple `who-am-i` service that just tells us who the user is. First, let's look at our code:

```
@app.route('/who-am-i')
def who_am_i():
    user_info = request.headers["User-Info"]
    user_info_json = base64.b64decode(user_info).decode("utf8")
    user_info_obj = json.loads(user_info_json)
    ret_val = {
        "name": user_info_obj["sub"],
        "groups": user_info_obj["groups"]
    }
    return json.dumps(ret_val)
```

This is pretty basic. We're getting the header from our request. Next, we decode it from base64, and finally, we get the JSON and add it to a return. If this were a more complex service, this is where we might query a database to determine what entitlements our user has.

In addition to not requiring that our code knows how to verify the JWT, this also makes it easier for us to develop our code in isolation from Istio. Open a shell in your `run-service` pod and try accessing this service directly with any user:

```
kubectl exec -ti run-service-785775bf98-g86gl -n istio-hello-world - bash
# export USERINFO=$(echo -n '{"sub":"marc","groups":["group1","group2"]}' |
base64 -w 0)
# curl -H "User-Info: $USERINFO" http://localhost:8080/who-am-i
{"name": "marc", "groups": ["group1", "group2"]}
```

We were able to call our service without having to know anything about Istio, JWTs, or cryptography! Everything was offloaded to Istio so that we could focus on our service. While this does make for easier development, what are the impacts on security if there's a way to inject any information we want into our service?

Let's try this directly from a namespace that doesn't have the Istio sidecar:

```
$ kubectl run -i --tty curl --image=alpine --rm=true - sh
/ # apk update add curl
/ # curl -H "User-Info $(echo -n '{"sub":"marc","groups":["group1","group2"]}' |
base64 -w 0)" http://run-service.istio-hello-world.svc/who-am-i
RBAC: access denied
```

Our `RequestAuthentication` and `AuthorizationPolicy` stop the request. While we're not running the sidecar, our service is, and it redirects all traffic to Istio where our policies will be enforced. What about if we try to inject our own `User-Info` header from a valid request?

```
export USERINFO=$(echo -n '{"sub":"marc","groups":["group1","group2"]}' |
base64 -w 0)
curl -H "Authorization: Bearer $(curl --insecure -u 'mmosley:start123' https://k8sou.192-168-2-119.nip.io/k8s-api-token/token/user 2>/dev/null| jq -r '.token.id_token')" -H "User-Info: $USERINFO" http://service.192-168-2-119.nip.io/who-am-i
>{"name": "mmosley", "groups": ["cn=group2,ou=Groups,DC=domain,DC=com",
"cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"]}
```

Once again, our attempt to override who the user is outside of a valid JWT has been foiled by Istio. We've shown how Istio injects a user's identity into our service; now, we need to know how to authorize a user's entitlements.

## Authorizing user entitlements

So far, we've managed to add quite a bit of functionality to our service without having to write any code. We added token-based authentication and coarse-grained authorization. We know who the user is and have determined that, at the service level, they are authorized to call our service. Next, we need to decide if the user is allowed to do the specific action they're trying to do. This is often called fine-grained authorization or entitlements. In this section, we'll walk through multiple approaches you can take, discussing how you should choose an approach.

### Authorizing in service

Unlike coarse-grained authorizations and authentication, entitlements are generally not managed at the service mesh layer. That's not to say it's impossible. We'll talk about ways you can do this in the service mesh, but in general, it's not the best approach. Authorizations are generally tied to business data that's usually locked up in a database. Sometimes, that database is a generic relational database, like MySQL or SQL Server, but it could really be anything. Since the data used to make the authorization decision is often owned by the service owner, not the cluster owner, it's generally easier and more secure to make entitlement decisions directly in our code.

Earlier, we discussed in our check-writing service that we don't want someone responsible for the ERP to cut checks to the facilities vendor. Where is the data that determines that? Well, it's probably in your enterprise's ERP system. Depending on how big you are, this could be a homegrown application or a SAP or Oracle. Let's say you wanted Istio to make the authorization decision for our check-writing service. How would it get that data? Do you think the people responsible for the ERP want you, as a cluster owner, to talk to their database directly? Do you, as a cluster owner, want that responsibility? What happens when something goes wrong with the ERP and someone points the finger at you for the problem? Do you have the resources to prove that you, and your team, were not responsible?

It turns out that the silos in enterprises that benefit from the management aspects of microservice design also work against centralized authorization. In our example of determining who can write the check for a specific vendor, it's probably just easiest to make this decision inside our service. This way, if there's a problem, it's not the Kubernetes team's responsibility to determine the issue, and the people who are responsible are in control of their own destiny.

That's not to say there isn't an advantage to a more centralized approach to authorization. Having teams implement their own authorization code will lead to different standards being used and different approaches. Without careful controls, it can lead to a compliance nightmare. Let's look at how Istio could provide a more robust framework for authorization.

### Using OPA with Istio

Using the Envoy filters feature discussed in *Chapter 16, An Introduction to Istio*, you can integrate the Open Policy Agent (OPA) into your service mesh to make authorization decisions. We discussed OPA in *Chapter 11, Extending Security Using Open Policy Agent*. There are a few key points about OPA we need to review:

- OPA does not (typically) reach out to external data stores to make authorization decisions. Much of the benefit of OPA requires that it uses its own internal database.

- OPA's database is not persistent. When an OPA instance dies, it must be repopulated with data.
- OPA's databases are not clustered. If you have multiple OPA instances, each database must be updated independently.

To use OPA to validate whether our user can write a check for a specific vendor, OPA would either need to be able to pull that data directly from the JWT or have the ERP data replicated in its own database. The former is unlikely to happen for multiple reasons. First, the issues with your cluster talking to your ERP will still exist when your identity provider tries to talk to your ERP. Second, the team that runs your identity provider would need to know to include the correct data, which is a difficult task and is unlikely something they're interested in doing. Finally, there could be numerous folks, from security to the ERP team, who are not comfortable with this data being stored in a token that gets passed around. The latter option, syncing data into OPA, is more likely to be successful.

There are two ways you could sync your authorization data from your ERP into your OPA databases. The first is by pushing the data. A “bot” could push updates to each OPA instance. This way, the ERP owner is responsible for pushing the data, with your cluster just being a consumer. However, there's no simple way to do this, and security would be a concern to make sure someone doesn't push false data. The alternative is to write a pull “bot” that runs as a sidecar to your OPA pods. This is how GateKeeper works. The advantage here is that you have the responsibility of keeping your data synced without having to build a security framework to push data.

In either scenario, you'll need to understand whether there are any compliance issues with the data you are storing. Now that you have the data, what's the impact of losing it in a breach? Is that a responsibility you want?

Centralized authorization services have been discussed for entitlements long before Kubernetes or even RESTful APIs existed. They even predate SOAP and XML! For enterprise applications, it's never really worked because of the additional costs in data management, ownership, and bridging silos. If you own all of the data, this is a great approach. When one of the main goals of microservices is to allow silos to better manage their own development, forcing a centralized entitlements engine is not likely to succeed.

With all that said, there has been a move towards centralizing authorization services. This movement has spawned several commercial companies and projects outside of OPA:

- **Cedar:** An open source project from Amazon Web Services that creates a new policy language. Amazon has also created a service built on this language: <https://github.com/cedar-policy>.
- **Topaz:** Built on OPA and Zanzibar, Topaz provides the OPA authorization engine with relationship-based authorizations from Zanzibar. There's also a commercial offering: <https://github.com/aserto-dev/topaz>.
- **OpenFGA:** Another engine built on Zanzibar's relationship-based authorization system, built by Auth0/Okta: <https://github.com/openfga>.

We're not going to dive into any of these solutions in detail; the point is that there has been a clear movement toward building authorization solutions, similar to how externalized authentication has been a product and project category for multiple decades.

Now that we know some of the issues involved in creating a centralized authorization, let's build out an authorization rule with OPA for Istio.

## Creating an OPA Authorization Rule

Earlier in this section, we discussed writing checks. A common rule for writing checks is that the person who writes the check is now also allowed to sign the check. This rule is called a “separation of duties.” It’s designed to build checkpoints for potentially harmful and costly processes. For instance, if an employee were allowed to both write the check and sign it, there’s no chance for someone to ask if the check is being written for a valid reason.

We’ve already implemented an `AuthorizationPolicy` that validates group membership, but for separation of duties, what we want is to implement a rule that validates that a user is a member of one group while NOT a member of another group. This sort of complex decision isn’t possible with a generic `AuthorizationPolicy`, so we’re going to need to build our own. We can use OPA as our authorization engine while instructing Istio to use our policy.

Before we deploy our policy, let’s review it. The full policy is in `chapter17/opa/rego` and includes test cases:

```
package istio.authz
import input.attributes.request.http as http_request
import input.parsed_path
default allow = false
contains_element(arr, elem) = true {
    arr[_] = elem
} else = false { true }
verify_headers() = payload {
    startswith(http_request.headers["authorization"], "Bearer ")
    [header, payload, signature] := io.jwt.decode(trim_prefix(http_request.
headers["authorization"]), "Bearer "))
} else = false {true}

allow {
    payload := verify_headers()
    payload.groups
    contains_element(payload.groups, "cn=k8s-cluster-admins,ou=Groups,DC=do-
main,DC=com")
    not contains_element(payload.groups, "cn=group2,ou=Groups,DC=domain,DC=com")
}
allow {
    payload := verify_headers()
    payload.groups
    payload.groups == "cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"
}
```

We removed the comments to make the code more compact. The basics of this policy are that we:

1. Verify that there's an authorization header.
2. Verify that the authorization header is a Bearer token.
3. Verify that the bearer token is a JWT.
4. Parse the JWT and verify that there is a groups claim.
5. If the groups claim is a list, make sure that it contains the k8s-cluster-admins group, but NOT the group2 group.
6. If the groups claim is not a list, only validate that it's the k8s-cluster-admins group.

We are validating that the authorization header is present and properly formatted because Istio does not require a token to be present to pass authentication. This was done by our previous `AuthorizationPolicy`, either explicitly by requiring that a principal be present or implicitly by requiring that a specific claim has a specific value.

Once we have validated that the authorization header is properly formatted, we parse it for a payload. We're not validating the JWT based on its public key, validity, or issuer because our `RequestAuthentication` object is doing that for us. We just need to make sure that the token is there.

Finally, we have two potential `allow` policies. The first is if the `groups` claim is a list, so we need to apply array logic to see if the correct group is present and that the forbidden group is not present. The second `allow` policy will trigger if the `groups` claim is not a list but only a single value. In this case, we only care that the group's value is our admin group.

To deploy this policy, go to the `chapter17/opa` directory and run `deploy_opa_istio.sh`. The script will enable authorization and deploy our policy:

1. **Configure istiod:** Updates the `istio ConfigMap` that stores the mesh configuration to enable the `envoyExtAuthzGrpc` extension provider.
2. **Deploy an OPA mutating admission controller:** A mutating admission controller is deployed to automate the creation of an OPA instance on pods that runs alongside your services.
3. **Deploys our policy:** The policy we created earlier is created as a `ConfigMap`.
4. **Redeploy our service:** Deletes the pod so that it is recreated with our authorization policy.
5. Once everything is deployed, we can now verify that our policy is being enforced using a `curl` command. If we try to call our headers service now with our `mmosley` user, it will fail because `mmosley` is a member of both the `k8s-cluster-admin` group and the `group2` group:

```
$ curl -v -H "Authorization: Bearer $(curl --insecure -u 'mmosley:start123' https://k8sou.apps.192-168-2-96.nip.io/k8s-api-token/token/user 2>/dev/null| jq -r '.token.id_token')" http://service.192-168-2-96.nip.io/headers
* Trying 192.168.2.96:80...
* Connected to service.192-168-2-96.nip.io (192.168.2.96) port 80
> GET /headers HTTP/1.1
> Host: service.192-168-2-96.nip.io
> User-Agent: curl/8.4.0
```

```
> Accept: */*
> Authorization: Bearer ...
>
< HTTP/1.1 403 Forbidden
< date: Tue, 27 Feb 2024 19:49:49 GMT
< server: istio-envoy
< content-length: 0
< x-envoy-upstream-service-time: 8
<
* Connection #0 to host service.192-168-2-96.nip.io left intact
```

However, if we use our `pipeline_svc_account` user, it succeeds because this user is only a member of the `k8s-cluster-admin` group:

```
$ curl -H "Authorization: Bearer $(curl --insecure -u 'pipeline_svc_ac-
count:start123' https://k8sou.apps.192-168-2-96.nip.io/k8s-api-token/token/user
2>/dev/null| jq -r '.token.id_token')" http://service.192-168-2-96.nip.io/headers
{"headers": "Host: service.192-168-2-96.nip.io\r\nUser-Agent: curl/8.4.0\r\nAc-
cept: */*\r\nX-Forwarded-For: 192.168.3.6\r\nX-Forwar..."}
```

Now, we can build more complex policies than what's possible with Istio's `AuthorizationPolicy`'s built-in authorization capabilities.

Having determined how to integrate entitlements into our services, the next question we need to answer is, how do we securely call other services?

## Calling other services

We've written services that do simple things, but what about when your service needs to talk to another service? Just like with almost every other set of choices in your cluster rollout, you have multiple options to authenticate to other services. Which choice you make will depend on your needs. We'll first cover the OAuth2 standard way of getting new tokens for service calls and how Istio works with it. We'll then cover some alternatives that should be considered anti-patterns but that you may choose to use anyway.

## Using OAuth2 Token Exchange

Your service knows who your user is but needs to call another service. How do you identify yourself to the second service? The OAuth2 specification, which OpenID Connect is built on, has RFC 8693 – OAuth2 Token Exchange for this purpose. The basic idea is that your service will get a fresh token from your identity provider for the service call, based on the existing user. By getting a fresh token for your own call to a remote service, you're making it easier to lock down where tokens can be used and who can use them, allowing yourself to more easily track a call's authentication and authorization flow. The following diagram gives a high-level overview.

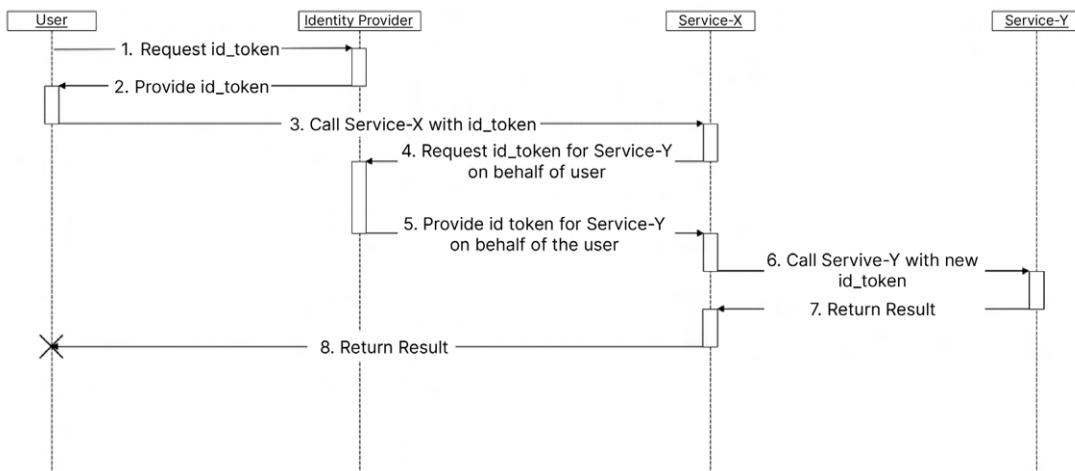


Figure 17.6: OAuth2 Token Exchange sequence

There are some details we'll walk through that depend on your use case:

1. The user requests an `id_token` from the identity provider. How the user gets their token doesn't really matter for this part of the sequence. We'll use a utility in OpenUnison for our lab.
2. Assuming you're authenticated and authorized, your identity provider will give you an `id_token` with an `aud` claim that will be accepted by Service-X.
3. The user uses the `id_token` as a bearer token to call Service-X. It goes without saying that Istio will validate this token.
4. Service-X requests a token for Service-Y from the identity provider on behalf of the user. There are two potential methods to do this. One is impersonation; the other is delegation. We'll cover both in detail later in this section. You'll send your identity provider your original `id_token` and something to identify the service to the identity provider.
5. Assuming Service-X is authorized, the identity provider sends a new `id_token` to Service-X with the original user's attributes and an `aud` scoped to Service-Y.
6. Service-X uses the new `id_token` as the `Authorization` header when calling Service-Y. Again, Istio validates the `id_token`.

Steps 7 and 8 in the previous diagram aren't really important here.

If you think this seems like quite a bit of work to make a service call, you're right. There are several authorization steps going on here:

1. The identity provider authorizes the user to generate a token scoped to Service-X.
2. Istio validates the token and that it's properly scoped to Service-X.
3. The identity provider authorizes Service-X to get a token for Service-Y and to do so for our user.
4. Istio validates that the token used by Service-X for Service-Y is properly scoped.

These authorization points provide a chance for an improper token to be stopped, allowing you to create very short-lived tokens that are harder to abuse and are more narrowly scoped. For instance, if the token used to call Service-X was leaked, it couldn't be used to call Service-Y on its own. You'd still need Service-X's own token before you could get a token for Service-Y. That's an additional step an attacker would need to take in order to get control of Service-Y. It also means breaching more than one service, providing multiple layers of security. This lines up with our discussion of defense in depth from *Chapter 11, Extending Security Using Open Policy Agent*. With a high-level understanding of how OAuth2 Token Exchange works, the next question we need to answer is, how will your services authenticate themselves to your identity provider?

## Authenticating your service

In order for the token exchange to work, your identity provider needs to know who the original user is and which service wants to exchange the token on behalf of the user. In the check-writing service example we've discussed, you wouldn't want the service that provides today's lunch menu to be able to generate a token for issuing a check! You accomplish this by making sure your identity provider knows the difference between your check-writing services and your lunch menu service by authenticating each service individually.

There are three ways a service running in Kubernetes can authenticate itself to the identity provider:

1. Use the Pod's ServiceAccount token
2. Use Istio's mTLS capabilities
3. Use a pre-shared "client secret"

Throughout the rest of this section, we're going to focus on option #1, using the Pod's built-in ServiceAccount token. This token is provided by default for each running pod. This token can be validated by either submitting it to the API server's `TokenReview` service or by treating it as a JWT, validating it against the public key published by the API server.

In our examples, we're going to use the `TokenReview` API to test the passed-in ServiceAccount token against the API server. This is the most backward-compatible approach and supports any kind of token integrated into your cluster. For instance, if you're deployed in a managed cloud with its own IAM system that mounts tokens, you could use that as well. This could generate a considerable amount of load on your API server, since every time a token needs to be validated, it gets sent to the API server.

The `TokenRequest` API discussed in *Chapter 6, Integrating Authentication into Your Cluster*, can be used to cut down on this additional load. Instead of using the `TokenReview` API, we can call the API server's issuer endpoint to get the appropriate token verification public key and use that key to validate the token's JWT. While this is convenient and scales better, it does have some drawbacks:

1. Starting in 1.21, ServiceAccount tokens are mounted using the `TokenRequest` API but with lifespans of a year or more. You can manually change this to be as short as 10 minutes.
2. Validating the JWT directly against a public key won't tell you if the pod is still running. The `TokenReview` API will fail if a ServiceAccount token is associated with a deleted pod, adding an additional layer of security.

3. Enabling this feature requires enabling anonymous authentication in your cluster, which can be leveraged to elevate privileges with misconfigured RBAC or potential bugs.

We're not going to use Istio's mTLS capabilities because it's not as flexible as tokens. It's primarily meant for intra-cluster communications, so if our identity provider were outside of the cluster, it would be much harder to use. Also, since mTLS requires a point-to-point connection, any TLS termination points would break its use. Since it's rare for an enterprise system to host its own certificate, even outside of Kubernetes, it would be very difficult to implement mTLS between your cluster's services and your identity provider.

Finally, we're not going to use a shared secret between our services and our identity provider because we don't need to. Shared secrets are only needed when you have no other way to give a workload an identity. Since Kubernetes gives every pod its own identity, there's no need to use a client secret to identify our service.

Now that we know how our services will identify themselves to our identity provider, let's walk through an example of using OAuth2 Token Exchange to securely call one service from another.

## Deploying and running the check-writing service

Having walked through much of the theory of using a token exchange to securely call services, let's deploy an example check-writing service. When we call this service, it will call two other services. The first service, `check-funds`, will use the impersonation profile of OAuth2 Token Exchange, while the second service, `pull-funds`, will use delegation. We'll walk through each of these individually. First, use Helm to deploy an identity provider. Go into the `chapter17` directory and run:

```
helm install openunison-service-auth openunison-service-auth -n openunison
NAME: openunison-service-auth
LAST DEPLOYED: Mon Sep 13 01:08:09 2021
NAMESPACE: openunison
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

We're not going to go into the details of OpenUnison's configuration. Suffice it to say, this will set up an identity provider for our services and a way to get an initial token. Next, deploy the `write-checks` service:

```
cd write-checks/
./deploy_write_checks.sh
getting oidc config
getting jwks
namespace/write-checks created
configmap/service-source created
deployment.apps/write-checks created
service/write-checks created
```

```
gateway.networking.istio.io/service-gateway created
virtualservice.networking.istio.io/service-vs created
requestauthentication.security.istio.io/write-checks-auth created
authorizationpolicy.security.istio.io/service-level-az created
```

This should look pretty familiar after the first set of examples in this chapter. We deployed our service as Python in a ConfigMap and the same Istio objects we created in the previous service. The only major difference is in our RequestAuthentication object:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: write-checks-auth
  namespace: write-checks
spec:
  jwtRules:
    - audiences:
        - users
        - checkfunds
        - pullfunds
      forwardOriginalToken: true
      issuer: https://k8sou.apps.192-168-2-119.nip.io/auth/idp/service-idp
      jwksUri: https://k8sou.apps.192-168-2-119.nip.io/auth/idp/service-idp/certs
      outputPayloadToHeader: User-Info
  selector:
    matchLabels:
      app: write-checks
```

There's an additional setting, `forwardOriginalToken`, that tells Istio to send the service the original JWT used to authenticate the call. We'll need this token in order to prove to the identity provider that we should even attempt to perform a token exchange. You can't ask for a new token if you can't provide the original. This keeps someone with access to your service's pod from requesting a token on your behalf with just the service's ServiceAccount.

Earlier in the chapter, we said we couldn't leak a token we didn't have, so we shouldn't have access to the original token. This would be true if we didn't need it to get a token for another service. Following the concept of least privilege, we shouldn't forward the token if we don't need to. In this case, we need it for a token exchange, so it's worth the increased risk to have more secure service-to-service calls.

With our example check-writing service deployed, let's run it and work backward. Just like with our earlier examples, we'll use `curl` to get the token and call our service. In `chapter17/write-checks`, run `call_service.sh`:

```
./call_service.sh
{
```

```
"msg": "hello world!",
"host": "write-checks-84cdbfff74-tgmzh",
"user_jwt": "...",
"pod_jwt": "...",
"impersonated_jwt": "...",
"call_funds_status_code": 200,
"call_funds_text": "{\"funds_available\": true, \"user\": \"mmosley\"}",
"actor_token": "...",
"delegation_token": "...",
"pull_funds_text": "{\"funds_pulled\": true, \"user\": \"mmosley\", \"actor\": \"system:serviceaccount:write-checks:default\"}"
}
```

The output you see is the result of the calls to `/write-check`, which then calls `/check-funds` and `/pull-funds`. Let's walk through each call, the tokens that are generated, and the code that generates them.

# Using Impersonation

We're not talking about the same Impersonation you used in *Chapter 6, Integrating Authentication into Your Cluster*. It's a similar concept, but this is specific to token exchange. When `/write-check` needs to get a token to call `/check-funds`, it asks OpenUnison for a token on behalf of our user, `mmosley`. The important aspect of Impersonation is that there's no reference to the requesting client in the generated token. The `/check-funds` service does not know that the token it's received wasn't retrieved by the user themselves. Working backward, the `impersonated_jwt` in the response to our service call is what `/write-check` uses to call `/check-funds`. Here's the payload after dropping the result into `jwt.io`:

```
    "pwd"
]
}
```

The two important fields here are `sub` and `aud`. The `sub` field tells `/check-funds` who the user is and the `aud` field tells Istio which services can consume this token. Compare this to the payload from the original token in the `user_jwt` response:

```
{
  "iss": "https://k8sou.192-168-2-119.nip.io/auth/idp/service-idp",
  "aud": "users",
  "exp": 1631497059,
  "jti": "C8Qh8iY9FJdFzEO3pLRQzw",
  "iat": 1631496999,
  "nbf": 1631496879,
  "sub": "mmosley",
  "name": " Mosley",
  "groups": [
    "cn=group2,ou=Groups,DC=domain,DC=com",
    "cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"
  ],
  "preferred_username": "mmosley",
  "email": "mmosley@tremolo.dev",
  "amr": [
    "pwd"
  ]
}
```

The original `sub` is the same, but the `aud` is different. The original `aud` is for users, while the impersonated `aud` is for `checkfunds`. This is what differentiates the impersonated token from the original one. While our Istio deployment is configured to accept both audiences for the same service, that's not a guarantee in most production clusters. When we call `/check-funds`, you'll see that, in the output, we echo the user of our token, `mmosley`.

Now that we've seen the end product, let's see how we get it. First, we get the original JWT that was used to call `/write-check`:

```
# let's first get the original JWT. We'll
# use this as an input for impersonation
az_header = request.headers["Authorization"]
user_jwt = az_header[7:]
```

Once we have the original JWT, we need the Pod's `ServiceAccount` token:

```
# next, get the pod's ServiceAccount token
```

```
# so we can identify the pod to the IdP for
# an impersonation token
pod_jwt = Path('/var/run/secrets/kubernetes.io/serviceaccount/token').read_
text()
```

We now have everything we need to get an impersonation token. We'll create a POST body and an Authorization header to authenticate us to OpenUnison to get our token:

```
# with the subject (user) jwt and the pod
# jwt we can now request an impersonated
# token for our user from openunison
impersonation_request = {
    "grant_type": "urn:ietf:params:oauth:grant-type:token-exchange",
    "audience": "checkfunds",
    "subject_token": user_jwt,
    "subject_token_type": "urn:ietf:params:oauth:token-type:id_token",
    "client_id": "sts-impersonation"
}
impersonation_headers = {
    "Authorization": "Bearer %s" % pod_jwt
}
```

The first data structure we created is the body of an HTTP POST that will tell OpenUnison to generate an impersonation token for the `clientfunds` aud, using our existing user (`user_jwt`). OpenUnison will authenticate our service by verifying the JWT sent in the Authorization header as a Bearer token, using the TokenReview API.

OpenUnison will then apply its internal policy to verify that our service is able to generate a token for `mmosley` for the `clientfunds` audience, and then generate an `access_token`, `id_token`, and `refresh_token`. We'll use the `id_token` to call `/check-funds`:

```
resp = requests.post("https://k8sou.apps.IPADDR.nip.io/auth/idp/service-idp/to-
ken", verify=False, data=impersonation_request, headers=impersonation_headers)
response_payload = json.loads(resp.text)
impersonated_id_token = response_payload["id_token"]
# with the impersonated user's id_token, call another
# service as that user
call_funds_headers = {
    "Authorization": "Bearer %s" % impersonated_id_token
}
resp = requests.get("http://write-checks.IPADDR.nip.io/check-funds", verify=-
False, headers=call_funds_headers)
```

Since the final JWT makes no mention of the impersonation, how do we track a request back to our service? Hopefully, you're piping your logs into a centralized logging system. If we look at the `jti` claim of our impersonation token, we can find the impersonation call in the OpenUnison logs:

```
INFO AccessLog - [AzSuccess] - service-idp - https://k8sou.apps.192-168-2-119.nip.io/auth/idp/service-idp/token - username=system:serviceaccount:write-checks:default,ou=oauth2,o=Tremolo - client 'sts-impersonation' impersonating 'mmosley', jti : 'C8Qh8iY9FJdFzE03pLRQzw'
```

So, we at least have a way of tying them together. We can see that our Pod's service account was authorized to create the impersonation token for `mmosley`.

Having worked through an example of impersonation, let's cover token delegation next.

## Using delegation

In the last example, we used impersonation to generate a new token on behalf of our user, but our downstream service had no knowledge that the impersonation happened. Delegation is different in that the token carries information about both the original user and the service, or actor, that requested it.

This means that the service being called knows both the originator of the call and the service that makes the call. We can see this in the `pull_funds_text` value from the response of our `call_service.sh` run. It contains both our original user, `mmosley`, and the `ServiceAccount` for the service that made the call, `system:serviceaccount:write-checks:default`. Just as with impersonation, let's look at the generated token:

```
{
  "iss": "https://k8sou.apps.192-168-2-119.nip.io/auth/idp/service-idp",
  "aud": "pullfunds",
  "exp": 1631497059,
  "jti": "xkaQhMgKgRvGBqAsOWDlXA",
  "iat": 1631496999,
  "nbf": 1631496879,
  "nonce": "272f1900-f9d9-4161-a31c-6c6dde80fcb9",
  "sub": "mmosley",
  "amr": [
    "pwd"
  ],
  "name": " Mosley",
  "groups": [
    "cn=group2,ou=Groups,DC=domain,DC=com",
    "cn=k8s-cluster-admins,ou=Groups,DC=domain,DC=com"
  ],
  "preferred_username": "mmosley",
  "email": "mmosley@tremolo.dev",
```

```

"act": {
  "sub": "system:serviceaccount:write-checks:default",
  "amr": [
    "k8s-sa"
  ],
  .
  .
  .
}
}

```

In addition to the claims that identify the user as `mmosley`, there's an `act` claim that identifies the `ServiceAccount` that's used by `/write-checks`. Our service can make additional authorization decisions based on this claim or simply log it, noting that the token it received was delegated to a different service. In order to generate this token, we start by getting the original subject's JWT and the Pod's `ServiceAccount` token.

Instead of calling OpenUnison for a delegated token, our client first has to get an actor token by using the `client_credentials` grant. This will get us the token that will eventually go into the `act` claim:

```

client_credentials_grant_request = {
  "grant_type": "client_credentials",
  "client_id" : "sts-delegation"
}
delegation_headers = {
  "Authorization": "Bearer %s" % pod_jwt
}
resp = requests.post("https://k8sou.IPADDR.nip.io/auth/idp/service-idp/token", -
verify=False,data=client_credentials_grant_request,headers=delegation_headers)
response_payload = json.loads(resp.text)
actor_token = response_payload["id_token"]

```

We authenticate to OpenUnison using our Pod's native identity. OpenUnison returns an `access_token` and an `id_token`, but we only need the `id_token`. With our actor token in hand, we can now get our delegation token:

```

delegation_request = {
  "grant_type":"urn:ietf:params:oauth:grant-type:token-exchange",
  "audience":"pullfunds",
  "subject_token":user_jwt,
  "subject_token_type":"urn:ietf:params:oauth:token-type:id_token",
  "client_id":"sts-delegation",
  "actor_token": actor_token,
  "actor_token_type": "urn:ietf:params:oauth:token-type:id_token"
}

```

```
    }
    resp = requests.post("https://k8sou.IPADDR.nip.io/auth/idp/service-idp/token",-
        verify=False,data=delegation_request)
    response_payload = json.loads(resp.text)
    delegation_token = response_payload["id_token"]
```

Similarly to impersonation, in this call, we not only send the original user's token (`user_jwt`) but also the `actor_token` we just received from OpenUnison. We also don't send an Authorization header. The `actor_token` authenticates us already. Finally, we're able to use our returned token to call `/pull-funds`.

Now that we've looked at the most correct way to call services, using both impersonation and delegation, let's take a look at some anti-patterns and why you shouldn't use them.

## Passing tokens between services

Whereas, in the previous section, we used an identity provider to generate either impersonation or delegation tokens, this method skips that and just passes the original token from service to service. This is a simple approach that's easy to implement. It also creates a larger blast radius. If the token gets leaked (and given that it's now being passed to multiple services, the likelihood of it leaking goes up quite a bit), you've now not only exposed one service; you've also exposed all the services that trust that token.

While using OAuth2 Token Exchange does require more work, it will limit your blast radius should a token be leaked. Next, we'll look at how you can simply tell a downstream service who's calling it.

## Using simple impersonation

Where the previous examples of service-to-service calls rely on a third party to generate a token for a user, direct impersonation is where your service's code uses a service account (in the generic sense, not the Kubernetes version) to call the second service and just tells the service who the user is as an input to the call. For instance, instead of calling OpenUnison to get a new token, `/write-check` could have just used the Pod's ServiceAccount token to call `/check-funds`, with a parameter containing the user's ID. Something like the following would work:

```
call_headers = {
    "Authorization": "Bearer %s" % pod_jwt
}
resp = requests.post("https://write-checks.IPADDR.nip.io/check-funds?user=mmos-
    ley",verify=False,data=impersonation_request,headers=call_headers)
```

This is, again, very simple. You can tell Istio to authenticate a Kubernetes ServiceAccount. This takes two lines of code to do something that took 15 to 20 lines using a token service. Just like with passing tokens between services, this approach leaves you exposed in multiple ways. First, if anyone gets the ServiceAccount used by our service, they can impersonate anyone they want without checks. Using the token service ensures that a compromised service account doesn't lead to it being used to impersonate anyone.

You might find this method very similar to the impersonation we used in *Chapter 6, Integrating Authentication into Your Cluster*. You’re correct. While this uses the same mechanism, a `ServiceAccount` and some parameters to specify who the user is, the type of impersonation Kubernetes uses for the API server is often referred to as a **protocol transition**. This is used when you are moving from one protocol (OpenID Connect) to another (a Kubernetes service account). As we discussed in *Chapter 5*, there are several controls you can put in place with Kubernetes impersonation, including using `NetworkPolicies`, RBAC, and the `TokenRequest` API. It’s also a much more isolated use case than a generic service.

We’ve walked through multiple ways for services to call and authenticate each other. While it may not be the simplest way to secure access between services, it will limit the impact of a leaked token. Now that we know how our services will talk to each other, the last topic we need to cover is the relationship between Istio and API gateways.

## Do I need an API gateway?

If you’re using Istio, do you still need an API gateway? In the past, Istio was primarily concerned with routing traffic for services. It got traffic into the cluster and figured out where to route it to. API gateways have typically focused more on application-level functionality such as authentication, authorization, input validation, and logging.

For example, earlier in this chapter, we identified schema input validation as a process that needs to be repeated for each call and shouldn’t need to be done manually. This is important to protect against attacks that can leverage unexpected input, and it also makes for a better developer experience, providing feedback to developers sooner in the integration process. This is a common function for API gateways but is not available in Istio.

Another example of a function that is not built into Istio but is common for API gateways is logging authentication and authorization decisions and information. Throughout this chapter, we leveraged Istio’s built-in authentication and authorization to validate service access, but Istio makes no record of that decision, other than that a decision was made. It doesn’t record who accessed a particular URL, only where it was accessed from. Logging who accessed a service, from an identity standpoint, is left to each individual service. This is a common function for API gateways.

Finally, API gateways are able to handle more complex transformations. Gateways will typically provide functionality for mapping inputs and outputs, or even integrating with legacy systems.

These functions could all be integrated into Istio, either directly or via Envoy filters. We saw an example of this when we looked at using OPA to make more complex authorization decisions than what the `AuthorizationPolicy` object provides. However, over the last few releases, Istio has moved further into the realm of traditional API gateways, and API gateways have begun taking on more service mesh capabilities. I suspect there will be considerable overlap between these systems in the future, but at the time of writing, Istio isn’t yet capable of fulfilling all the functions of an API gateway.

We’ve had quite the journey building out the services for our Istio service mesh. You should now have the tools you need to begin building services in your own cluster.

## Summary

In this chapter, we learned how both monoliths and microservices run in Istio. We explored why and when to use each approach. We deployed a monolith, taking care to ensure our monolith's session management worked. We then moved into deploying microservices, authenticating requests, authorizing requests, and finally, how services can securely communicate. To wrap things up, we discussed whether an API gateway is still necessary when using Istio.

Istio can be complex, but when used properly, it can provide considerable power. What we didn't cover in this chapter is how to build containers and manage the deployment of our services. We're going to tackle that next, in *Chapter 18, Provisioning a Multitenant Platform*.

## Questions

1. True or false: Istio is an API Gateway.
  - a. True
  - b. False

Answer: b – False. Istio is a service mesh, and while it has many of the functions of a gateway, it doesn't have all of them (such as schema checking).

2. Should I always build applications as microservices?
  - a. Obviously – this is the right way.
  - b. Only if a microservices architecture aligns with your organization's structure and needs.
  - c. No. Microservices are more trouble than they're worth.
  - d. What's a microservice?

Answer: b – Microservices are great when you have a team that is able to make use of the granularity they provide.

3. What is a monolith?
  - a. A large object that appears to be made from a single piece by an unknown maker
  - b. An application that is self-contained
  - c. A system that won't run on Kubernetes
  - d. A product from a new start-up

Answer: b – A monolith is a self-contained application that can run quite well on Kubernetes.

4. How should you authorize access to your services in Istio?
  - a. You can write a rule that limits access in Istio by a claim in the token.
  - b. You can integrate OPA with Istio for more complex authorization decisions.
  - c. You can embed complex authorization decisions in your code.
  - d. All of the above.

Answer: d – These are all valid strategies from a technical standpoint. Each situation is different, so look at each one to determine which one is best for you!

5. True or false: Calling services on behalf of a user without token exchange is a secure approach.
  - a. True
  - b. False

Answer: b. False – Without using token exchange to get a new token for when the user uses the next service, you leave yourself open to various attacks because you can't limit calls or track them.

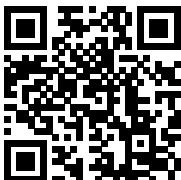
6. True or false: Istio supports sticky sessions.
  - a. True
  - b. False

Answer: a. True – They are not a default, but they are supported.

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>



# 18

## Provisioning a Multitenant Platform

Every chapter in this book, up until this point, has focused on the infrastructure of your cluster. We have explored how to deploy Kubernetes, how to secure it, and how to monitor it. What we haven't talked about is how to deploy applications.

In these, our final chapters, we're going to work on building an application deployment platform using what we've learned about Kubernetes. We're going to build our platform based on some common enterprise requirements. Where we can't directly implement a requirement, because building a platform on Kubernetes could fill its own book, we'll call it out and provide some insights.

In this chapter, we will cover the following topics:

- Designing a pipeline
- Designing our platform architecture
- Using Infrastructure as Code for deployment
- Automating tenant onboarding
- Considerations for building an Internal Developer Platform

You'll have a good conceptual starting point for building out your own GitOps platform on Kubernetes by the end of this chapter. We're going to use the concepts we cover in this chapter to drive how we build our Internal Developer Portal in the final chapter.

### Technical requirements

This chapter will be all theory and concepts. We're going to cover implementation in the final chapter.

## Designing a pipeline

The term **pipeline** is used extensively in the Kubernetes and DevOps world. Very simply, a pipeline is a process, usually automated, that takes code and gets it running. This usually involves the following:

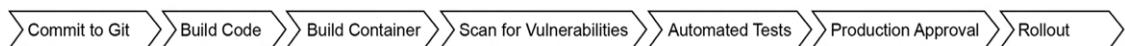


*Figure 18.1: A simple pipeline*

Let's quickly run through the steps involved in this process:

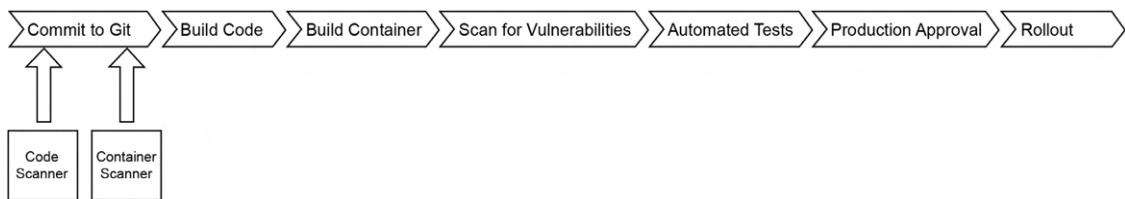
1. Storing the source code in a central repository, usually Git
2. When code is committed, building it and generating artifacts, usually a container
3. Telling the platform – in this case, Kubernetes – to roll out the new containers and shut down the old ones

This is about as basic as a pipeline can get and isn't of much use in most deployments. In addition to building our code and deploying it, we want to make sure we scan containers for known vulnerabilities. We may also want to run our containers through some automated testing before going into production. In enterprise deployments, there's often a compliance requirement where someone takes responsibility for the move to production as well. Taking this into account, the pipeline starts to become more complex.



*Figure 18.2: Pipeline with common enterprise requirements*

The pipeline has added some extra steps, but it's still linear with one starting point, a commit. This is also very simplistic and unrealistic. The base containers and libraries your applications are built on are constantly being updated as new **Common Vulnerabilities and Exposures** (CVEs), a common way to catalog and identify security vulnerabilities, are discovered and patched. In addition to having developers who are updating application code for new requirements, you will want to have a system in place that scans both the code and the base containers for available updates. These scanners watch your base containers and can do something to trigger a build once a new base container is ready. While the scanners could call an API to trigger a pipeline, your pipeline is already waiting on your Git repository to do something, so it would be better to simply add a commit or a pull request to your Git repository to trigger the pipeline.



*Figure 18.3: Pipeline with scanners integrated*

This means your application code is tracked and your operational updates are tracked in Git. Git is now the source of truth for not only what your application code is but also operation updates. When it's time to go through your audits, you have a ready-made change log! If your policies require you to enter changes into a change management system, simply export the changes from Git.

So far, we have focused on our application code and just put **Rollout** at the end of our pipeline. The final rollout step usually means patching a Deployment or StatefulSet with our newly built container, letting Kubernetes do the work of spinning up new pods and scaling down the old ones. This could be done with a simple API call, but how are we tracking and auditing that change? What's the source of truth?

Our application in Kubernetes is defined as a series of objects stored in etcd that are generally represented as code using YAML files. Why not store those files in a Git repository too? This gives us the same benefits as storing our application code in Git. We have a single source of truth for both the application source and the operations of our application! Now, our pipeline involves some more steps.

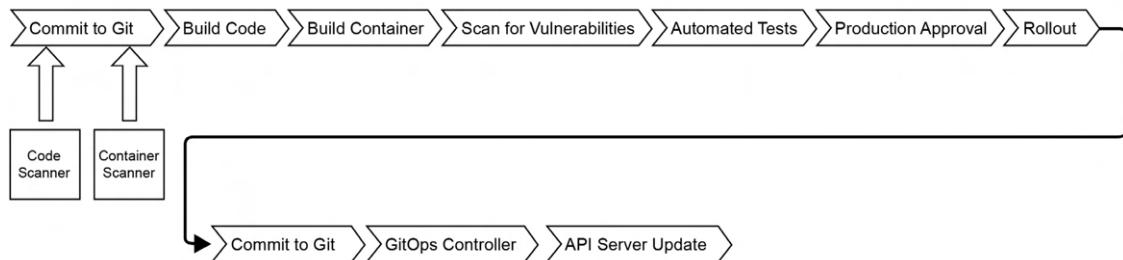


Figure 18.4: GitOps pipeline

In this diagram, our rollout updates a Git repository with our application's Kubernetes YAML. A controller inside our cluster watches for updates to Git and when it sees them, gets the cluster in sync with what's in Git. It can also detect drift in our cluster and bring it back to alignment with our source of truth.

This focus on Git is called **GitOps**. The idea is that all of the work of an application is done via code, not directly via APIs. How strict you are with this idea can dictate what your platform looks like. Next, we'll explore how opinions can shape your platform.

## Opinionated platforms

Kelsey Hightower, a developer advocate for Google and leader in the Kubernetes world, once said: "Kubernetes is a platform for building platforms. It's a better place to start; not the endgame." When you look at the landscape of vendors and projects building Kubernetes-based products, they all have their own opinions of how systems should be built. As an example, Red Hat's **OpenShift Container Platform (OCP)** wants to be a one-stop shop for multi-tenant enterprise deployment. It builds in a great deal of the pipeline we discussed. You define a pipeline that is triggered by a commit, which builds a container and pushes it into its own internal registry that then triggers a rollout of the new container. Namespaces are the boundaries of tenants. Canonical is a minimalist distribution that doesn't include any pipeline components. Managed vendors such as Amazon, Azure, and Google provide the building blocks of a cluster and the hosted build tools of a pipeline, but leave it to you to build out your platform.

There is no correct answer as to which platform to use. Each is opinionated and the right one for your deployment will depend on your own requirements. Depending on the size of your enterprise, it wouldn't be surprising to see more than one platform deployed!

Having looked at the idea of opinionated platforms, let's explore the security impacts of building a pipeline.

## Securing your pipeline

Depending on your starting point, this can get complex quickly. How much of your pipeline is one integrated system, or could it be described using a colorful American colloquialism involving duct tape? Even in platforms where all the components are there, tying them together can often mean building a complex system. Most of the systems that are part of your pipeline will have a visual component. Usually, the visual component is a dashboard. Users and developers may need access to that dashboard. You don't want to maintain separate accounts for all those systems, do you? You'll want to have one login point and portal for all the components of your pipeline.

After determining how to authenticate the users who use these systems, the next question is how to automate the rollout. Each component of your pipeline requires configuration. It can be as simple as an object that gets created via an API call or as complex as tying together a Git repo and build process with SSH keys to automate security. In such a complex environment, manually creating pipeline infrastructure will lead to security gaps. It will also lead to impossible-to-manage systems. Automating the process and providing consistency will help you both secure your infrastructure and keep it maintainable.

Finally, it's important to understand the implications of GitOps on our cluster from a security standpoint. We discussed authenticating administrators and developers to use the Kubernetes API and authorizing access to different APIs in *Chapter 6, Integrating Authentication into Your Cluster*, and *Chapter 7, RBAC Policies and Auditing*. What is the impact if someone can check in a `RoleBinding` that assigns them the `admin` `ClusterRole` for a namespace and a GitOps controller automatically pushes it through to the cluster? As you design your platform, consider how developers and administrators will want to interact with it. It's tempting to say "Let everyone interact with their application's Git registry," but that means putting the burden on you as the cluster owner for many requests. As we discussed in *Chapter 7, RBAC Policies and Auditing*, this could make your team the bottleneck in an enterprise. Understanding your customers, in this case, is important in knowing how they want to interact with their operations even if it's not how you intended.

Having touched on some of the security aspects of GitOps and a pipeline, let's explore the requirements for a typical platform and how we will build it.

## Building our platform's requirements

Kubernetes deployments, especially in enterprise settings, will often have the following basic requirements:

- **Development and test environments:** At least two clusters to test the impacts of changes on the cluster level on applications
- **Developer sandbox:** A place where developers can build containers and test them without worrying about impacts on shared namespaces
- **Source control and issue tracking:** A place to store code and track open tasks

In addition to these basic requirements, enterprises will often have additional requirements, such as regular access reviews, limiting access based on policy, and workflows that assign responsibility for actions that could impact a shared environment. Finally, you'll want to make sure that policies are in place to protect nodes.

For our platform, we want to encompass as many of these requirements as possible. To better automate deployments onto our platform, we're going to define each application as having the following:

- **A development namespace:** Developers are administrators
- **A production namespace:** Developers are viewers
- **A source control project:** Developers can fork
- **A build process:** Triggered by updates to Git
- **A deploy process:** Triggered by updates to Git

In addition, we want our developers to have their own sandbox so that each user will get their own namespace for development.

To provide access to each application, we will define three roles:

- **Owners:** Users who are application owners can approve access for other roles inside their application. This role is assigned to the application requestor and can be assigned by application owners. Owners are also responsible for pushing changes into development and production.
- **Developers:** These are users who will have access to an application's source control and can administer the application's development namespace. They can view objects in the production namespace but can't edit anything. This role can be requested by any user and is approved by an application owner.
- **Operations:** These users have the same capabilities as developers, but can also make changes to the production namespace as needed. This role can be requested by any user and is approved by the application owner.

We will also create some environment-wide roles:

- **System approvers:** Users with this role can approve access to any system-wide roles.
- **Cluster administrators:** This role is specifically for managing our clusters and the applications that comprise our platform. It can be requested by anyone and must be approved by a member of the system approvers role.
- **Developers:** Anyone who logs in gets their own namespace for development on the development cluster. These namespaces cannot be requested for access by other users. These namespaces are not directly connected to any CI/CD infrastructure or Git repositories.

Even with our very simple platform, we have six roles that need to be mapped to the applications that make up our pipeline. Each application has its own authentication and authorization processes that these roles will need to be mapped to. This is just one example of why automation is so important to the security of your clusters. Provisioning this access manually based on email requests can become unmanageable quickly.

The workflow that developers are expected to go through with an application will line up with the GitOps flow we designed previously:

- Application owners will request that an application is created. Once approved, a Git repository will be created for application code, and Kubernetes manifests. Development and production namespaces will be created in the appropriate clusters, with the appropriate RoleBinding objects. Groups will be created that reflect the roles for each application, with approval for access to those groups delegated to the application owner.
- Developers and operations staff are granted access to the application by either requesting it or having it provided directly by an application owner. Once granted access, updates are expected in both the developer's sandbox and the development namespace. Updates are made in a user's fork for the Git repository, with pull requests used to merge code into the main repositories that drive automation.
  - All builds are controlled via scripts in the application's source control.
  - All artifacts are published to a centralized container registry.
  - All production updates must be approved by application owners.

This basic workflow doesn't include typical components of a workflow, such as code and container scans, periodic access recertifications, or requirements for privileged access. The topic of this chapter could easily be a complete book on its own. The goal isn't to build a complete enterprise platform but to give you a starting point for building and designing your own system.

## Choosing our technology stack

In the previous parts of this section, we talked about pipelines and platforms in a generic way. Now, let's get into the specifics of what technology is needed in our pipeline. We identified earlier that every application has application source code and Kubernetes manifest definitions. It also has to build containers. There needs to be a way to watch for changes to Git and update our cluster. Finally, we need an automation platform so that all these components work together.

Based on our requirements for our platform, we want technology that has the following features:

1. **Open source:** We don't want you to buy anything just for this book!
2. **API-driven:** We need to be able to provide components and access in an automated way
3. **A visual component that supports external authentication:** This book focuses on enterprise, and everyone in the enterprise loves their GUIs, just not having different credentials for each application
4. **Supported on Kubernetes:** This is a book on Kubernetes

To meet these requirements, we're going to deploy the following components to our cluster:

- **Git Registry – GitLab:** GitLab is a powerful system that provides a great UI and experience for working with Git that supports external authentication (that is, **Single Sign-On (SSO)**). It has integrated issue management and an extensive API. It also has a Helm chart that we have tailored for the book to run a minimal install.
- **Automated Builds – GitLab:** GitLab is designed to be a development monolith. Given that it has an integrated pipeline system that is Kubernetes native, we're going to use it instead of an external system like Jenkins or TektonCD.
- **Container Registry – Harbor:** In past editions, we've used a simple Docker registry, but since we're going to be building out a multi-cluster environment, it's important that we use a container registry that is designed for production use. Harbor gives us the ability to store our containers and manage them via a web UI that supports OpenID Connect for authentication and has an API for management.
- **GitOps – ArgoCD:** ArgoCD is a project from Intuit to build a feature-rich GitOps platform. It's Kubernetes native, has its own API, and stores its objects as Kubernetes custom resources, making it easier to automate. Its UI and CLI tools both integrate with SSO using OpenID Connect.
- **Access, authentication, and automation – OpenUnison:** We'll continue to use OpenUnison for authentication into our cluster. We're also going to integrate the UI components of our technology stack to provide a single portal for our platform. Finally, we'll use OpenUnison's workflows to manage access to each system based on our role structure and provision the objects needed for everything to work together. Access will be provided via OpenUnison's self-service portal.
- **Node Policy Enforcement – GateKeeper:** The GateKeeper deployment from *Chapter 12, Node Security with Gatekeeper*, will enforce the fact that each namespace has a minimum set of policies.
- **Tenant Isolation – vCluster:** We used vCluster to provide each tenant with their own virtual cluster in *Chapter 9*. We'll build on this to provide individual tenants with their own virtual clusters so they can better control their environment.
- **Secrets Management – HashiCorp Vault:** We already know how to deploy a vCluster with Vault, so we'll continue to use it to externalize our secrets.

Reading through this technology stack, you might ask “Why didn't you choose XYZ?” The Kubernetes ecosystem is diverse, with no shortage of great projects and products for your cluster. This is by no means a definitive stack, nor is it even a “recommended” stack. It's a collection of applications that meets our requirements and lets us focus on the processes being implemented, rather than learning a specific technology.

You might also find that there's quite a bit of overlap between even the tools in this stack. For instance, GitLab could be used for more than Git and pipelines, but we wanted to show how different components integrate with each other. It's not unusual, especially in an enterprise where components are managed by different organizations, to only use a system for what the group that uses it specializes in. For instance, a group that specializes in GitLab may not want you using it as your identity provider because they're not in the identity provider business even though GitLab has this capability. They don't want to support it.

Finally, you'll notice that Backstage isn't mentioned. Backstage is a popular open-source internal developer platform that is often associated with any project related to "platform engineering." We decided not to use Backstage to build out our platform because there's no way to cover it in a single chapter! There have been multiple books written about Backstage and it's a topic that requires a considerable amount of its own analysis to handle properly. The goal of these next two chapters is to help see how many of the technologies we've built through this book come together. It's meant as a starting point, not a complete solution. If you want to integrate Backstage or any other of the internal developer platform systems, you'll find your approach won't be very different.

With our technology stack in hand, the next step is to see how we will integrate these components.

## Designing our platform architecture

In previous chapters, all of our work centered around a single cluster. This made the labs easier, but the reality of the world in IT doesn't work that way. You want to separate out your development and production clusters at a minimum, not only so you can isolate the workloads, but so that you can test your operations processes outside of production. You may need to isolate clusters for other risk- and policy-based reasons as well. For instance, if your enterprise spans multiple nations, you may need to respect each nation's data sovereignty laws and run workloads on infrastructure in that nation. If you are in a regulated industry that requires different levels of security for different kinds of data, you may need to separate your clusters. For this, and many reasons, these two chapters will move beyond a single cluster into a multiple cluster design.

To keep things simple, we're going to assume we can have one cluster for development and one cluster for production. There is a problem with this design though: where do you deploy all the technology in our management stack? They're "production" systems, so you might want to deploy them onto the production cluster, but since these are generally privileged systems, that might cause an issue with security and policy. Since many of the systems are development-related, you may think they should be deployed into the development cluster. This can also be an issue because you don't want a development system to be in control of a production system.

In order to solve these issues, we're going to add a third cluster that will be our "control plane" cluster. This cluster will host OpenUnison, GitLab, Harbor, ArgoCD, and Vault. That will leave tenants on the development and production clusters. Each tenant will run a vCluster in its namespace and that vCluster will run its own OpenUnison, as we did in the vCluster chapter. This leaves our architecture as:

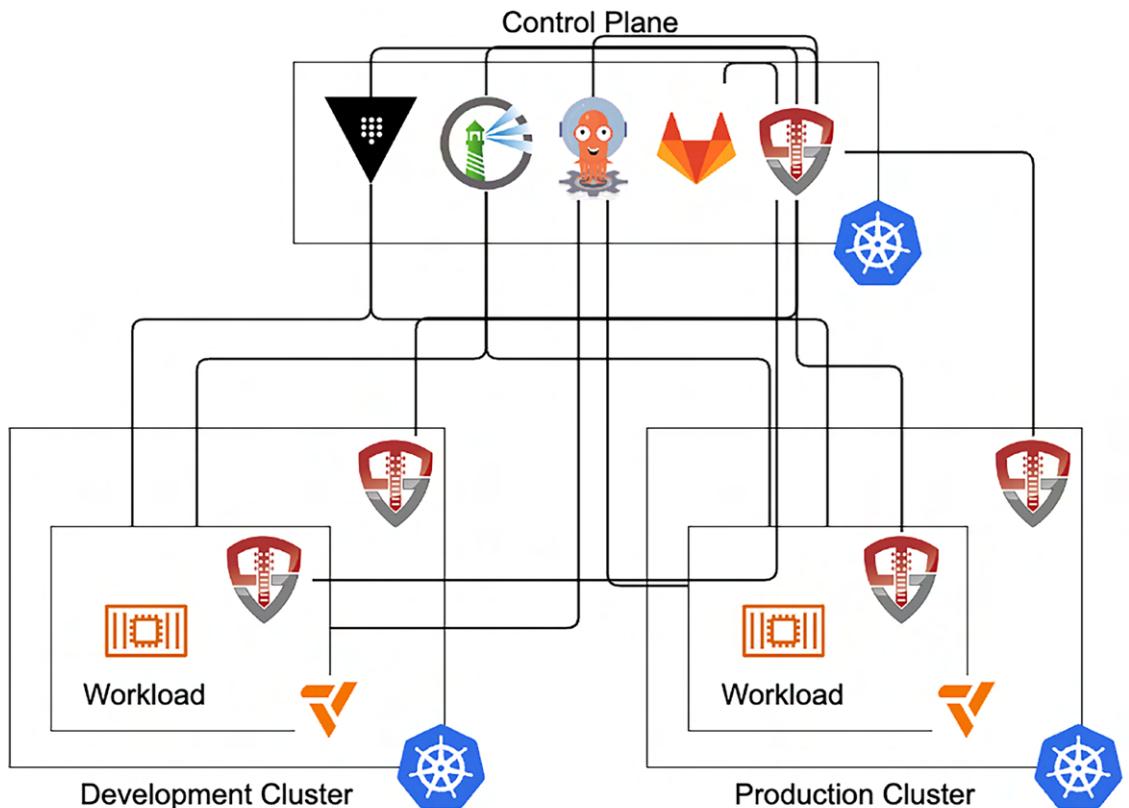


Figure 18.5: Developer platform architecture

Looking at our diagram, you can see that we've created a fairly complex infrastructure. That said, because we're taking advantage of multitenancy, it's much simpler than it could be. If each tenant had its own cluster and related infrastructure, you would need to manage and update all those systems. Throughout this book, we've had a focus on identity as an important security boundary. We've already discussed how OpenUnison and Kubernetes should interact with Vault, but what about the other components of our stack?

## Securely managing a remote Kubernetes cluster

In *Chapter 6, Integrating Authentication into Your Cluster*, we covered how external pipelines should securely communicate with Kubernetes clusters. We used the example of a pipeline generating an identity for a remote cluster based on its own identity issued by Kubernetes to each pod or by using a credential issued by Active Directory. We didn't use a ServiceAccount token for the remote cluster though, identifying this approach as an anti-pattern in Kubernetes. ServiceAccount tokens were never meant to be used as a credential for the cluster from outside the cluster, and while since Kubernetes 1.24 the default has been to generate tokens with a finite time to live, it still requires token rotation and violates the reason for having ServiceAccounts.

We're going to avoid this anti-pattern by relying on OpenUnison's built-in capabilities as an identity provider. When we integrate a node or tenant cluster into our control plane OpenUnison, an instance of kube-oidc-proxy is deployed that trusts OpenUnison. Then, when OpenUnison needs to issue an API call to one of the node or tenant clusters, it can do so with a short-lived token.

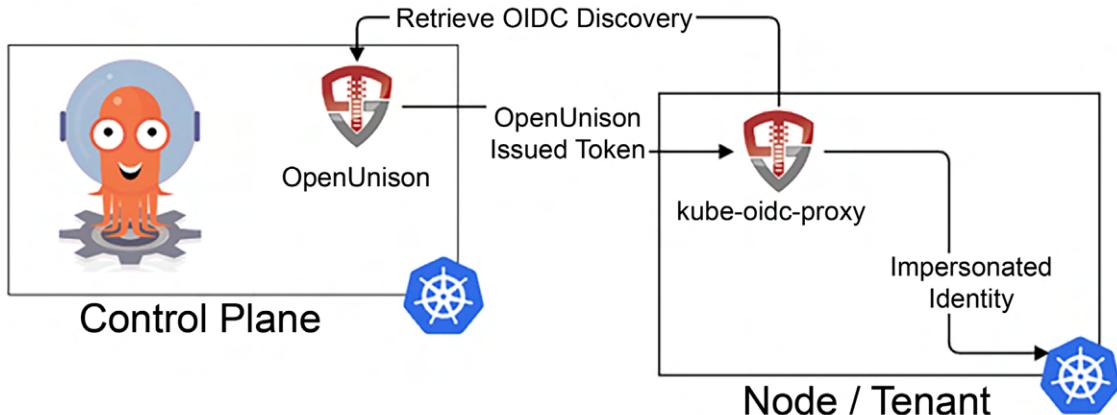


Figure 18.6: Control plane API integration with tenants and nodes

In *Figure 18.6*, our tenant cluster runs an instance of kube-oidc-proxy that is configured to trust an identity provider configured in the control plane cluster's OpenUnison. In this context, we're calling the proxy a "management" proxy since it's only going to be used by OpenUnison and ArgoCD to interact with the node or tenant's API server. When OpenUnison wants to call the remote API, it first generates a one-minute-lived token for the call. This lines up with our goal to use short-lived tokens to communicate with remote clusters. This way, if the token is leaked, it will likely be useless once the token is obtained by an attacker.

We're using kube-oidc-proxy because Kubernetes, prior to 1.29, only supported a single OpenID Connect issuer. Starting in 1.29, Kubernetes introduced as an alpha feature the ability to define multiple token issuers, eliminating the need for an impersonating proxy for this use case. We decided not to use this feature because:

1. It's still an alpha feature at the time of writing and is likely to change.
2. Even once it goes GA, the feature is not implemented as an API, but as a static configuration that must be deployed to each control plane. This is similar to how clusters are configured using API Server command line flags and will impose similar challenges on managed clusters.

For these reasons, we decided not to include this feature in our design and are instead going to rely on kube-oidc-proxy.

Now that we know how OpenUnison will interact with remote clusters, we need to think about how ArgoCD will interact with remote clusters, too. Similar to OpenUnison, it needs to be able to call the APIs of our tenants and node clusters. Just as with OpenUnison, we don't want to use a static ServiceAccount token. Thankfully, we have all the components we need to make this work.

Since OpenUnison is already capable of generating a short-lived token that is trusted by our remote clusters, what we need now is a way to securely get that token into ArgoCD when it needs it, and to tell ArgoCD to use it. Since ArgoCD uses the client-go SDK for Kubernetes, it's able to use a credential plugin that can call a remote API to retrieve a credential. In this case, we're going to use a similar pattern that we used in *Chapter 6, Integrating Authentication into Your Cluster*, to generate a token. Instead of using an Active Directory credential, we're going to use the identity of the ArgoCD controller pod to generate the needed token:

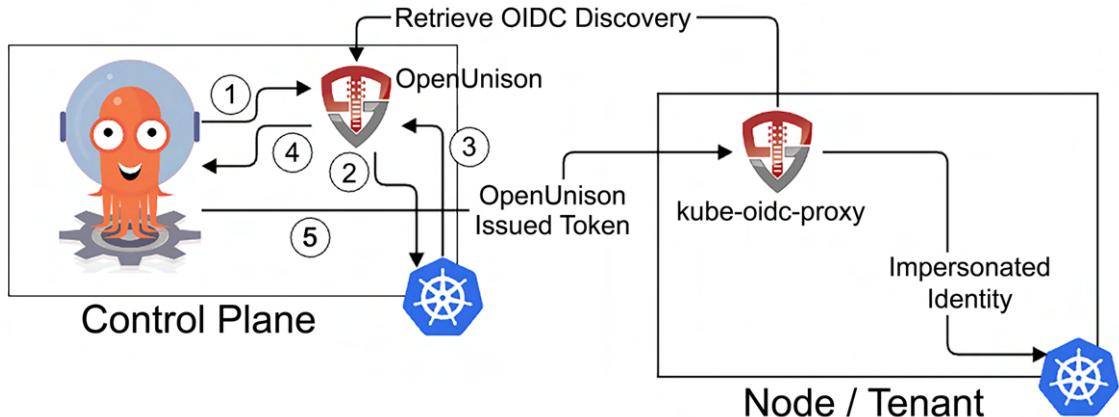


Figure 18.7: ArgoCD integration with tenants and nodes using short-lived credentials

We're able to leverage a credential plugin for go-sdk. In step 1, we generate an HTTP request to a service we deployed into OpenUnison with our Pod's credentials to get a token:

```
#!/bin/bash

OPENUNISON_CP_HOST=$1
CLUSTER_NAME=$2
PATH_TO_POD_TOKEN=$3

REMOTE_TOKEN=$(curl -H "Authorization: Bearer $(<$PATH_TO_TOKEN)" https://$OPENUNISON_CP_HOST/api/get-target-token?targetName=$CLUSTER_NAME 2>/dev/null)
```

This is a snippet from our credential provider plugin. Credential providers are simply executables, such as this bash script, that the SDK calls to get a token or certificate and private key. We'll tell ArgoCD to pass the control plane's OpenUnison host, the name of the cluster in OpenUnison we want to work with, and the path to our Pod's token. The script then makes a curl call with that token to OpenUnison to get the token, using the Pod's identity as a bearer token.

When the request hits OpenUnison, it will run step 2, where OpenUnison will issue a TokenRequest to the API server to make sure that the token provided in the API call is valid. In order for this to succeed, the token must not have expired yet and the pod bound to the token must still be running. If someone were to obtain a token from an expired pod, but that hasn't yet expired, this call would still fail. At this point, the request is authenticated and step 3 has the API server returning its determination to OpenUnison. We don't want just any identity to allow for getting a token for our remote clusters.

Next, OpenUnison needs to authorize the request. In our Application configuration for our API, we defined the azRule as `(sub=system:serviceaccount:argocd:argocd-application-controller)`, making sure that only the controller pod is able to get a token for our remote clusters. This will make sure that if there is a breach of the ArgoCD web interface, an attacker can't just generate a token with the identity of that pod. They'd also need to get into the application controller pod.

With the request authenticated and authorized, step 4 has OpenUnison look up the target and return a generated token. Finally, in step 5, we generate some JSON in our credential plugin that tells the client-go SDK what token to use:

```
echo -n "{\"apiVersion\": \"client.authentication.k8s.io/v1\", \"kind\": \"ExecCredential\", \"status\": {\"token\": \"$REMOTE_TOKEN\"}}"
```

Once the token is obtained by ArgoCD, it will use it while interacting with our remote clusters. We've designed a way for our platform to use centralized ArgoCD while not relying on long-lived credentials!

We're not quite done yet though. ArgoCD is configured to use our token in a two-step process:

1. Define a Secret that contains the cluster connection configuration and define a label to identify it
2. Create an ApplicationSet that specifies the target cluster

For instance, here's an example Secret:

```
---
apiVersion: v1
kind: Secret
metadata:
  name: k8s-kubernetes-satelite
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
    tremolo.io/clustername: k8s-kubernetes-satelite
type: Opaque
stringData:
  name: k8s-kubernetes-satelite
  server: https://oumgmt-proxy.idp-dev.tremolo.dev
  config: |
```

```
"execProviderConfig": {
    "command": "/custom-tools/remote-token.sh",
    "args": ["k8sou.idp-cp.tremolo.dev","k8s-kubernetes-satelite","/var/run/secrets/ubernetes.io/serviceaccount/token"],
    "apiVersion": "client.authentication.k8s.io/v1"
},
"tlsClientConfig": {
    "insecure": false,
    "caData": "LS0tL...
}
}
```

You can see that our configuration doesn't include any secret information! The label `acrocd.argoproj.io/secret-type: cluster` is what tells ArgoCD this Secret is used to configure a remote cluster. The additional label `tremolo.io/clustername` is how we know which cluster to support. Then, we define an ApplicationSet that ArgoCD's operator will use to generate an ArgoCD Application object and a cluster configuration:

```
---
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: test-remote-cluster
  namespace: argocd
spec:
  goTemplate: true
  goTemplateOptions: ["missingkey=error"]
  generators:
  - clusters:
    selector:
      matchLabels:
        tremolo.io/clustername: k8s-kubernetes-satelite
  template:
    metadata:
      name: '{{.name}}-guestbook' # 'name' field of the Secret
  spec:
    project: "default"
    source:
      repoURL: https://github.com/mlbiam/test-argocd-repo.git
      targetRevision: HEAD
      path: yaml
      directory:
```

```
reurse: true
destination:
  server: '{{.server}}' # 'server' field of the secret
  namespace: myns
```

The `spec.generators[0]` identifies a `clusters` generator, which matches the label `tremolo.io/clustername: k8s-kubernetes-satelite`. Since we're building a multi-tenant platform, we need to make sure that we define a GateKeeper policy that will stop users from specifying a cluster label that they don't own when creating their `ApplicationSet` objects.

Now that we've worked out how both OpenUnison and ArgoCD are going to securely communicate with remote clusters in our platform, we next need to work out how our clusters will securely pull images from our image repository.

## Securely pushing and pulling images

In addition to having to securely call the APIs of remote clusters, we need to be able to securely push and pull images from our image registry. It would be great to use the same technique to work with our registry as we do with other APIs, but unfortunately, we won't be able to. Kubernetes doesn't provide a dynamic way for image pull secrets to be generated, meaning that we'll need to generate a static token. The token will need to be stored as a `Secret` in our API server too. Since we're going to be using Vault anyway, we plan on using the External Secrets Operator in each cluster so we can synchronize the pull secret from Vault.

We've worked through our technology stack, and how the various components are going to communicate. Next, we'll work through how we're going to deploy all this technology.

## Using Infrastructure as Code for deployment

Throughout this book, we've used bash scripts to deploy all our labs. We were able to do this because most of our labs were straightforward, with minimal integration, and didn't require any repeatability. This is generally not the case when working in enterprises. You'll want to have multiple environments for development and testing. You may need to deploy to multiple or different clouds. You might need to rebuild environments across international borders to maintain data sovereignty regulations. Finally, your deployment might just need more complex logic than what bash is able to easily provide.

This is where an **Infrastructure as Code (IaC)** tool begins to provide value. IaC tools are popular because they provide a layer of abstraction between your code and the APIs needed to deploy your infrastructure. For instance, an IaC tool can provide a common API for creating Kubernetes resources and creating resources in your cloud provider. They won't be exactly the same, but if you know how to use one then the patterns will generally apply to other providers.

There are two common approaches to IaC tools:

- **Imperative Scripting:** An IaC tool can be something that just makes it easier to re-run commands across multiple systems and in a re-usable way. It provides minimal abstractions and doesn't maintain any internal "state" between runs. Ansible is a great example of this kind of tool. It makes it easy to run commands against multiple hosts but doesn't handle "drift" from a known configuration.
- **State Reconciliation:** Many IaC tools store what the expected state of an environment is and reconcile against this state. This is very similar to the idea of GitOps, where the state is stored in a Git repository. The big benefit to this approach is that you can keep your infrastructure aligned with an expected state, so if your infrastructure were to "drift," your IaC tool knows how to bring it back. One of the challenges of this approach is that you now have state you need to manage and maintain.

There is no "correct" approach here; it really depends on what you're trying to accomplish. There are numerous open source IaC tools. For our platform, we're going to use Pulumi (<https://www.pulumi.com/>). One of the reasons I like Pulumi is that it doesn't have its own domain-specific language or markup – it provides APIs for Python, Java, Go, JavaScript, etc. So, while you still need an additional binary to run it, it makes for an easier learning curve, and I think easier long-term maintenance.

As far as managing state goes, Pulumi offers its own cloud for free, or you can use an object storage system like Amazon S3, or your local file system. Since we don't want you to have to sign up for anything, we're going to use the local file system for all our examples.

When working with Pulumi programs, one of the key points to understand is that you're not working on the infrastructure itself, you're working on the state you want to create, and then Pulumi reconciles the state your program creates with the reality of your existing infrastructure. To make this work, Pulumi runs two passes of your program. First, a pass to generate an expected state, and then again to apply the unknowns of that state. As an example, let's say you're going to deploy OpenUnison and the Kubernetes Dashboard using Pulumi. Part of OpenUnison's Helm chart requires knowing the name of the Service that exposes the dashboard's deployment. Pulumi controls the names of resources by default, so you won't know the name of the Service when writing your code, but it's provided to you via a variable. That variable isn't available in the first pass, but it is in the second pass of your code. Here's the Python code for deploying the dashboard via Pulumi:

```
k8s_db_release = k8s.helm.v3.Release(
    'kubernetes-dashboard',
    k8s.helm.v3.ReleaseArgs(
        chart=chart_name,
        version=chart_version,
        namespace='kubernetes-dashboard',
        skip_await=False,
        repository_opts= k8s.helm.v3.RepositoryOptsArgs(
```

```
        repo=chart_url
    ),
),
opts=pulumi.ResourceOptions(
    provider = k8s_provider,
    depends_on=[dashboard_namespace],
    custom_timeouts=pulumi.CustomTimeouts(
        create="8m",
        update="10m",
        delete="10m"
    )
)
)
```

The important part of this code is that once the chart is deployed, it's also made available to other parts of the code. Next, when we go to create OpenUnison's Helm chart values, we need to get the Service name:

```
openunison_helm_values["dashboard"]["service_name"] = k8s_db_release.name.
apply(lambda name: name)
```

Here, we're not getting the name directly from the release as a variable because, depending on which phase of the deployment you're in, your program won't know. So instead, you use a lambda in Python to inject a function that will return the value so that Pulumi can generate it at the right point. This was a big mental block for me when I first started working with Pulumi, so I wanted to make sure to point it out here.

We're not going to dive into more Pulumi implementation details here. There are several books on Pulumi and they have great documentation on their website for all of the languages they support. We wanted to focus on a brief introduction and some key concepts to set the stage. We'll walk through the deployment of our platform, including how to store and retrieve configuration information, in the next chapter as we deploy our platform.

We've covered the infrastructure for our platform, how that infrastructure interconnects, and how we plan to deploy it. Next, we'll turn our attention to how our tenants will be deployed.

## Automating tenant onboarding

Earlier, in the vCluster chapter, we deployed the OpenUnison NaaS portal to provide a self-service way for users to request tenants and have them deployed. This portal lets users request new namespaces to be created and allows developers to request access to these namespaces via a self-service interface. We built on this capability to include the creation of a vCluster in our namespace in addition to the appropriate RoleBinding objects. While that implementation was a good start, it ran everything on a single cluster and only integrated with the components that were needed to run a vCluster.

What we want to do is build a workflow that integrates our platform and creates all the objects we need to fulfill our requirements across all of our projects. The goal is that we'll be able to deploy a new application into our environment without having to run the `kubectl` command (or at least minimize its use).

This will require careful planning. Here's how our developer workflow will run:

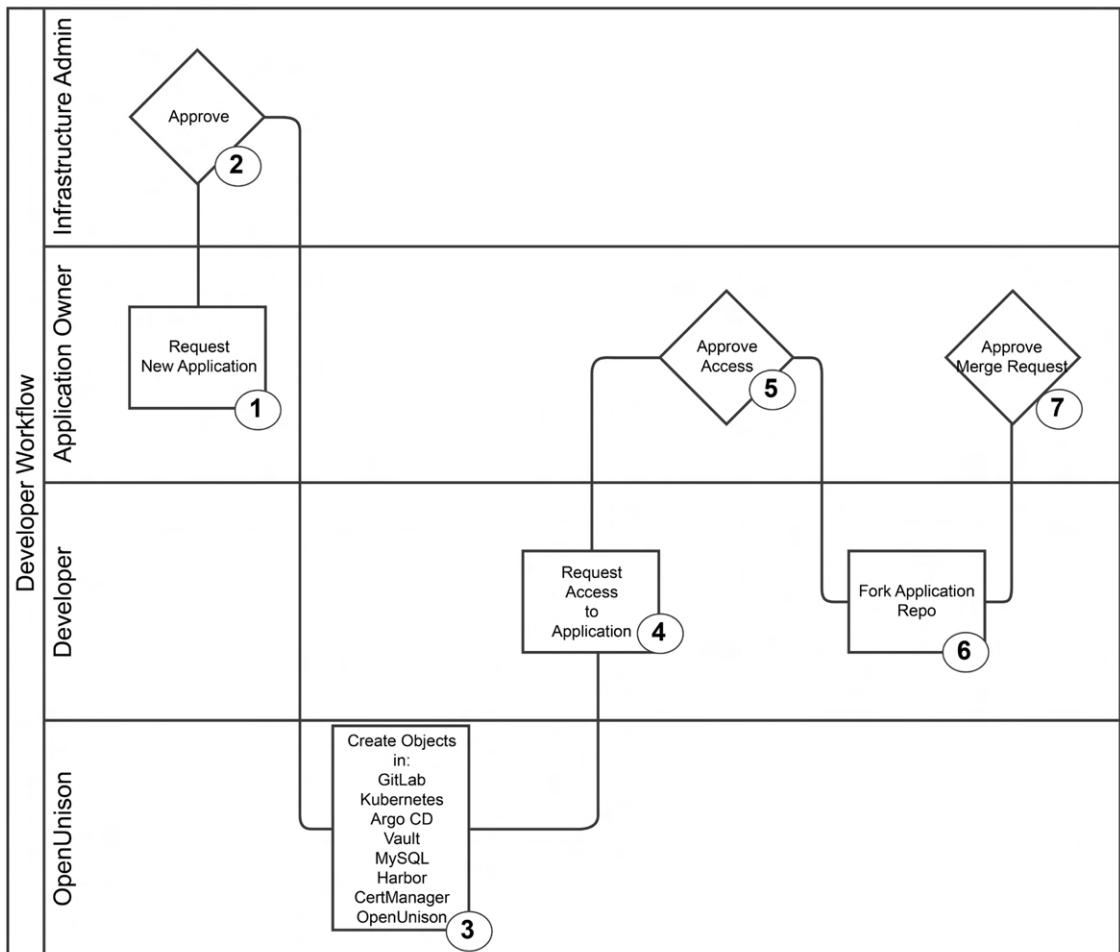


Figure 18.8: Platform developer workflow

Let's quickly run through the workflow that we see in the preceding figure:

1. An application owner will request an application be created.
2. The infrastructure admin approves the creation.
3. At this point, OpenUnison will deploy the objects we created manually. We'll detail those objects shortly.
4. Once created, a developer is able to request access to the application.
5. The application owner(s) approves access to the application.
6. Once approved, the developer will fork the application source base and do their work. They can launch the application in their developer workspace. They can also fork the build project to create a pipeline and the development environment operations project to create manifests for the application.
7. Once the work is done and tested locally, the developer will push the code into their own fork and then request a merge request.
8. The application owner will approve the request and merge the code from GitLab.

Once the code is merged, ArgoCD will synchronize the operations projects. GitLab will kick off a pipeline that will build our container and update the development operations project with the tag for the latest container. ArgoCD will synchronize the updated manifest into our application's development namespace. Once testing is completed, the application owner submits a merge request from the development operations workspace to the production operations workspace, triggering ArgoCD to launch into production.

Nowhere in this flow is there a step called "operations staff uses `kubectl` to create a namespace." This is a simple flow and won't totally prevent your operations staff using `kubectl`, but it should be a good starting point. All this automation requires an extensive set of objects to be created:

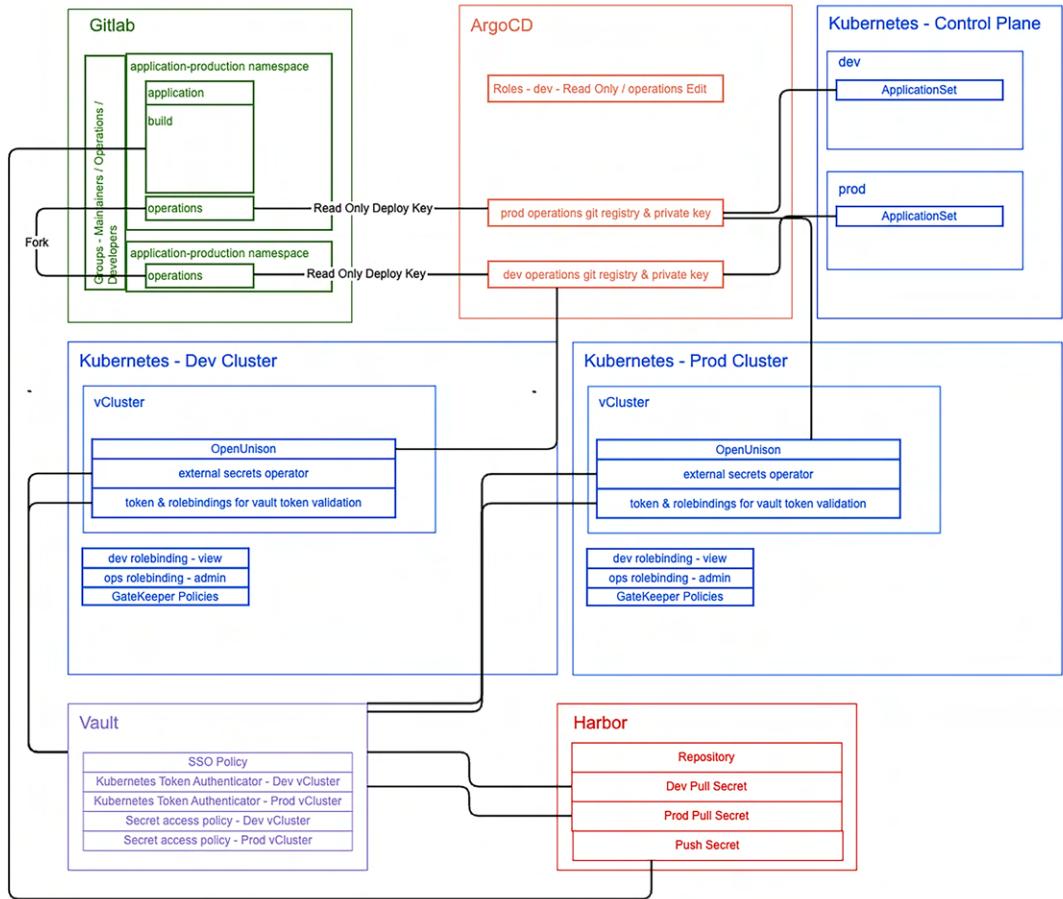


Figure 18.9: Application onboarding object map

The above diagram shows the objects that need to be created in our environment and the relationships between them. With so many moving parts, it's important to automate the process. Creating these objects manually is both time-consuming and error-prone. We'll work through that automation in the next chapter.

In GitLab, we create a project for our application code and operations. We also fork the operations project as a development operations project. For each project, we generate deploy keys and register webhooks. We also create groups to match the roles we defined earlier in this chapter. Since we’re using GitLab to build our image, we’ll need to register a key so that it can push images into Harbor as well.

For Kubernetes, we create namespaces for the development and production environments. We define vClusters in the tenant namespaces, backed by each cluster’s MySQL database. Next, we deploy an OpenUnison to each vCluster, using our control plane OpenUnison as the identity provider. This will enable our control plane OpenUnison to generate identities for each vCluster and allow Argo CD to manage them without having to use static keys.

Once OpenUnison is deployed, we need to add our vClusters to Vault for secrets management. We’ll also create namespace and ApplicationSets in the control plane cluster to configure Argo CD to generate Application objects for our tenant clusters. Since Argo CD doesn’t have any controls to make sure that an ApplicationSet only uses specific clusters, we’ll need to add a GateKeeper policy to make sure that users don’t attempt to create ApplicationSets for other tenants.

We also need to provision resources and credentials in Harbor so that our tenants can manage their containers. Next, we’ll onboard each vCluster into Vault and add external secret operator deployments for each vCluster. We’ll then provision the pull secrets into each cluster via Vault.

Finally, we add RBAC rules to ArgoCD so that our developers can view their application synchronization status but owners and operations can make updates and changes.

If that seems like quite a bit of work, you’re right! Imagine if we had to do all the work manually. Thankfully, we don’t. Before we get into our final chapter and start our deployment, let’s talk about what GitOps is and how we’ll use it.

## Designing a GitOps strategy

We have outlined the steps we want for our developer workflow and how we’ll build those objects. Before we get into talking about implementation, let’s work through how Argo CD, OpenUnison, and Kubernetes will interact with each other.

So far, we’ve deployed everything manually in our cluster by running `kubectl` commands off of manifests that we put in this book’s Git repo. That’s not really the ideal way to do this. What if you needed to rebuild your cluster? Instead of manually recreating everything, wouldn’t it be better to just let Argo CD deploy everything from Git? The more you can keep in Git, the better.

That said, how will OpenUnison communicate with the API server when it performs all this automation for us? The “easiest” way is for OpenUnison to just call the API server.

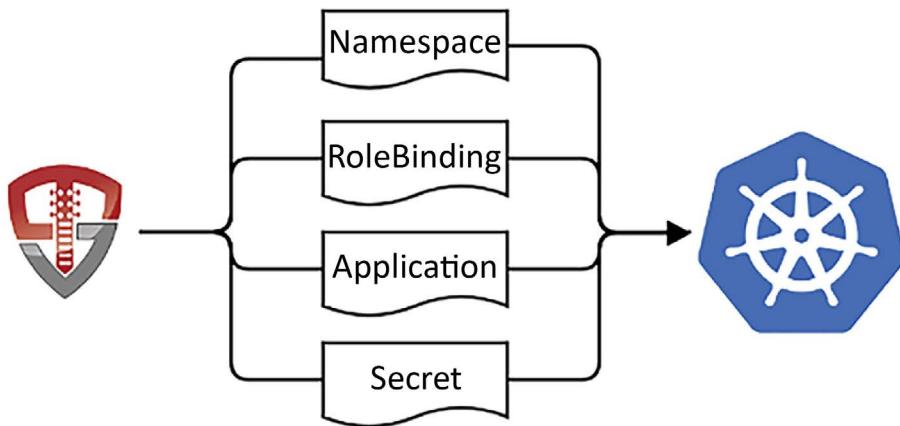


Figure 18.10: Writing objects directly to the API server

This will work. We'll get to our end goal of a developer workflow using GitOps, but what about our cluster management workflow? We want to get as many of the benefits from GitOps as cluster operators as our developers do! To that end, a better strategy would be to write our objects to a Git repository. That way, when OpenUnison creates these objects, they're tracked in Git, and if changes need to be made outside of OpenUnison, those changes are tracked too.

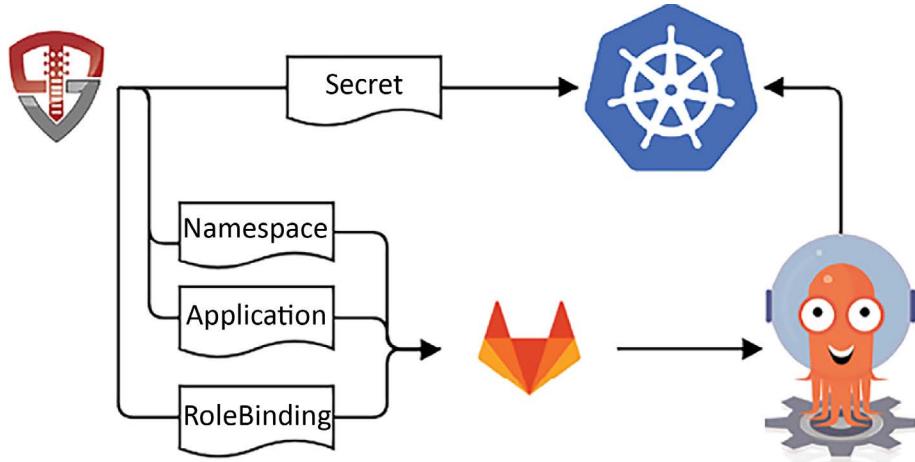


Figure 18.11: Writing objects to Git

When OpenUnison needs to create objects in Kubernetes, instead of writing them directly to the API server, it will write them into a management project in GitLab. Argo CD will synchronize these manifests into the API server.

This is where we'll write any objects we don't want our users to have access to. This would include cluster-level objects, such as Namespaces, but also namespace objects we don't want our users to have write access to, such as RoleBindings. This way, we can separate operations object management from application object management.

Here's an important security question to answer: if Argo CD is writing these objects for us, what's stopping a developer from checking a RoleBinding or a ResourceQuota into their repo and letting Argo CD synchronize it into the API server? At the time of publication, the only way to limit this is to tell Argo CD which objects can be synchronized in the AppProject object. This isn't quite as useful as relying on RBAC. We'll be able to work around this limitation by using a vCluster for tenant isolation. Yes, Argo CD will have access to the tenant's remote cluster as cluster-admin, but the user won't be able to check in ApplicationSets that talk to other clusters.

Finally, look at *Figure 18.11* and you'll notice that we're still writing Secret objects to the API server. We don't want to write secret information to Git. It doesn't matter if the data is encrypted or not; either way, you're asking for trouble. Git is specifically designed to make it easier to share code in a decentralized way, whereas your secret data should be tracked carefully by a centralized repository. These are two opposing requirements.

As an example of how easy it is to lose track of sensitive data, let's say you have a repo with Secrets in it on your workstation. A simple `git archive HEAD` will remove all Git metadata and give you clean files that can no longer be tracked. How easy is it to accidentally push a repo to a public repository by accident? It's just too easy to lose track of the code base.

Another example of why Git is a bad place to store secret information is that Git doesn't have any built-in authentication. When you use SSH or HTTPS when accessing a Git repo, either GitHub or GitLab is authenticating you, but Git itself has no form of built-in authentication. If you have followed the exercises in this chapter, go look at your Git commits. Do they say "root" or do they have your name? Git just takes the data from your Git configuration. There's nothing that ties that data to you. Is that an audit trail that will work for you as regards your organization's secret data? Probably not.

Some projects attempt to fix this by encrypting sensitive data in the repo. That way, even if the repo were leaked, you would still need the keys to decrypt the data. Where's the Secret for the encryption being stored? Is it in use by developers? Is there special tooling that's required? There are several places where it could go wrong. It's better to not use Git at all for sensitive data, such as Secrets.

In a production environment, you want to externalize your Secrets though, just like your other manifests. We will write our secret data into HashiCorp's Vault and let the clusters determine how they want to extract that information, either using the external secrets operator or a Vault sidecar.

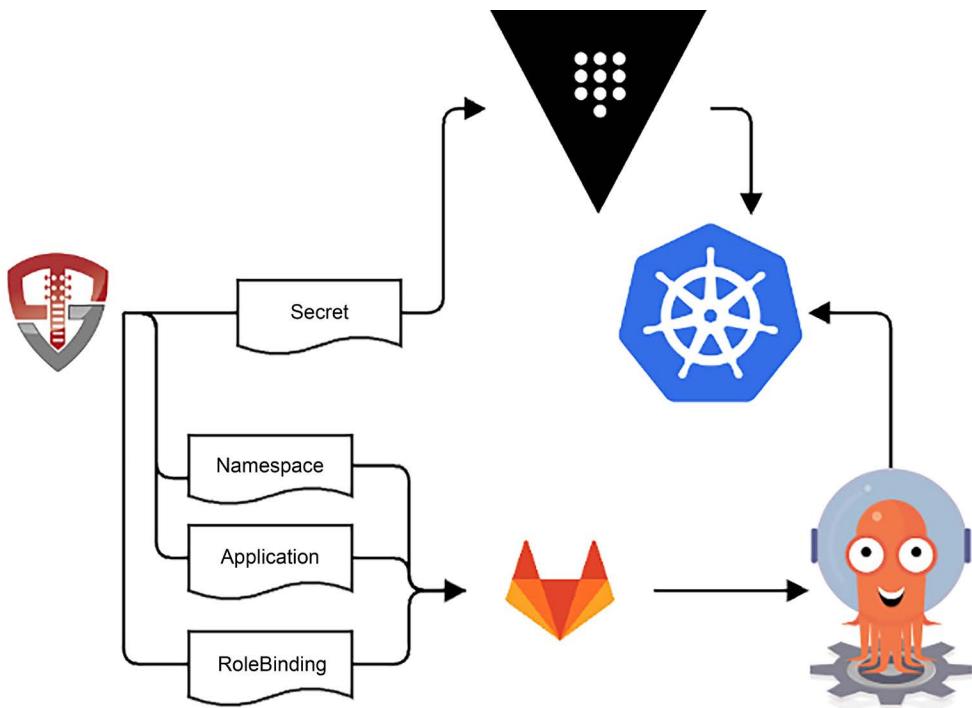


Figure 18.12: Writing secrets to Vault

With our developer workflow designed and example projects ready to go, next, we'll update OpenUnion, GitLab, and ArgoCD to get all this automation to work!

## Considerations for building an Internal Developer Platform

When developing an **Internal Developer Platform**, or IDP, it's important to keep some things in mind to avoid common antipatterns. When infrastructure teams build application support platforms, it's common to want to build in as much as possible to minimize the amount of work application teams need to do in order to run their application.

For instance, you can take this to an extreme where you simply provide a place for your code to go, and automate the rest. Most of the things we have identified in this chapter can be accomplished via boilerplate templates, right? Why bother even exposing it to the developers? Just let them check in their code and we'll do the rest! This is often referred to as "Serverless" or "Function as a Service." When appropriate, it's great because your developers don't need to know much about infrastructure.

The key phrase in the above paragraph was “when appropriate.” Throughout this book, we’ve stressed not only the impacts technology has on Kubernetes, but also the silos that are built in enterprises. While in DevOps, we often refer to “knocking silos down,” in enterprises, those silos are a result of management structures. As we’ve discussed throughout this book in different situations, hiding a deployment under so many layers of abstraction can lead to conflicts with those silos and put your team in a position to become a bottleneck.

When the infrastructure team becomes the bottleneck, often because they’ve tried to assume too much responsibility in an application rollout, there’s often a backlash that leads to a swing to the other extreme, where developers are given an empty “cloud” to deploy their own infrastructure. This isn’t helpful either as it leads to a tremendous amount of duplicated effort and expertise across teams.

In addition to keeping teams from their infrastructure, too much abstraction can lose the ability of your application teams to implement the logic they need to. There’s no scenario where an application infrastructure team can anticipate every edge case application teams need and lead to your application teams looking for alternatives for implementation. This is often where the idea of “Shadow IT” comes from. Application developers had requirements that infrastructure teams couldn’t fulfill, so they turned to cloud-based options.

Brian Gracely, a Senior Director at Red Hat and co-host of the Cloud Cast podcast, often says (paraphrasing) “Guard rails, not tracks.” This means it’s important for infrastructure teams to provide guardrails to best maintain a common infrastructure, without being so restrictive that application teams aren’t impeding the work that the application teams need to do.

When designing your internal developer platform, avoid going to too much of an extreme. If you want to offer low infrastructure offerings like Serverless/Function as a Service, that’s great. Just make sure that it’s an option, not the only approach.

We’ve covered quite a bit of theory in this chapter. In our next chapter, we’ll put this theory to the test and build out our platform!

## Summary

The previous chapters in this book focused on building out manifests and infrastructure. While we began looking at applications with Istio, we didn’t touch on how to manage their rollout. In this chapter, we moved our focus from building infrastructure to rolling out applications using pipelines, GitOps, and automation. We learned what components are built into a pipeline to move an application from code to deployment. We dove into what GitOps is and how it works. Finally, we designed a self-service multi-tenant platform that we’re going to implement in our final chapter!

Using the information in this chapter should give you some direction as to how you want to build your own platform. Using the practical examples in this chapter will help you map the requirements of your organization to the technology needed to automate your infrastructure. The platform we built in this chapter is far from complete. It should give you a map for planning your own platform that matches your needs.

## Questions

1. True or false: A pipeline must be implemented to make Kubernetes work.
  - a. True
  - b. False
2. What are the minimum steps of a pipeline?
  - a. Build, scan, test, and deploy
  - b. Build and deploy
  - c. Scan, test, deploy, and build
  - d. None of the above
3. What is GitOps?
  - a. Running GitLab on Kubernetes
  - b. Using Git as an authoritative source for operations configuration
  - c. A silly marketing term
  - d. A product from a new start-up
4. What is the standard for writing pipelines?
  - a. All pipelines should be written in YAML.
  - b. There are no standards; every project and vendor has its own implementation.
  - c. JSON combined with Go.
  - d. Rust.
5. How do you deploy a new instance of a container in a GitOps model?
  - a. Use `kubectl` to update the Deployment or StatefulSet in the namespace.
  - b. Update the Deployment or StatefulSet manifest in Git, letting the GitOps controller update the objects in Kubernetes.
  - c. Submit a ticket that someone in operations needs to act on.
  - d. None of the above.
6. True or false: All objects in GitOps need to be stored in your Git repository.
  - a. True
  - b. False
7. True or false: You can automate processes any way you want.
  - a. True
  - b. False

## Answers

1. a – True. It's not a requirement, but it certainly makes life easier!
2. d – There's no minimum number of steps. How you implement a pipeline will depend on your requirements.
3. b – Instead of interacting with the Kubernetes API, you store your objects in a Git repository, letting a controller keep them in sync.
4. b – Each pipeline tool has its own approach.
5. b – Your manifests in Git are your source of truth!
6. b – In the operators model, an operator will create objects based on your checked-in manifest. They should be ignored by your GitOps tools based on annotations or labels.
7. a – Kubernetes is a platform for building platforms. Build it how you need to!

# 19

## Building a Developer Portal

One of the more popular concepts in recent years of DevOps and automation is to provide an **Internal Developer Portal (IDP)**. The purpose of this portal is to provide a single point of service for your developers and infrastructure team to be able to access architectural services without having to send an email to “your guy” in IT. This is often the promise of cloud-based services, though it requires considerable custom development to achieve. It also provides the foundation for creating the guardrails needed to develop a manageable architecture.

This chapter is going to combine the theory we walked through in *Chapter 18, Provisioning a Multitenant Platform*, along with most of the concepts and technologies we’ve learned throughout this book to create an IDP. Once you’ve completed this chapter, you’ll have an idea of how to build an IDP for your infrastructure, as well as context for how the various technologies we have built and integrated into Kubernetes through this book should come together.

This chapter will cover the following topics:

- Technical requirements
- Deploying our IDP
- Onboarding a tenant
- Deploying an application
- Expanding our platform

Finally, before we dive into the chapter, we’d like to say *thank you!* This book has been quite the journey for us. It’s amazing to see how much has changed in our industry since we wrote the second edition, and how far we have come. Thank you for joining us on our quest to build out enterprise Kubernetes, and explore the different technologies and how the enterprise world impacts how we create that technology.

## Technical Requirements

This chapter has more significant technology requirements than the previous chapters. You'll need three Kubernetes clusters with the following requirements:

- **Compute:** 16 GB memory and 8 cores. You're going to be running GitLab, Vault, OpenUnison, Argo CD, etc. It's going to require some real horsepower.
- **Access:** Make sure you can update and access the local nodes. You'll need this to add our CA certificate to your nodes so that it can be trusted when pulling container images.
- **Networking:** You won't need public IPs, but you will need to be able to access all three clusters from your workstation. It will make the implementation easier if you use load balancers for each, but it's certainly not a requirement.
- **Pulumi and Python 3:** We're going to be using Pulumi to deploy our platform, running on Python 3. The workstation you use will need to be able to run these tools. We built and wrote this chapter on macOS, using Homebrew (<https://brew.sh/>) for Python.

Before we start building, we're going to spend some time on the technical requirements for this chapter and why they are requirements, relating them back to common enterprise scenarios. First, let's look at our compute requirements.

## Fulfilling Compute Requirements

Throughout the rest of this book, our goal was to run all the labs on a single VM. We did this for a few reasons:

- **Cost:** We know how quickly costs can climb when learning technology and we wanted to make sure we weren't asking you to spend more money for this.
- **Simplicity:** Kubernetes is hard enough without getting into the details of how compute and networking are set up in enterprise environments! We also didn't want to have to worry about storage, which brings several complications to the table.
- **Ease of Implementation and Support:** We wanted to make sure we could help with the labs, so limiting how you deployed them made that much easier for us.

With that all said, this chapter is different. You could use three VMs running KinD, but that would probably start causing more problems than it would be worth. There are two primary options we're going to cover: using the cloud and building a home lab.

## Using Cloud-Managed Kubernetes

It's very popular to use a managed Kubernetes when there's nothing to deploy. Every major cloud has its own, and so do most smaller clouds. These are great if you are OK with spending some money and are more focused on Kubernetes than the infrastructure that runs it. Make sure, though, that when you set up your clusters, you are able to directly access your worker nodes via SSH or some other means.

Many cloud-based managed clusters make it the default that you can't access your nodes, which, from a security standpoint, is great! You can't breach something you can't access! The downside is that you can't customize it either. We'll cover this in the next section, but most enterprises require customized nodes at some level, even when using cloud-managed Kubernetes.

Also, make sure you're able to handle the costs. A setup with three clusters is not likely to stay within whatever free credits you get for long. You'll want to make sure you can afford to spend the money. That said, if throwing money at a cloud isn't for you, then maybe a home lab will be your best bet.

## Building a Home Lab

The cloud can get really expensive, and that money is just thrown away. You don't have anything to show for it! The alternative is building a home lab to run your clusters. As of the time of writing this book, it's never been easier to run enterprise-grade infrastructure in your own home or apartment. It doesn't require a massive investment and can be far cheaper than a single cloud-managed cluster for just a month or two.

You can start very simply with a single refurbished or even home-built server for under \$500 on eBay and other auction sites. Once you have a server, install Linux and a hypervisor and you're ready to start. This, of course, requires more time being spent on underlying infrastructure than we've done so far, but can be very rewarding both from a personal level and an economic one. If you're going through this book in the hopes of breaking into the enterprise Kubernetes world, knowing about how your infrastructure works can be a real differentiator among other candidates.

If you're in the position to spend a few more dollars on your home lab, there are projects that make it easier to build out your lab. Two such projects are:

- **Metal as a Service (MaaS):** This project (<https://maas.io/>) from Canonical makes it easier to quickly onboard infrastructure to a lab by providing resource management, DNS, network boot, etc. Canonical is the same company that created the Ubuntu Linux distribution. While it started as a project for onboarding hardware quickly, it also supports KVM via the `virsh` protocol, which allows for the management of VMs via SSH. This is what I run my home lab on right now on a few home-built PCs running Ubuntu.
- **Container Craft Kargo:** A relatively new platform (<https://github.com/ContainerCraft/Kargo>) that combines several "enterprise" quality systems to build a home lab built on Kubernetes. The great thing about this project is it starts with Talos, a combination operating system and Kubernetes distribution, and uses KubeVirt to leverage the Kubernetes API for deploying VMs. It's a great project that I've started working on and using and will be moving my home lab, too.

Having worked through what to build a home lab on and where you can deploy your IDP, we'll next explore why you'll need to have direct access to your nodes.

## Customizing Nodes

When working with a managed Kubernetes such as Amazon or Azure, the nodes are provided for you. There's also an option to disable external access. This is great from a security standpoint because you don't need to secure what you can't access!



As we've said before, there's a difference between security and compliance. While a fully managed node may be more secure, your compliance rules may say that you, as the cluster manager, must have processes in place to manage node access. Simply removing all access may not cover this compliance issue.

There's a functional drawback to this approach though; you're now not able to customize the nodes. This drawback can manifest in several ways:

- **Custom Certificates:** We've made the point multiple times throughout this book that enterprises often maintain internal Certificate Authorities (CAs). We're mirroring this process by using our own internal CA for issuing certificates used by Ingresses and other components. This includes our Harbor instance, which means for our cluster to be able to pull images, the node it runs on must trust our CA. For this work, the node must be configured to trust our CA. There's no API for Kubernetes to trust a private CA, unfortunately.
- **Drivers:** While not as important with cloud-managed Kubernetes, it's not unusual for enterprises to use specific hardware stacks that are certified to work with specific hardware. For instance, your Storage Area Network (SAN) may have specific kernel drivers. If you don't have access to your nodes, you can't install these drivers.
- **Supported Operating Systems:** Many enterprises, especially those in highly regulated industries, want to make sure they're running a supported operating system and configuration. For instance, if you're running Azure Kubernetes but your enterprise has standardized on Red Hat Enterprise Linux (RHEL), you'll need to create a custom node image.

While requiring access to your nodes complicates your deployment in multiple ways, such as requiring a way to manage and secure that access, it's often a necessary evil to the deployment and management of Kubernetes.

While you may be most familiar with building nodes on Ubuntu, RHEL, or RHEL clones, Talos Linux from Sidero (<https://www.talos.dev/>) provides a novel approach by stripping down the OS to the bare minimum needed to start Kubernetes. This means that all interaction with your OS happens via an API, either from Kubernetes or Talos. This makes for some very interesting management because you no longer need to worry about patching the OS; the upgrades are all done via APIs. No more securing SSH, but you do still need to lock down the API. Not having access to the OS means you can't just deploy a driver either. The Kargo project we mentioned earlier uses Talos for its OS. When I wanted to integrate my Synology Network Attached Storage (NAS), I had to create a DaemonSet for the task to make it work with iSCSI (<https://github.com/ContainerCraft/Kargo/blob/main/ISCSI.md>).

Since we're using our own internal CA, your nodes will need to be customizable at least to the point of being able to include custom certificates.



You may think that an easy way to avoid this situation is to use Let's Encrypt (<https://letsencrypt.org/>) to generate certificates, avoiding the need for custom certificate authorities. The issue with this approach is that it avoids the common need to use custom certificates in enterprises, whereas Let's Encrypt doesn't provide a standard way of issuing internal certificates. Its automatic issuance APIs are built on public validation techniques such as having publicly available URLs or via DNS. Neither is acceptable to most enterprises so Let's Encrypt is generally not allowed for internal systems. Since Let's Encrypt isn't generally used for enterprises' internal systems, we won't use it here.

Now that we know why we need access to our nodes, next, we'll talk about network management.

## Accessing Services on Your Nodes

Throughout this book, we've assumed everything runs on a single VM. Even when we ran multiple nodes in KinD, we did tricks with port forwarding to get access to containers running on those nodes. Since these clusters are larger, you may need a different approach. We covered MetalLB in *Chapter 4, Services, Load Balancing, and Network Policies*, as a load balancer, which is potentially a great option for multiple node clusters. You can also deploy your Ingress as a DaemonSet, with the pods using host ports to listen across all your nodes, then use DNS to resolve all your nodes.

Regardless of which approach you use, we're going to assume that all services will be accessed via your Ingress controller. This includes text or binary protocols:

- Control Plane:
  - 80/443: http/https
  - 22: ssh
  - 3306: MySQL
- Dev Node:
  - 80/443: http/https
  - 3306: MySQL
- Production Node:
  - 80/443: http/https
  - 3306: MySQL

When we begin our rollout, we'll see that NGINX can be used to forward both web protocols and binary protocols, allowing you to use one LoadBalancer per cluster. It's important to note that your control plane cluster will need to be able to access HTTPS and MySQL on both your dev and production nodes. Also note that port 22 will be needed by our control plane cluster, so if you plan on supporting SSH directly for your nodes, you'll need to configure it for another port if you're not using an external LoadBalancer like MetalLB.

We know how we're going to run our clusters, how we'll customize the worker nodes, and how we'll access their services. Our last step is to get Pulumi ready.

## Deploying Pulumi

In the last chapter, we introduced the concept of **Infrastructure as Code (IaC)** and said that we would be using Pulumi's IaC tooling for deploying our IDP. In order to use Pulumi, you'll need a workstation to host and run the client.

Before we dive too deeply into how to deploy Pulumi, it's important to understand some key concepts relating to IaC that are often glossed over. All IaC tools are made up of at least three components:

- **Controller:** The controller is generally a workstation or service that runs the IaC tooling. For our book, we're assuming a workstation will run our Pulumi programs. For larger-scale or production implementations, it's generally better to deploy a controller service that runs the IaC tooling on your behalf. With Pulumi, this could be their own SaaS service or a Kubernetes operator.
- **Tooling:** This is the core component of IaC. It's the part that you create for building your infrastructure and is specific to each IaC tool.
- **Remote APIs:** Each IaC tool interacts with remote systems via an API. The original IaC tools interacted with Linux servers via SSH, and then with clouds via their own APIs. Today, IaC tools interact with individual systems using their own providers that wrap the target's APIs. One of the more difficult aspects of this is how to secure these APIs. We've spent much of this book stressing the importance of short-lived tokens, which can also be applied to our IaC implementation.

In addition to the above three components, many IaC tools include some kind of state management file. In the previous chapter, we described how IaC tools, like Pulumi and Terraform, generate an expected state based on your IaC tooling that then is applied to the downstream systems. This state will contain all the same privileged information as your infrastructure and should be treated as a "secret." For instance, if you were to provision a password for your database via IaC, your state file has a copy of that password.

For our deployment, we're going to use a local state file. Pulumi offers options for storing state on remote services like S3 buckets or using its own cloud offering. While any of these options are better for management than a local file, we didn't want you to have to sign up for anything to read this book and run the exercises, so we're using a local file for all Pulumi state management.



If you've been observant of IaC industry news, you may have seen that HashiCorp, the company that created Terraform (and Vault), changed the open source license to a "Business Source License" in the summer of 2023. This was to combat the large number of SaaS providers that were offering "Terraform as a Service" that weren't paying anything back to HashiCorp. This change in license led to many of these SaaS providers creating OpenTofu, a fork of Terraform under the original Apache 2 license. We're not making any judgments or recommendations on the situation, only to point out that managed services around state and controllers are where IaC companies make most of their revenue.

Since Pulumi is a commercial, albeit open source, package, you'll want to follow their instructions for getting the command-line Pulumi tools installed onto the workstation you want to run as your controller: <https://www.pulumi.com/docs/install/>.

Finally, you'll need the chapter's Git repository from GitHub. You can access the code for this chapter at the following GitHub repository: <https://github.com/PacktPublishing/Kubernetes-An-Enterprise-Guide-Third-Edition/tree/main/chapter19>.

Now that we've covered the environment and requirements for our IDP build, we can dive into the deployment of our IDP.

## Deploying our IDP

With our technical requirements out of the way, let's deploy our portal! First off, I assume that you have three running clusters. If you've got a `LoadBalancer` solution for each, then the next step is to deploy NGINX. We didn't include NGINX in the Pulumi tooling because, depending on how your clusters are deployed, this can change how you deploy NGINX. For example, I didn't use typical clusters with `LoadBalancer`; I just used single-node clusters and patched NGINX with host ports for 80 and 443.

We're also assuming you have some kind of native storage attached and have set a default `StorageClass`.

We're going to run NGINX assuming that it will be the Ingress for HTTP(S), MySQL, and SSH. This is pretty easy to do with the Helm chart for NGINX. On all three clusters, run the following:

```
helm upgrade --install ingress-nginx ingress-nginx \
--repo https://kubernetes.github.io/ingress-nginx \
--namespace ingress-nginx --create-namespace \
--set tcp.3306=mysql/mysql:3306
```

This will deploy NGINX as an Ingress controller and launch a `LoadBalancer`.



If you're using a single-node cluster, now would be the time to patch your Deployment with something like: `kubectl patch deployments ingress-nginx-controller -n ingress-nginx -p '{"spec":{"template":{"spec":{"containers":[{"name":"controller","ports":[{"containerPort":80,"hostPort":80,"protocol":"TCP"}, {"containerPort":443,"hostPort":443,"protocol":"TCP"}, {"containerPort":22,"hostPort":22,"protocol":"TCP"}, {"containerPort":3306,"hostPort":3306,"protocol":"TCP"}]}]}}}'`.

This will force the ports through the NGINX deployed on your cluster. The command is in `chapter19/scripts/patch-nginx.txt`.

Once NGINX is deployed, you'll need DNS wildcards for all three of your LoadBalancer IPs. It's tempting to just use IP addresses if you don't have access to DNS, but don't! IP addresses can be handled in odd ways with certificate management. If you don't have a domain name you can use, then use `nip.io` the way we have throughout this book. I'm using three domains, which I'll use in all examples:

- **Control Plane** – `*.idp-cp.tremolo.dev`
- **Development Cluster** – `*.idp-dev.tremolo.dev`
- **Production Cluster** – `*.idp-prod.tremolo.dev`

With our environment now ready for deployment, let's begin by creating a Pulumi virtual environment.

## Setting Up Pulumi

Before we can begin running our Pulumi program to start our rollout, we first need to create a Python virtual environment. Python dynamically links to libraries in ways that can create problems and conflict with other systems built on Python. To avoid these conflicts, which back in our Windows programming days was referred to as "DLL Hell," you need to create a virtual environment that will be isolated for just your Pulumi program. In a new directory, run the following:

```
$ python3 -m venv .
```

This creates a virtual environment that you can now use with Pulumi without interfering with other systems. Next, we'll need to "source" this environment so that our execution uses it:

```
$ . ./bin/activate
```

The last Python step is to download your dependencies, assuming you've checked out the latest Git repository for the book:

```
$ pip3 install -r /path/to/Kubernetes-An-Enterprise-Guide-Third-Edition/
chapter19/pulumi/requirements.txt
```

The `pip3` command reads all the packages named in `requirements.txt` and installs them into our virtual environment. At this point, Python is ready, and we need to initialize our stack.

The first step to getting Pulumi ready is to "log in" to store your state file. There are multiple options, from using Pulumi's cloud to S3 buckets to your localhost. You can see the various options on its website: <https://www.pulumi.com/docs/concepts/state/>. We're going to use our local directory:

```
$ pulumi login file://.
```

This creates a directory in your current directory called `./pulumi` that will contain your backend. Next, we need to initialize a Pulumi "stack" to track the state for your deployment:

```
$ cd chapter19
$ git archive --format=tar HEAD > /path/to/venv/chapter19/chapter19.tar
$ cd /path/to/venv/chapter19/
$ tar -xvf chapter19.tar
$ rm chapter19.tar
```

Next, edit `/path/to/venv/chapter19/pulumi/Pulumi.yaml`, changing `runtime.options.virtualenv` to point to `/path/to/venv`. Finally, we can initialize our stack:

```
$ cd /path/to/venv/chapter19/pulumi  
$ pulumi stack init bookv3-platform
```

You'll be asked to provide a password to encrypt your secrets. Make sure to write it down someplace secure! You'll now have a file called `/path/to/venv/chapter19/pulumi/Pulumi.bookv3-platform.yaml` that is used to track your state.

Our environment is now prepped and ready to go! Next, we'll configure our variables and start our rollout.

## Initial Deployment

*Eventual Consistency is a Lie – Ancient Cloud-Native Sith Proverb*

In the Kubernetes world, we often assume the idea of “eventual consistency,” where we create a control loop that waits for our expected conditions to become reality. This is generally an overly simplistic outlook on systems, especially when working with enterprise systems. All this is to say that even though almost all of our deployment is managed in a single Pulumi program, it will need to be run multiple times to get the environment fully deployed. As we walk through each step, we’ll explain why it needed to be run on its own.

With that out of the way, we need to configure our variables. We wanted to minimize the amount of configuration, so you’ll need to set the following:

Option	Description	Example
<code>openunison.cp.dns_suffix</code>	The DNS domain name of the control plane cluster	<code>idp-cp.tremolo.dev</code>
<code>kube.cp.context</code>	The Kubernetes context for the control plane in the control plane's kubectl configuration	<code>kubernetes-admin@kubernetes</code>
<code>harbor:url</code>	The URL for Harbor after deployment	<code>https://harbor.idp-cp.tremolo.dev</code>
<code>kube.cp.path</code>	The path to the kubectl configuration file for your control plane cluster	<code>/path/to/idp-cp</code>
<code>harbor:username</code>	The admin username for Harbor	<code>Always admin</code>
<code>openunison.dev.dns_suffix</code>	The DNS suffix for the development cluster	<code>idp-dev.tremolo.dev</code>

openunison.prod.dns_suffix	The DNS suffix for the production cluster	idp-prod.tremolo.dev
----------------------------	---	----------------------

Table 19.1: Configuration options in Kubernetes clusters

To make it easier, you can customize `chapter19/scripts/pulumi-initialize.sh` and run it. You can set each one of these options manually by running:

```
$ pulumi config set option value
```

where `option` is the option you want to set and `value` is its `value`. Finally, we can run the deployment:

```
$ cd /path/to/venv/chapter19/pulumi
$ pulumi up -y
```

You'll be asked to provide the password to decrypt your secrets. Once done, this initial deployment will take a while. Depending on the speed of your network connection and how powerful your control plane cluster is, it could take 10 to 15 minutes.

Once everything is deployed, you'll see a message like this:

```
.
.
.

Resources:
+ 53 created

Duration: 7m6s
```

If you look at the output of the command, you'll see all the resources that were created! This lines up with the design we put together in the previous chapter.



We're not going to walk through all of the code. There are over 45,000 lines! We'll cover the highlights after everything is deployed.

At this point, we have some gaps:

- **Vault:** Vault is deployed but hasn't been configured. We can't configure Vault until we've unsealed it.
- **GitLab:** The baseline of GitLab has been deployed, but we don't have a way to run workflows. We also need to generate an access token so OpenUnison can interact with it.
- **Harbor:** Harbor is running, but we can't complete SSO integration without having the Harbor admin password. We'll also need this password for integration with OpenUnison.

- **OpenUnison:** The baseline OpenUnison Namespace as a Service portal has been deployed, but we haven't deployed any of the additional configurations needed to power our IDP.

Next, let's get Vault unsealed and ready for deployment.

## Unsealing Vault

Remember how in *Chapter 8, Managing Secrets*, we had to “unseal” Vault by extracting randomly generated keys and running a script in the pod to unlock the running Vault. There's no easy way to do this in Pulumi, so we need to use a Bash script. We also want to be able to store the unsealed keys in a safe space because once you've retrieved them, you can't get them a second time. Thankfully, Pulumi's secret management makes it easy to store the keys in the same place as the rest of our configuration. First, let's unseal our Vault:

```
$ cd /path/to/venv/chapter19/pulumi  
$ export KUBECONFIG=/path/to/cp/kubectl.conf  
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword  
$ ./vault/unseal.sh
```

It's important to set either `PULUMI_CONFIG_PASSPHRASE` or `PULUMI_CONFIG_PASSPHRASE_FILE` before running `unseal.sh`. Once done, you'll see that there are two more secrets if you run `pulumi config`:

```
$ pulumi pulumi config --show-secrets  
. . .  
vault.key  hvs.I...  
vault.tokens {  
    "unseal_keys_b64": [  
        . . .  
    ]  
}
```

Now your configuration is stored in your Pulumi configuration. If you're using a centralized configuration, such as with Pulumi Cloud or S3 buckets, this would probably be much more useful! If you need to restart your pod for whatever reason, you can unseal it again by running:

```
$ cd /path/to/venv/chapter19/pulumi  
$ export KUBECONFIG=/path/to/cp/kubectl.conf  
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword  
$ ./vault/unseal_after_init.sh
```

With Vault ready to be configured, next, we'll get Harbor's configuration ready.

## Completing the Harbor Configuration

Configuring Harbor is actually very simple:

```
$ cd /path/to/venv/chapter19/pulumi  
$ export KUBECONFIG=/path/to/cp/kubectl.conf  
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword  
$ ../../scripts/harbor-get-root-password.sh
```

This script does two things:

- Gets the randomly generated password from the `harbor-admin` secret and stores it in the Pulumi configuration
- Sets a flag so our Pulumi program knows to finish the SSO configuration

We had to go through this step because the Harbor provider uses configuration options specific to the `harbor` namespace. This is different from our other configuration options. Let's consider code that looks like this in your Pulumi program:

```
my_config = pulumi.config("someconfig")
```

Your code isn't saying "Get the configuration called `someconfig`"; it's saying "Get the configuration called `someconfig` in my stack's namespace." The separation between namespaces means that our code can't get configuration information from another namespace. From a practical standpoint, this means we're defining the same information multiple times between `harbor:url` and `harbor:password`, as well as `harbor:username`.

This approach seems inefficient and error-prone in our scenario, but at scale, it makes for a great way to secure separate silos. In many deployments, the people who own Harbor aren't the same people who might own the automation. By not allowing our code to have access to the `harbor` namespace, but being able to call libraries that depend on it, we're able to use this secret data without ever actually knowing it! Of course, since we're using a single secret set that we have access to, this security benefit is negated. However, if you're using a centrally managed service for your Pulumi controller, it allows developers to write code that never knows the secret data it relies upon.

Now that Harbor is ready for its final configuration, we need to run some manual steps in GitLab. We'll do that in the next section.

## Completing the GitLab Configuration

There are two key components we're missing from GitLab. First, we need to generate a token for OpenUnison to use when automating GitLab. The other is we need to manually configure a runner. Later on, we're going to use GitLab's integrated workflows to build a container and push it into Harbor. GitLab does this by launching a pod, which requires some automation. The service that launches these pods needs to be registered with GitLab. This makes sense because you might want to run services on a local Kubernetes cluster or a remote cloud. To handle this scenario, we need to tell GitLab to generate a runner and give us a registration token. First, we'll generate the runner registration token.

### Generating a GitLab Runner

The first step is to log in to GitLab. We haven't configured SSO yet, so you'll need to log in with the root credentials. These are stored as a Secret in the GitLab namespace that ends with `gitlab-initial-root-password`. Once you have the password, log in to GitLab with the username `root`. The URL will be `https://gitlab.controlplane.dns.suffix`, where `controlplane.dns.suffix` is the DNS suffix for your control plane cluster. For me, the URL is `https://gitlab.idp-cp.tremolo.dev/`.

Once logged in, click on **Admin Area** in the lower left-hand corner:

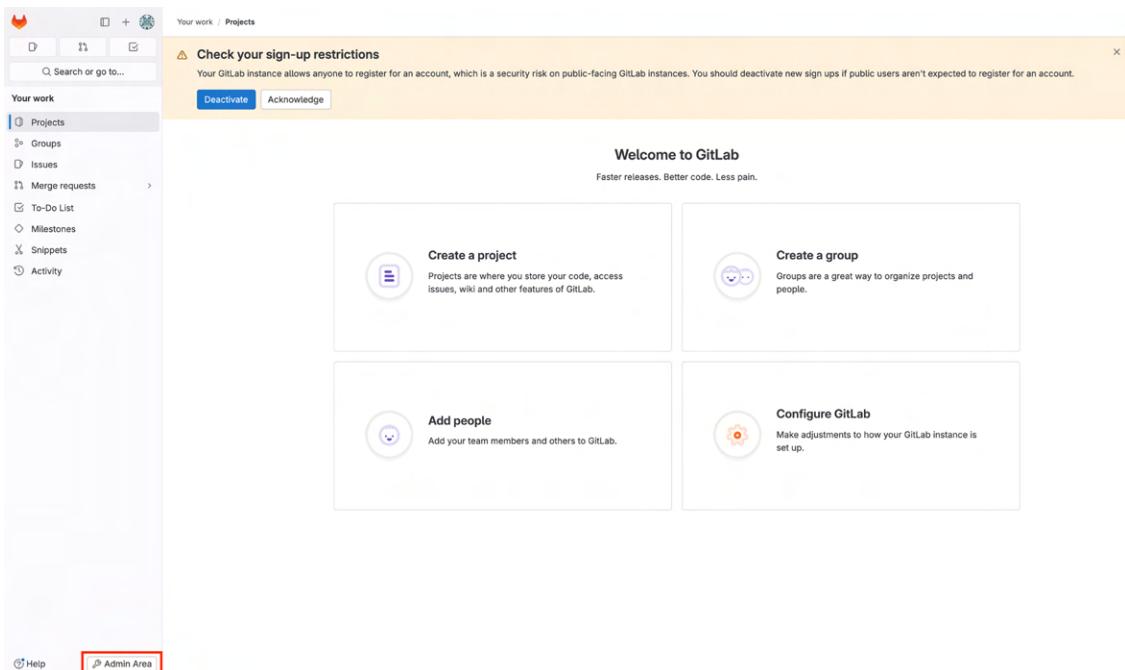


Figure 19.1: GitLab main screen

Next, expand CI/CD and click on Runners:

The screenshot shows the GitLab Admin Area interface. On the left, there is a sidebar with various administrative sections like Overview, Projects, Users, Groups, Topics, GitLab Servers, Analytics, Monitoring, Messages, System Hooks, Applications, Abuse Reports, Deploy Keys, Labels, and Settings. The 'CI/CD' section is expanded, and its sub-options 'Runners' and 'Jobs' are visible. A red box highlights the 'CI/CD' section. The main content area has a header 'Check your sign-up restrictions' with 'Deactivate' and 'Acknowledge' buttons. Below it is a 'Get security updates from GitLab and stay up to date' section with a 'Sign up for the GitLab newsletter' button. The 'Instance overview' section displays statistics: 0 projects, 1 user, and 0 groups. The 'Statistics' section shows metrics like Forks, Issues, Merge requests, Notes, Snippets, SSH Keys, Milestones, and Active Users. The 'Features' section lists various components with their status (e.g., Sign up, LDAP, Gravatar, OmnAuth, Reply by email, Container Registry, GitLab Pages, Instance Runners) and version numbers. The 'Components' section shows a list of components with their versions, including GitLab (v17.1.0), GitLab Shell (14.38.0), GitLab Workhorse (v17.1.0), GitLab API (v4), GitLab KAS (17.1.0), Ruby (3.1.5p253), Rails (7.0.8.4), PostgreSQL (main) (14.8), PostgreSQL (cl) (14.8), Redis (6.2.7), and GitLab Servers. At the bottom, there are buttons for 'Latest projects', 'Latest users', and 'Latest groups'.

Figure 19.2: GitLab Admin Area

Once the screen loads, click on New instance runner:

The screenshot shows the 'Runners' page within the GitLab Admin Area. The sidebar is identical to Figure 19.2, with 'Runners' selected. The main content area has a header 'Check your sign-up restrictions' with 'Deactivate' and 'Acknowledge' buttons. Below it is a 'Runners' section with tabs for All, Instance, Group, and Project. There is a search bar and a filter for 'Created date'. A large circular icon with a person icon is displayed. The 'Get started with runners' section contains the text: 'Runners are the agents that run your CI/CD jobs. Create a new runner to get started.' and 'Still using registration tokens?'. A red box highlights the 'New Instance runner' button at the top right of the page.

Figure 19.3: GitLab Runners

When the **New instance runner** screen loads, check **Run untagged jobs** since we're only running jobs on our own cluster. You can use these tags to manage running jobs across multiple platforms, similar to how you can use node tags to manage where to run workloads in Kubernetes. Next, click **Create runner**:

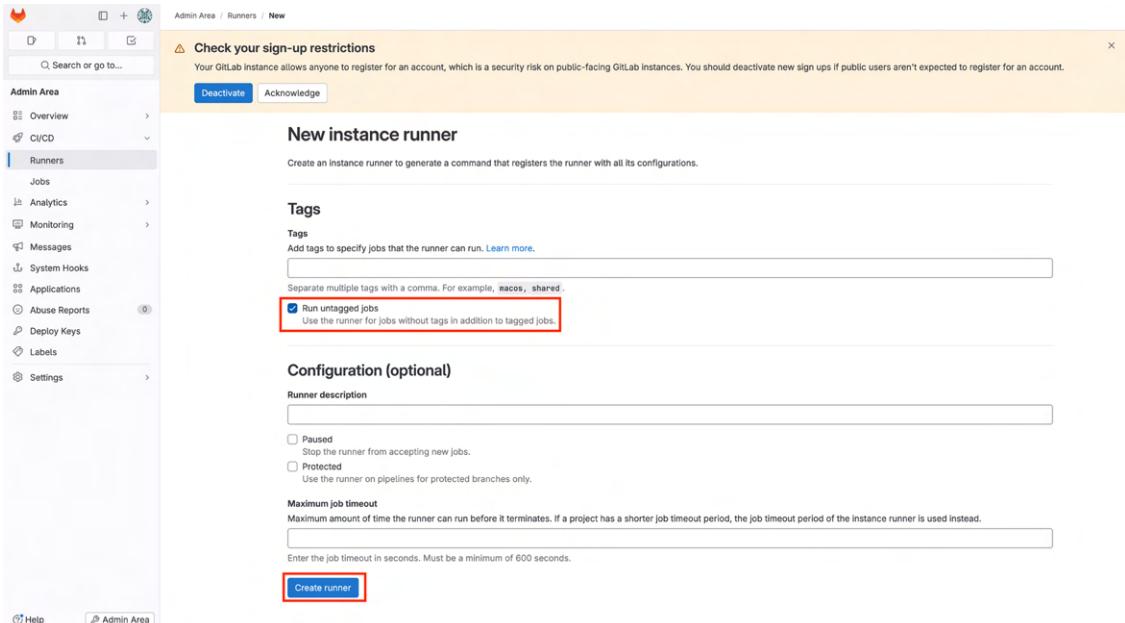


Figure 19.4: GitLab new instance runner

Finally, you'll have a token that we can configure in our Pulumi configuration:

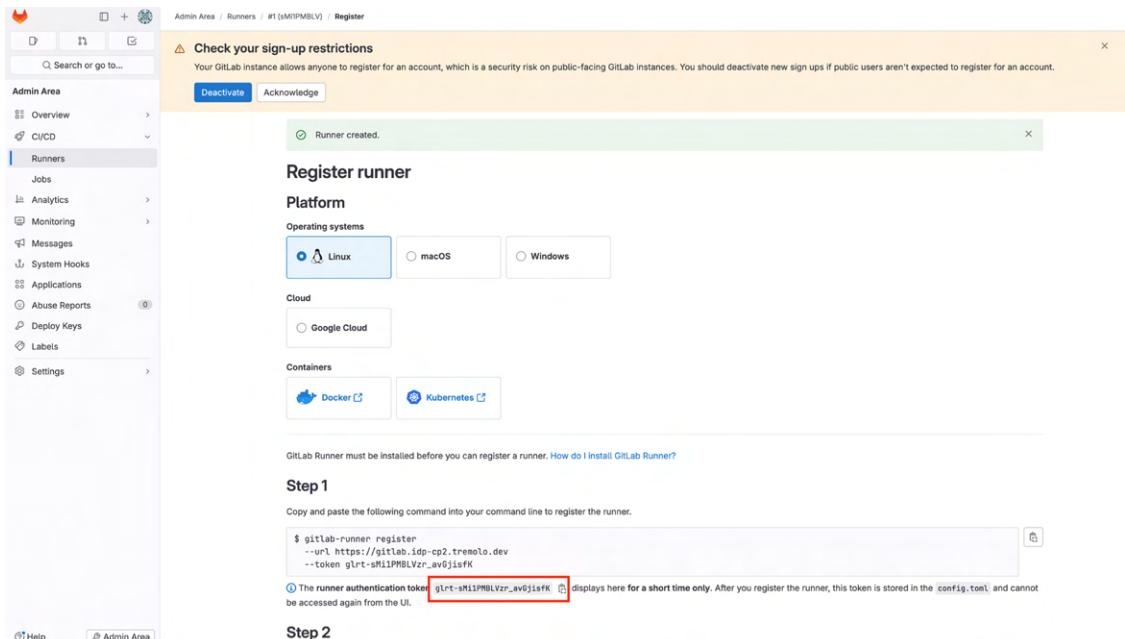


Figure 19.5: New runner token

Copy this token and configure it in Pulumi:

```
$ cd /path/to/venv/chapter19/pulumi  
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword  
$ pulumi config set gitlab.runner.token \  
'glrt-Y-fSdvy_6_xgXcyFW_PW' --secret
```

Next, we'll configure a token for automating GitLab.

## Generating a GitLab Personal Access Token

In the previous section, we configured a runner. Next, we need a token so that OpenUnison can automate provisioning into GitLab. Unfortunately, GitLab doesn't provide an alternative to tokens. You can make the token expire regularly, but you'll need to replace it. That said, while logged in to GitLab as root, click on the colorful icon in the upper left-hand corner and click **Preferences**:

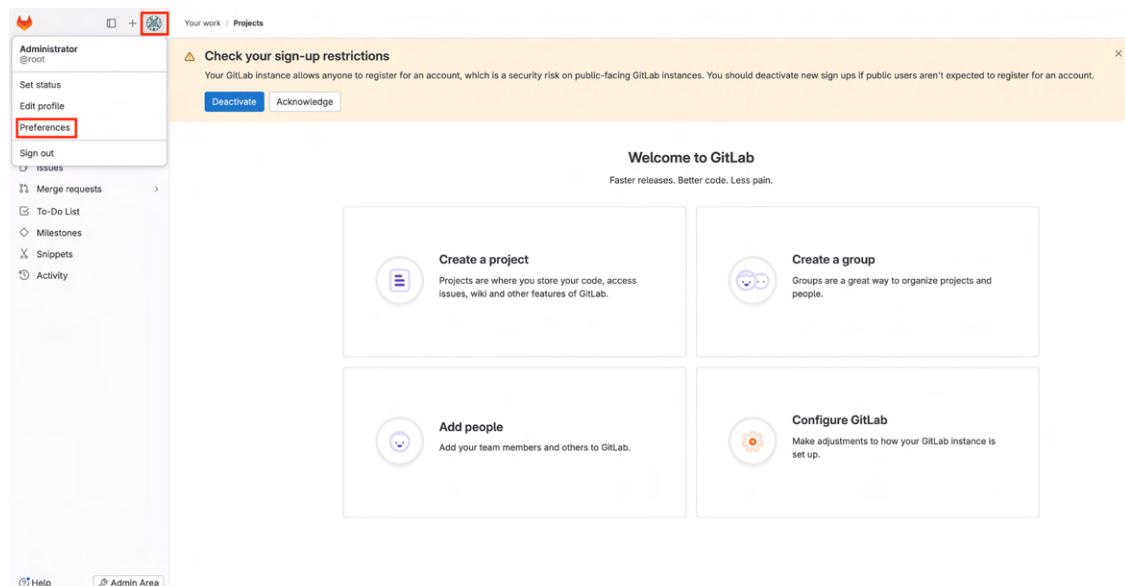


Figure 19.6: GitLab preferences

Once the Preferences screen loads, click on Access Tokens and then Add new token:

The screenshot shows the GitLab User Settings interface. On the left, a sidebar lists various settings categories: Profile, Account, Applications, Chat, Access Tokens (which is selected and highlighted with a blue border), Emails, Password, Notifications, SSH Keys, GPG Keys, Preferences, Comment Templates, Active Sessions, Authentication Log, and Usage Quotas. At the bottom of the sidebar are links for Help and Admin Area.

The main content area is titled "User Settings / Access Tokens". It features a search bar labeled "Search settings". Below it is a section titled "Personal Access Tokens" with a sub-section "Active personal access tokens 0". A table header for this section includes columns for "Token name", "Scopes", "Created", "Last Used", "Expires", and "Action". To the right of the table is a red rectangular box highlighting the "Add new token" button. A message below the table states, "This user has no active personal access tokens."

Further down, there is a "Feed token" section with a placeholder token value consisting of several dots and two small icons (a copy icon and a refresh/circular arrow icon). A note below says, "Keep this token secret. Anyone who has it can read activity and issue RSS feeds or your calendar feed as if they were you. If that happens, [reset this token](#)."

Figure 19.7: GitLab access tokens

When the new token screen loads, you need to give it a name and an expiration and click on the **api** option to give it full access. Once that's done, click on the **Create personal access token** button at the bottom of the screen:

The screenshot shows the GitLab user settings interface for managing access tokens. The left sidebar lists various user settings categories, with 'Access Tokens' selected. The main content area is titled 'Personal Access Tokens' and explains that tokens can be generated for applications needing API access or for Two-Factor Authentication. A sub-section titled 'Active personal access tokens' shows a table with columns for Token name, Scopes, Created, Last Used, Expires, and Action. Below this is a form for 'Add a personal access token' with fields for Token name (containing 'openunison'), Expiration date (set to 2024-07-23), and a list of scopes. The 'api' scope is checked and highlighted with a red box. Other scopes listed include read\_api, read\_user, create\_runner, manage\_runner, k8s.proxy, read\_repository, write\_repository, read\_registry, write\_registry, ai.features, sudo, admin\_mode, and read\_service\_ping. At the bottom of the token creation form are 'Create personal access token' and 'Cancel' buttons, both highlighted with red boxes.

User settings

- Profile
- Account
- Applications
- Chat
- Access Tokens
- Emails
- Password
- Notifications
- SSH Keys
- GPG Keys
- Preferences
- Comment Templates
- Active Sessions
- Authentication Log
- Usage Quotas

Help Admin Area

User Settings / Access Tokens

Search settings

## Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API. You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

### Active personal access tokens

Token name	Scopes	Created	Last Used	Expires	Action

### Add a personal access token

Token name

Expiration date

Select scopes

Scopes set the permission levels granted to the token. [Learn more](#).

**api**  
Grants complete read/write access to the API, including all groups and projects, the container registry, the dependency proxy, and the package registry.

**read\_api**  
Grants read access to the API, including all groups and projects, the container registry, and the package registry.

**read\_user**  
Grants read-only access to your profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.

**create\_runner**  
Grants create access to the runners.

**manage\_runner**  
Grants access to manage the runners.

**k8s.proxy**  
Grants permission to perform Kubernetes API calls using the agent for Kubernetes.

**read\_repository**  
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.

**write\_repository**  
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).

**read\_registry**  
Grants read-only access to container registry images on private projects.

**write\_registry**  
Grants write access to container registry images on private projects. You need both read and write access to push images.

**ai.features**  
Grants access to GitLab Duo related API endpoints.

**sudo**  
Grants permission to perform API actions as any user in the system, when authenticated as an admin user.

**admin\_mode**  
Grants permission to perform API actions as an administrator, when Admin Mode is enabled.

**read\_service\_ping**  
Grant access to download Service Ping payload via API when authenticated as an admin user

**Create personal access token** Cancel

This user has no active personal access tokens.

### Feed token

Your feed token authenticates you when your RSS reader loads a personalized RSS feed or when your calendar application loads a personalized calendar. It is visible in those feed URLs. It cannot be used to access any other data.

### Feed token

.....

Keep this token secret. Anyone who has it can read activity and issue RSS feeds or your calendar feed as if they were you. If that happens, [reset this token](#).

*Figure 19.8: Create a new GitLab personal access token*

Once the token is generated, the last step is to copy it and then configure it as a Pulumi configuration secret:

The screenshot shows the GitLab User Settings interface with the 'Access Tokens' section selected. A success message at the top states: 'Your new personal access token has been created.' Below this is a search bar and a section titled 'Personal Access Tokens'. A modal window is open, titled 'Your new personal access token', displaying a long string of characters (the token itself) which is highlighted with a red box. Below the token, a note says 'Make sure you save it - you won't be able to access it again.' To the right of the token are three small icons: a copy icon, a refresh icon, and a trash icon. Below the modal is a table titled 'Active personal access tokens' with one row of data. At the bottom of the page, there's a 'Feed token' section with a note about its purpose and a copy icon.

Token name	Scopes	Created	Last Used	Expires	Action
openunison	api	Jun 23, 2024	Never	in 4 weeks	<span style="font-size: 2em;">□</span>

Figure 19.9: GitLab personal access token

Copy the token and set it in the Pulumi configuration:

```
$ cd /path/to/venv/chapter19/pulumi
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword
$ pulumi config set gitlab.root.token \
  'glpat-suHtqupeNetAsYfGwVyz' --secret
```

We've now finished the extra steps needed to complete the control plane's configuration. Next, we'll complete the control plane rollout in our Pulumi program.

## Finishing the Control Plane Rollout

The next step is to rerun our Pulumi program to complete the integrations:

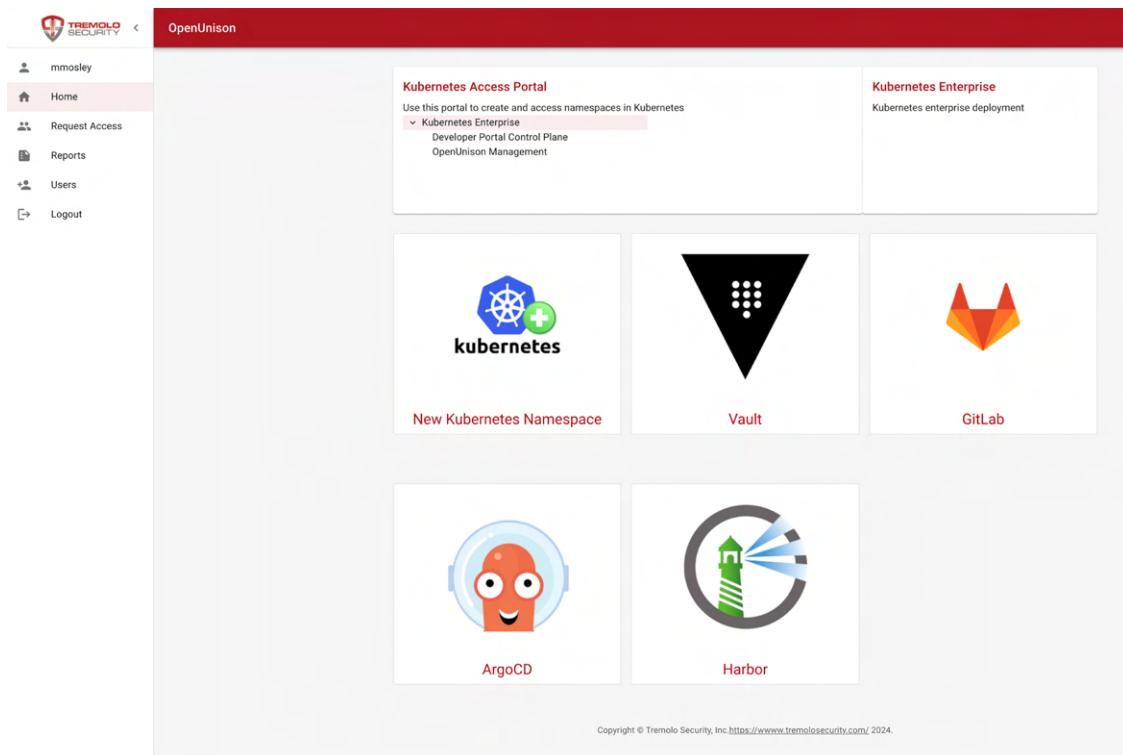
```
$ cd /path/to/venv/chapter19/pulumi
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword
$ pulumi up -y
```

This should take less time than the initial run but will still take a few minutes. OpenUnison needs to be rolled out again with the new configuration options, which will take the longest amount of time.

Once the rollout is done, we have one more task to complete on our control plane. We need to update NGINX to forward SSH on port 22 to the GitLab shell Service. We can do this by getting the name of the shell Service in the GitLab namespace and updating our control plane NGINX:

```
$ cd chapter19/scripts
$ ./patch_nginx_ssh.sh
```

Once NGINX is running again, you should be able to ssh into GitLab. At this point, you can log in to OpenUnison by accessing <https://k8sou.idp-cp.tremolo.dev> (replace idp-cp.tremolo.dev with your control plane suffix) and log in with the username mmosley and the password start123:



*Figure 19.10: OpenUnison Main Page*

We haven't finished integrating our dev or production systems yet, but you should login to Vault, GitLab, Harbor, ArgoCD, and the control plane Kubernetes using SSO with OpenUnison. Since you're the first person to log in, you are automatically both the control plane cluster administrator and the top-level approver.

With the control plane configured, the last step in Pulumi is to onboard the development and production clusters, which we'll cover next.

## Integrating Development and Production

So far, we've spent all our time on the control plane. There won't be any user workloads here, however. Our tenants will be on the development and production clusters. For our automation plan to work, we're going to need to integrate these clusters into OpenUnison so that we're using short-lived tokens for all automation API calls. Thankfully, OpenUnison already has everything we need, and it's been integrated into the OpenUnison Helm charts.

The first step is to deploy NGINX to each cluster. We're going to port-forward port 3306 for MySQL as well so that OpenUnison on the control plane can talk to MySQL on each cluster. While we could have used the control plane's MySQL for vClusters, we don't want to be in a situation where a problem on the control plane takes down either development or production. By running MySQL on each cluster, an outage on one doesn't stop the operations for another. Run the following:

```
$ export KUBECONFIG=/path/to/idp-dev.conf
$ helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  --set tcp.3306=mysql:mysql:3306
$ export KUBECONFIG=/path/to/idp-prod.conf
$ helm upgrade --install ingress-nginx ingress-nginx \
  --repo https://kubernetes.github.io/ingress-nginx \
  --namespace ingress-nginx --create-namespace \
  --set tcp.3306=mysql:mysql:3306
```

Once running, you'll be able to update the configuration for Pulumi.

Option	Description	Example
kube.dev.path	The path to the kubectl configuration file for your development cluster	/path/to/idp-dev
kube.dev.context	The Kubernetes context for the development cluster in its kubectl configuration	kubernetes-admin@kubernetes
kube.prod.path	The path to the kubectl configuration file for your production cluster	/path/to/idp-prod
kube.prod.context	The Kubernetes context for the prod cluster in its kubectl configuration	kubernetes-admin@kubernetes

Table 19.2: Configuration options for Kubernetes and paths in development and production clusters

Once your configuration is added, we can finish our Pulumi rollout:

```
$ cd /path/to/venv/chapter19/pulumi  
$ export PULUMI_CONFIG_PASSPHRASE=mysecretpassword  
$ pulumi up -y
```

This will take a few minutes to run, but once we're done, we'll have a final step in OpenUnison to finish the rollout. If all goes well, you should see something like:

```
Resources:  
+ 46 created  
~ 2 updated  
+-1 replaced  
49 changes. 68 unchanged  
  
Duration: 4m59s
```

Congratulations, our infrastructure is deployed! Three clusters, a dozen systems, all integrated! Next, we'll use OpenUnison's workflows to finish the last integration steps.

## Bootstrapping GitOps with OpenUnison

We've deployed several systems to support our GitOps workflows, and we've integrated them via SSO so that users can log in, but we haven't started the GitOps bootstrapping process. What we mean by "bootstrapping" in this context is to set up some initial repositories in GitLab, integrate them into Argo CD, and make it so they sync to the control plane, development, and production clusters. This way, as we add new tenants, we'll do so by creating manifests in Git instead of writing directly to the API servers of our clusters. We'll still write to the API servers for ephemeral objects, like Jobs we'll be using to deploy vClusters and integrate them with our control plane, but otherwise, we want to write everything into Git.

We're going to do this last step in OpenUnison instead of Pulumi. You may be wondering why we would use OpenUnison for this part when we could have used Pulumi. That was the original plan, but unfortunately, a known bug with Pulumi's GitLab provider kept us from being able to create groups in the GitLab Community Edition. Since OpenUnison's workflow engine has this capability already, and this is a step that would only ever be run once, we decided to just do it in OpenUnison's workflow engine.

With that said, log in to OpenUnison using the instructions from the last section. Next, click on **Request Access** on the left-hand side, choose **Kubernetes Administration**, and add the development and production clusters by adding **Kubernetes-prod Cluster Administrator** and **Kubernetes-dev Cluster Administrator** to your cart:

The screenshot shows the OpenUnison interface. On the left sidebar, under the 'Request Access' section, the 'Kubernetes Administration' role is highlighted with a red box. The main content area displays three cluster administrator roles: 'Developer Portal Control Plane Cluster Administrator', 'kubernetes-prod Cluster Administrator', and 'kubernetes-dev Cluster Administrator'. Each role has an 'ADD TO CART' button below it. A copyright notice at the bottom reads: 'Copyright © Tremolo Security, Inc. <https://www.tremolosecurity.com/> 2024.'

Figure 19.11: OpenUnison request access to Dev and Prod cluster administration

Once added to your cart, click on the new **Checkout** menu option on the left, add a reason for the request, and click on **SUBMIT YOUR REQUESTS**.

The screenshot shows the 'Finish Submitting Your Requests' page. The left sidebar has a 'Checkout' section with a red box around it. The main content area shows two submitted requests: 'kubernetes-prod Cluster Administrator' and 'kubernetes-dev Cluster Administrator'. Each request includes a 'Reason for request' field containing 'bootstrap', a 'Request For Someone Else' checkbox, and a 'REMOVE FROM CART' button. At the bottom center is a large red 'SUBMIT YOUR REQUESTS' button.

Figure 19.12: Submit access request for cluster administration

When you refresh your screen, you'll now see you have two open requests. Act on them just as you did in *Chapter 9, Building Multitenant Clusters with vClusters*, and log out. When you log back in, you'll have access to both the development and production clusters as well as the control plane.

Once you're logged back in, go back to **Request Access**, click on **OpenUnison Internal Management Workflows**, and add **Initialize OpenUnison** to your cart. Check it out of your cart with a reason, just as before. This time, there won't be an approval step. It will take a few minutes, but once it is done, you can log in to Argo CD and you'll see three new projects:

The screenshot shows the Argo CD application dashboard. On the left, there's a sidebar with navigation links for Applications, Settings, User Info, Documentation, Favorites Only, SYNC STATUS, HEALTH STATUS, and LABELS. The main area is titled "APPLICATIONS TILES" and contains three project cards. Each card has a star icon, a diamond icon, and the project name. Below each name is a brief description of the project's details. At the bottom of each card are three buttons: SYNC, REFRESH, and DELETE.

Project	Description
control-plane/k8s-cp-cp	Project: cluster-operations Status: Healthy Labels: Repository: git@gitlab-ssh.idp-cp2.tremolo.dev:clus... Target R.: HEAD Path: k8s-cp-cp Destination: in-cluster Namespace: cluster-operations Created: 06/23/2024 21:24:57 (a minute ago)
control-plane/k8s-cp-dev	Project: cluster-operations Status: Healthy Labels: Repository: git@gitlab-ssh.idp-cp2.tremolo.dev:clus... Target R.: HEAD Path: k8s-cp-dev Destination: k8s-cp-dev Namespace: cluster-operations Created: 06/23/2024 21:24:57 (a minute ago)
control-plane/k8s-cp-prod	Project: cluster-operations Status: Healthy Labels: Repository: git@gitlab-ssh.idp-cp2.tremolo.dev:clus... Target R.: HEAD Path: k8s-cp-prod Destination: k8s-cp-prod Namespace: cluster-operations Created: 06/23/2024 21:24:57 (a minute ago)

Figure 19.13: Argo CD after the OpenUnison initialization

The state is **Unknown** because we haven't yet trusted the keys from GitLab's ssh service. Download the Argo CD command-line utility using your favorite method and run the following:

```
$ argocd login --grpc-web \
--sso argocd.idp-cp.tremolo.dev
$ ssh-keyscan gitlab-ssh.idp-cp.tremolo.dev \
| argocd cert add-ssh --batch
```

This will add the correct keys to Argo CD. After a few minutes, you'll see that our applications in Argo CD are syncing! Now is a good time to look around both Argo CD and GitLab. You'll see how the basic scaffolding of our GitOps infrastructure will look. You can also look at the onboarding workflow in `chapter19/pulumi/src/helm/kube-enterprise-guide-openunison-idp/templates/workflows/initialization/init-openunison.yaml`. I think the most important thing you'll see is how we tie everything together via identity. This is something that is too often overlooked in the DevOps world. Especially in Argo CD, we create an `AppProject` that constrains which repositories and clusters can be added. We then create a `Secret` for each cluster, but the `Secret` doesn't contain any secret data. Finally, we generate `ApplicationSets` to generate the `Application` objects. We'll follow this pattern again when we deploy our tenants.

You now have a working multitenant IDP! It took about 30 pages of explanation, probably a few hours of cluster design and setup, and several thousand lines of automation, but you have it! Next, it's time to deploy a tenant!

## Onboarding a Tenant

So far, we've spent all our time setting up our infrastructure. We have a Git repository, clusters for our control plane as well as development and production, a GitOps controller, a secrets manager, and an SSO and automation system. Now we can build out our first tenant! The good news is this part is pretty easy and doesn't require any command-line tools. Like how we onboarded a vCluster in *Chapter 9, Building Multitenant Clusters with vClusters*, we're going to use OpenUnison as our portal for requesting and approving the new tenant. If you're already logged in to OpenUnison, log out and log back in, but this time, with the username `jjackson` and the password `start123`. You'll notice you have much fewer badges on the main page because you don't have access to anything yet! We'll fix that by creating a new tenant. Click on the **New Kubernetes Namespace** badge:

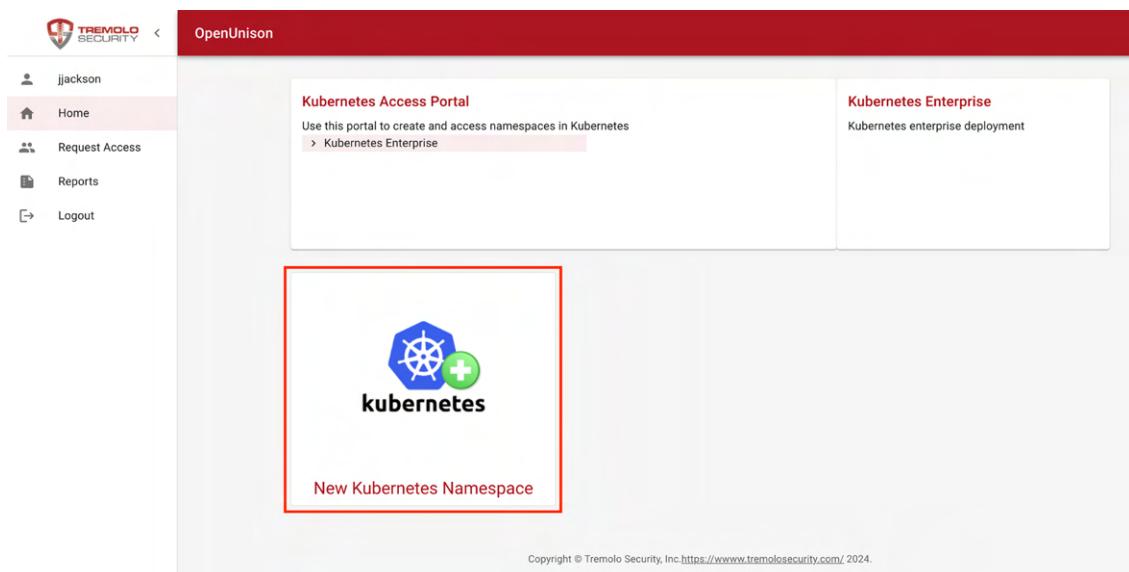


Figure 19.14: Log in to OpenUnison with jjackson

Once it's open, use the name `myapp` with the reason `new application` and then hit **SAVE**:

The screenshot shows the 'Create New Project' page of the OpenUnison interface. On the left, there's a sidebar with a Tremolo Security logo, a user icon, and navigation links for 'Home' (which is highlighted in pink) and 'Logout'. The main content area has a red header bar with the text 'OpenUnison'. Below it, a sub-header says 'Create New Project' with a sub-instruction: 'Use this page to request the creation of a new project. Once the project is approved you will be the first approver and owner. You will be notified when the project is ready.' There are two input fields: 'Namespace Name' containing 'myapp' and 'Reason' containing 'new application'. At the bottom are two buttons: 'SAVE' on the left and 'CANCEL REQUEST' on the right. A small footer at the bottom of the page reads 'Copyright © Tremolo Security, Inc. <https://www.tremolosecurity.com/> 2024.'

Figure 19.15: New Project Screen

Once saved, log out and log back in, but this time, as `mmosley` with the password `start123`. You'll see there's an open approval. Click on **Open Approvals**, **ACT ON REQUEST**, provide a justification, and click **APPROVE REQUEST**. When the button turns turquoise, click **CONFIRM APPROVAL**. This is going to take a while. It's not that the steps are very computationally expensive. The time is mostly waiting for systems to sync. That's because we're not writing directly to our cluster's API servers but are instead creating manifests in our GitLab deployment that are then synced into the clusters by Argo CD. We're also deploying multiple vClusters and OpenUnisons, which also takes time. If you want to watch the progress, you can watch the logs in the `openunison-orchestra` pod on the control plane. This can take 10 to 15 minutes to fully roll out. We don't have a working email server, so you'll know when the workflow is done by logging in to Argo CD and you'll see two new applications: one for our development vCluster and one for our production vCluster:

The screenshot shows the Argo CD application dashboard. On the left, there's a sidebar with sections for 'Applications' (selected), 'Settings', 'User Info', 'Documentation', 'Favorites Only' (0), 'SYNC STATUS' (Unknown: 0, Synced: 3, OutOfSync: 2), 'HEALTH STATUS' (Unknown: 0, Progressing: 0, Suspended: 0, Healthy: 5, Degraded: 0, Missing: 0), 'LABELS', 'PROJECTS', 'CLUSTERS', and 'NAMESPACES'. The main content area displays four application cards in a grid:

- control-plane/k8s-cp-cp**: Project: cluster-operations, Status: Healthy, Synced, Repository: git@gitlab-sh:ldp-cp2.tremolo.dev.clus..., Target R.: HEAD, Path: k8s-cp-cp, Destinat...: in-cluster, Created ...: 07/05/2024 08:42:53, Last Sync: 07/05/2024 09:28:06.
- control-plane/k8s-cp-dev**: Project: cluster-operations, Status: Healthy, OutOfSync, Repository: git@gitlab-sh:ldp-cp2.tremolo.dev.clus..., Target R.: HEAD, Path: k8s-cp-dev, Destinat...: k8s-cp-dev, Created ...: 07/05/2024 08:42:53, Last Sync: 07/05/2024 09:28:07.
- control-plane/k8s-cp-prod**: Project: cluster-operations, Status: Healthy, Synced, Repository: git@gitlab-sh:ldp-cp2.tremolo.dev.clus..., Target R.: HEAD, Path: k8s-cp-prod, Destinat...: k8s-cp-prod, Created ...: 07/05/2024 08:42:53, Last Sync: 07/05/2024 09:28:07.
- myapp/k8s-myapp-dev**: Project: myapp-dev, Status: Healthy, Synced, Repository: git@gitlab-sh:ldp-cp2.tremolo.dev/mya..., Target R.: HEAD, Path: yaml, Destinat...: k8s-myapp-dev, Created ...: 07/05/2024 09:18:05, Last Sync: 07/05/2024 09:21:06.

At the top of the main content area, there are buttons for '+ NEW APP', 'SYNC APPS', 'REFRESH APPS', and a search bar. On the right, there are buttons for 'Log out', 'APPLICATIONS TILES', and a sort/filter dropdown. The bottom right corner shows 'Sort: name ▾ Items per page: 10 ▾'.

Figure 19.16: Argo CD after the new tenant is deployed

If you click on the `myapp/k8s-myapp-prod` application, you'll see we don't have much synchronized in:

The screenshot shows the Argo CD interface with the following details:

- APPLICATIONS**: Shows one application named `k8s-myapp-prod`.
- APP HEALTH**: Shows the application is **Healthy**.
- SYNC STATUS**: Shows the application is **Synced** to HEAD (6a1486d).
- LAST SYNC**: Shows a successful sync 4 minutes ago.
- APPLICATION DETAILS TREE**: Displays a hierarchical tree of Kubernetes namespaces and their components. Most components are in a **Synced** state, except for a few in the `pull-secret` namespace which are in a **Syncing** state.

Figure 19.17: Production tenant Argo CD application

What we've created at this point is a `ServiceAccount` that can communicate with Vault and an `ExternalSecret` that syncs the pull secret we generated for Harbor into the default namespace. Next, we'll want to look at our new tenant! Make sure to log out of everything, or just open an incognito/private window in a different browser and log in as `jjackson` again.

The screenshot shows the OpenUnison interface for the user `jjackson`:

- Kubernetes Access Portal**: A section where the user can manage Kubernetes namespaces. It lists existing namespaces: `kubernetes-enterprise`, `kubernetes-dev`, `kubernetes-prod`, `myapp-dev`, and `myapp-prod`.
- Kubernetes Enterprise**: A link to a Kubernetes enterprise deployment.
- Components**: A grid of icons representing various tools:
  - New Kubernetes Namespace**: Kubernetes icon.
  - ArgoCD**: ArgoCD icon.
  - GitLab**: GitLab icon.
  - Harbor**: Harbor icon.
  - Vault**: Vault icon.

Figure 19.18: OpenUnison as jjackson after tenant deployment

You'll see that we now have badges for our cluster management apps. When you log in to them, you'll see that your view is limited. Argo CD only lets you interact with the Application objects for your tenant. GitLab only lets you see the projects associated with your tenant. Harbor only lets you see the images for your tenant, and finally, Vault only lets you interact with the secrets for your tenant. If you look in the myapp-dev and myapp-prod sections of the tree at the top of the screen, you'll see you now have tokens and a dashboard for your two vClusters too. Isn't identity amazing?

So far, we've built out a tremendous amount of infrastructure and created a tenant using a common identity, allowing each system's own policy-based system to determine how to draw boundaries. Next, we'll deploy an application to our tenant.

## Deploying an Application

We've built out quite a bit of infrastructure to support our multitenant platform and now have a tenant to run our application in place. Let's go ahead and deploy our application!

If we log in to GitLab as jjackson, we'll see there are three projects:

- **myapp-prod/myapp-application:** This repository will store the code for our application and the build to generate our container.
- **myapp-dev/myapp-ops:** The repository for the manifests for our development cluster.
- **myapp-prod/myapp-ops:** Where the production cluster's manifests are stored.

There's no direct fork from the development project to the production project. That was our original intent, but that stringent path from development to production doesn't work well. Development environments and production environments are rarely the same and often have different owners of infrastructure. For example, I maintain a public safety identity provider where our development environment doesn't have trust established with all the jurisdictions that our production environment does. To address this, we set up an additional system to stand in for those identity providers. These variances make it difficult to have a direct line to automatically merge changes from development into production.

With that said, let's build out our application. The first step is to generate an SSH key and add it to jjackson's profile in GitLab so we can check the code in. Once you have your SSH key updated, clone the myapp-dev/myapp-ops project. This project has the manifests for your development cluster. You'll see there are already manifests to support synchronizing the pull secret for Harbor into the default namespace. Create a folder called yaml/namespaces/default/deployments and add chapter19/examples/ops/python-hello.yaml to it. Commit and push to your GitLab. Within a few minutes, Argo CD will attempt to sync it, which will fail because the image points to something that doesn't exist. That's OK. Next, we'll take care of that.

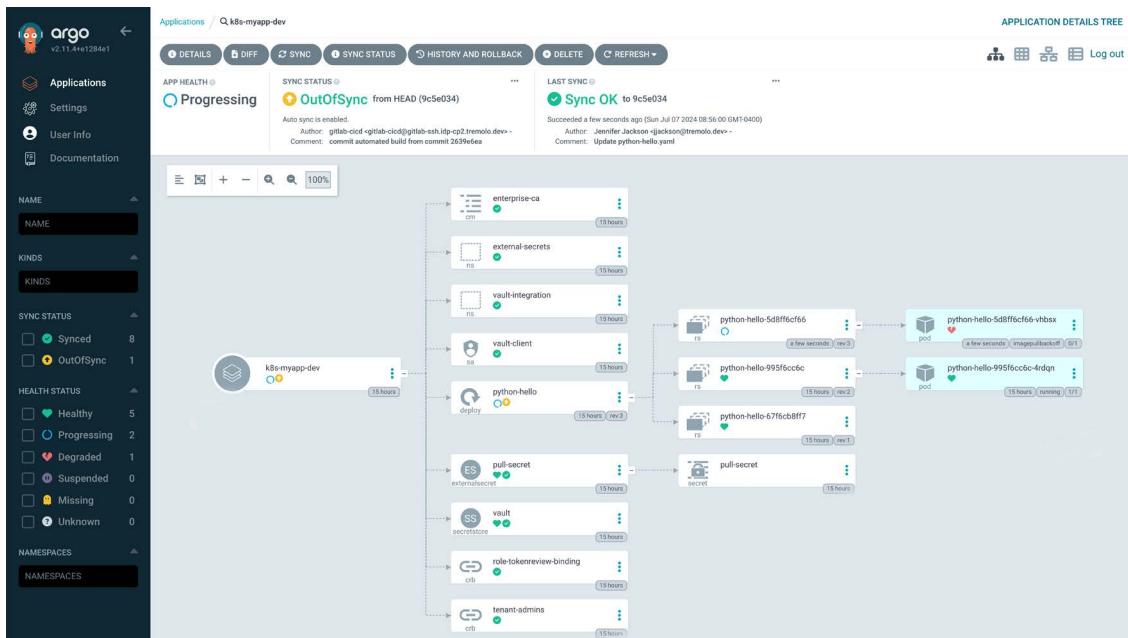


Figure 19.19: Broken Argo CD Rollout

The image tag in our deployment will be updated upon a successful build of our application. This gives us the automation we’re looking for and the ability to easily roll back if we need to. Our application is a simple Python web service. Clone the `myapp-prod/myapp-application` project and copy it in the `chapter19/examples/myapp` folder. There are two items to make note of:

- **source:** This folder contains our Python source code. It’s not very important and we won’t spend any time on it.
- **.gitlab-ci.yml:** This is the GitLab build script that’s responsible for generating a Docker container, pushing it into Harbor, then patching our Deployment in Git.

If you look at the `.gitlab-ci.yml` file, you might notice that it looks similar to the Tekton tasks we built in previous editions. What’s similar about GitLab’s pipelines is that each stage is run as a pod in our cluster. We have two stages. The first builds our container and the second deploys it.

The build stage uses a tool from Google called **Kaniko** (<https://github.com/GoogleContainerTools/kaniko>) for building and pushing Docker images without needing to interact with a Docker daemon. This means our build container doesn’t require a privileged container and makes it easier to secure our build environment. Kaniko uses a Docker configuration to manage credentials. If you are using AWS or any of the other major clouds, you can integrate directly with their IAM solution. In this instance, we’re using Harbor, so our OpenUnison workflow provisioned a `Docker config.json` as a variable into the GitLab project. The build process uses the short version of the Git SHA hash as a tag. This way, we can track each container to the build and the commit that produced it.

The second stage of the build is the deployment. This stage first checks out our `myapp-dev/myapp-ops` project, patches our Deployment with the correct image URL, and then commits and pushes it back to GitLab. Unfortunately, neither GitLab nor GitHub makes it easy to check out code using the identity of workflow. To make this work, our OpenUnison onboarding workflow created a “deployment key” for our Ops project, marking it as writeable, and then added the private key to the app project. This way, the app project can configure SSH to allow for cloning the DevOps repository, generating the patch, and then committing and pushing. Once completed, Argo CD will detect the change within a few minutes and synchronize it into your development vCluster.

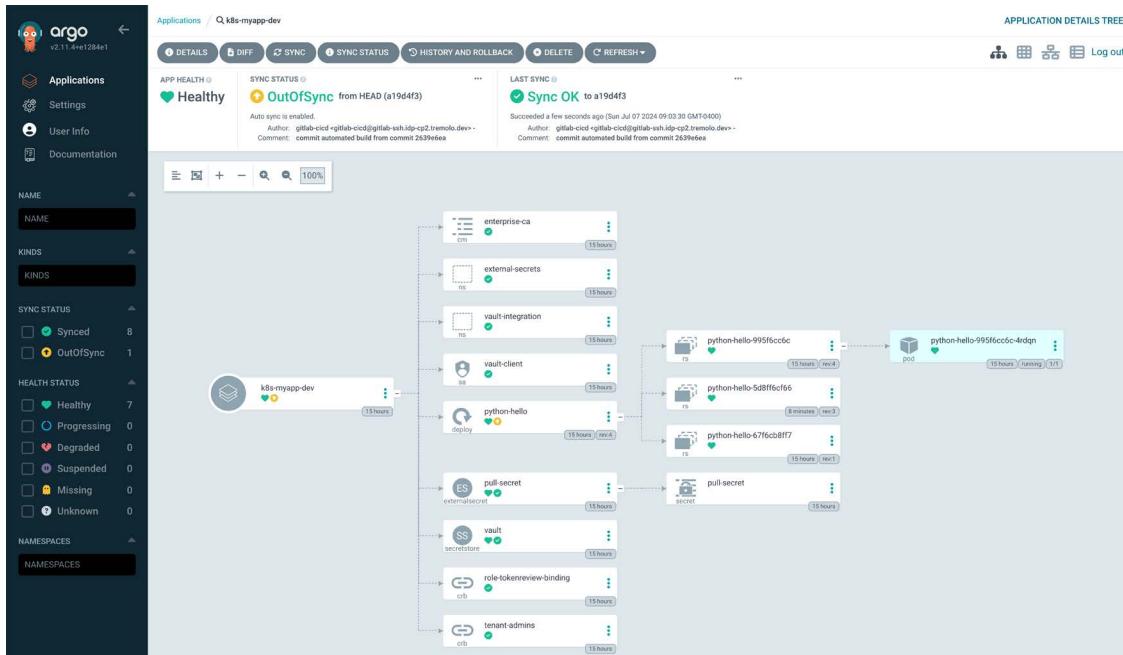


Figure 19.20: Argo CD with a working synced Deployment

Assuming everything went well, you now have an automatically updated development environment! You’re now ready to run any automated tests before promoting to production. Since we don’t currently have an automated process for this, let’s explore how we can approach it.

## Promoting to Production

So far, we've deployed our application into our development vCluster. What about our production vCluster? There's no direct relationship between the `myapp-dev/myapp-ops` project in GitLab and the `myapp-prod/myapp-ops` projects. If you read the first two editions of this book, you might remember that the dev project was a fork of the prod project. You would promote a container from development to production by submitting a merge request (GitLab's version of a pull request) and, once approved, the changes in development would be merged into production allowing Argo CD to synchronize them into our cluster. The problem with this approach is it assumes dev and prod are the exact same, and that's never true. Something as simple as a different hostname for a database would have broken this approach. We needed to break this pattern to make our cluster usable.

For our own lab, the easiest way to go is to create the `yaml/namespaces/default/deployments` directory in the `myapp-prod/myapp-ops` project, add `chapter19/examples/ops/python-hello.yaml` to it, and update the image to point to our current image in Harbor. This is not a well-automated process, but it would work. Eventually, Argo CD will finish synchronizing the manifests and our service will be running on our production cluster.

If you're looking to automate this more, you have multiple options:

- **Create GitLab workflows:** A workflow could be used to automate the updates in a similar way as production. You would need a way to trigger it, but this solution works nicely because you can leverage GitOps to track the rollouts.
- **Create custom jobs or scripts:** We're running Kubernetes, so there are multiple ways to create batch jobs to run the upgrade process. Depending on how you choose to run batch jobs in Kubernetes, this can also be done using GitOps.
- **Akuity Kargo:** A new project focused on solving this problem in a consistent way.

The first two options are variations on the same theme: a customized rollout script. There's nothing that says this is an antipattern. It just seems there's probably a better way, or at least a more consistent way, to do this. Enter the Akuity Kargo project (<https://github.com/akuity/kargo>), not to be confused with the Container Craft Kargo project! Akuity was founded by many of the original developers of Argo CD. The Kargo project is not under the Argo umbrella; it's completely separate. It takes an interesting approach to automating the process of syncing across repositories to promote systems across environments. We originally thought we'd integrate this tool directly into our cluster, but it's still pre-version-1, so we decided to give it a mention instead. It's certainly a project we'll be keeping our eyes on!

Now that we've rolled out our application to production, what can we do next? We can add more users of course! We'll explore that and how we can expand on our platform next.

## Adding Users to a Tenant

Now that we have our tenant created and our application deployed, it might be a good time to look at how we add new users. The good news is that we can add more users! OpenUnison makes it easy for team members to request access. Log in to OpenUnison, click on **Request Access**, and pick the application and the role you want. The application owner will be responsible for approving that access. The great thing about this approach is that the cluster owners never get involved. It's entirely up to the application owners who have access to their system. That access is then provisioned, just in time, into all of the components of your platform.

The screenshot shows the OpenUnison web interface. At the top, there's a navigation bar with the Tremolo Security logo and the text "OpenUnison". On the left, a sidebar menu includes "jjackson", "Home", "Request Access" (which is highlighted in pink), "Reports", and "Logout". The main content area has a heading "Request Access" with the sub-instruction "Use this portal to create and access namespaces in Kubernetes". Below this is a dropdown menu under "Developer Portal Control Plane" with options: "Approvers", "Delete Namespace", "Developers", "Operations" (which is highlighted in pink), and "Owners". To the right of this is a "Operations" section with "Project Operations". At the bottom of the main content area is a "Filter by label" input field. Below the main content is a separate box labeled "myapp Operations" with the sub-instruction "project operation for the myapp namespace" and a red "ADD TO CART" button. At the very bottom of the page is a small copyright notice: "Copyright © Tremolo Security, Inc. <https://www.tremolosecurity.com/> 2024."

Figure 19.21: Adding users to application roles in OpenUnison

We've deployed our platform, and we know how to deploy a tenant and how to add new members to that tenant. What can we do to improve our new platform? Where are the gaps? We'll walk through those next.

## Expanding Our Platform

We've covered quite a bit in the last two chapters to build out a multitenant platform. We walked through how GitOps works, different strategies, and how IaC tools like Pulumi make automation easier. Finally, we built out our multi-tenant platform over three clusters. Our platform includes Git and builds using GitLab, secrets management using Vault, GitOps with Argo CD, a Docker registry in Harbor, and finally, it's all integrated via identity using OpenUnison. That's it, right? No, unfortunately not. This section will cover some of the gaps or areas where our platform can be built out. First, we'll start with identity.

## Different Sources of Identity

One area we have taken a really focused view on throughout this book is how a user's identity crosses various boundaries of the systems that make up our clusters. In this platform, we use our Active Directory for user authentication and use OpenUnison's internal groups for authorization. Similar to *Chapter 9*, we could also integrate our enterprise's groups for authorization. We can also expand outside of Active Directory to use Okta, Entra ID (formerly Azure AD), GitHub, etc. A great addition is to integrate multi-factor authentication. We've said it multiple times throughout this book, but it bears repeating: multi-factor authentication is one of the easiest ways to help lock down your environment!

Having looked at how else to identify users, let's look at monitoring and logging.

## Integrating Monitoring and Logging

In *Chapter 15, Monitoring Clusters and Workloads*, we learned how to monitor Kubernetes with Prometheus and aggregate logs using OpenSearch. We didn't integrate either of these systems into our platform. We did this for a few reasons:

- **Simplicity:** These last two chapters were complex enough; they didn't need more stuff to integrate!
- **Lack of multi-tenancy:** Prometheus has no concept of identity and the Grafana Community edition only allows two roles. It does appear that OpenSearch supports multi-tenancy, but that would have required considerable engineering.
- **Complexity with vCluster:** This is similar to the multi-tenancy issue; would we have had a Prometheus for each vCluster? There are ways to do this, but they would likely require their own book.

Given that this solution is designed for a production environment, you would want to integrate some kind of monitoring and logging tied directly into both your host clusters and your vClusters. That can get complex, but would be well worth it. Imagine an application owner being able to view all their logs from a single location without having to log in to a cluster. The tools are all there, it's just a matter of integration!

We know it's important to monitor a cluster, but let's next talk about policy management.

## Integrating Policy Management

This book covered two chapters on policy management and included authorization management in our chapter on Istio. It's an important aspect of Kubernetes that we didn't include in our platform. We also spent a chapter on runtime security. We decided not to include policy management because while it would be needed for production, it didn't provide any additional benefit for the lab itself. It should definitely be included for any production deployment, though!

Now that we've covered which technologies we didn't include in our platform, let's talk about what you could replace.

## Replacing Components

We chose the components we did for our platform because we wanted to show how to build off what we learned throughout the book. We made a conscious decision to use only open source projects and to avoid any kind of service that would require a sign-up or trial. With that said, you could replace Vault with a service like [AKeyLess](#), replace GitLab with GitHub, etc. It's your environment, so deploy what works best for you!

The story is far from over on our platform, but this does bring us to the end of our book!

## Summary

This chapter covered considerable ground. We started by looking at how to build out three Kubernetes clusters to support our platform. Once we had our clusters, we deployed cert-manager, Argo CD, MySQL, GitLab, Harbor, Vault, and OpenUnison via Pulumi, integrating them all. With our platform in place, we deployed a tenant to see how to use automation and GitOps to simplify our management, and finally, talked about different ways to update and adjust our multi-tenant platform. That's quite a lot of ground to cover in just one chapter!

I want to say thank you to Kat Morgan for writing the Pulumi code I used as the starting point for this chapter and helping me out when I ran into issues.

Finally, *thank you!* Through 19 chapters and dozens of different technologies, labs, and systems, you joined us on our fantastic journey through the enterprise cloud-native landscape. We hope you had as much fun reading and working with this book as we did writing it. Please, if you run into issues or just want to say hi, open an issue on our GitHub repository. If you have thoughts or ideas, that would be great as well! We cannot say this enough, but once again, thank you for joining us!

## Questions

1. Kubernetes has an API for trusting certificates:
  - a. True
  - b. False
2. Where can the Pulumi store state?
  - a. Local file
  - b. S3 bucket
  - c. Pulumi's SaaS service
  - d. All of the above
3. What sources of identity can OpenUnison use?
  - a. Active Directory
  - b. Entra ID (formerly Azure AD)
  - c. Okta
  - d. GitHub
  - e. All of the above

4. GitLab lets you check out code from another repository using your workflow's token:
  - a. True
  - b. False
5. Argo CD can check repositories for changes:
  - a. True
  - b. False

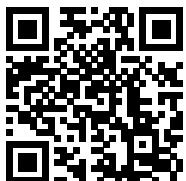
## Answers

1. b: False: The node's operating system must trust the remote certificate.
2. d: All of these are valid options.
3. e: OpenUnison supports all of these options.
4. b: False: Neither GitLab nor GitHub supports workflow tokens to check out other repositories.
5. a: True: Letting Argo CD check for updates to a repository instead of using a webhook can simplify deployment.

## Join our book's Discord space

Join the book's Discord workspace for a monthly *Ask Me Anything* session with the authors:

<https://packt.link/K8EntGuide>







[packt.com](http://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?

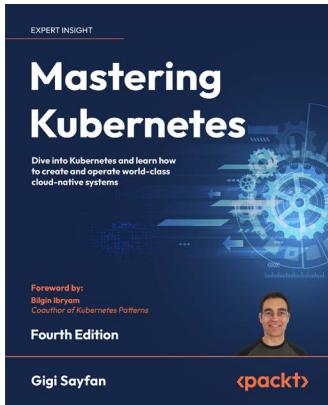
- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print,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

At [www.packt.com](http://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.



# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



## Mastering Kubernetes

Gigi Sayfan

ISBN: 9781804611395

- Learn how to govern Kubernetes using policy engines
- Learn what it takes to run Kubernetes in production and at scale
- Build and run stateful applications and complex microservices
- Master Kubernetes networking with services, Ingress objects, load balancers, and service meshes
- Achieve high availability for your Kubernetes clusters
- Improve Kubernetes observability with tools such as Prometheus, Grafana, and Jaeger
- Extend Kubernetes with the Kubernetes API, plugins, and webhooks

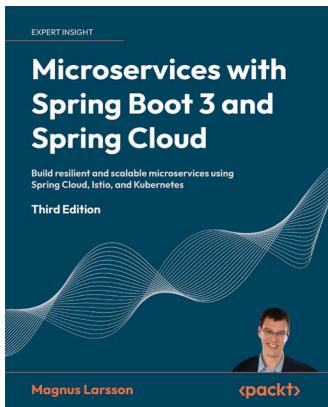


## Solutions Architect's Handbook

Saurabh Srivastava, Neelanjali Srivastav

ISBN: 9781835084236

- Explore various roles of a solutions architect in the enterprise
- Apply design principles for high-performance, cost-effective solutions
- Choose the best strategies to secure your architectures and boost availability
- Develop a DevOps and CloudOps mindset for collaboration, operational efficiency, and streamlined production
- Apply machine learning, data engineering, LLMs, and generative AI for improved security and performance
- Modernize legacy systems into cloud-native architectures with proven real-world strategies
- Master key solutions architect soft skills



## Microservices with Spring Boot 3 and Spring Cloud, Third Edition

Magnus Larsson

ISBN: 9781805128694

- Build reactive microservices using Spring Boot
- Develop resilient and scalable microservices using Spring Cloud
- Use OAuth 2.1/OIDC and Spring Security to protect public APIs
- Implement Docker to bridge the gap between development, testing, and production
- Deploy and manage microservices with Kubernetes
- Apply Istio for improved security, observability, and traffic management
- Write and run automated microservice tests with JUnit, test containers, Gradle, and bash
- Use Spring AOT and GraalVM to native compile the microservices
- Use Micrometer Tracing for distributed tracing

## Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](http://authors.packtpub.com) and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

## Share your thoughts

Now you've finished *Kubernetes - An Enterprise Guide, Third Edition*, we'd love to hear your thoughts! If you purchased the book from Amazon, please [click here](#) to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

# Index

## A

**Accuknox** 135

**Active Directory (AD)** 289

using, with Kubernetes 185, 186

**Active Directory groups**

mapping, to RBAC RoleBindings 186

**advanced pool configurations** 123

automatic address assignments, disabling 123

buggy networks, handling 128, 129

IP pool scoping 127, 128

multiple address pools, using 125-127

static IP address, assigning to service 123, 124

**AKeyLess** 622

**Alertmanager** 421

alerting with 450, 451

configuring 451-453

metrics-based alerts 453-455

silencing alerts 456

**Amazon CodeBuild** 207

**Amazon's Route53** 140

**Amazon Web Services (AWS)** 79

**ambient mesh** 519, 520

advantages 519, 520

reference link 519

**ApacheDS** 189

**API gateway**

need for 560

**API interactions**

sequence 176-178

**API server**

interacting with 64

kubectl commands 66

Kubernetes kubectl utility, using 64

verbose option 65

**apiservices** 70

**AppArmor** 373

**application** 272

adding, to K8GB with custom resources 163, 164

adding, to K8GB with Ingress annotations 164

deploying 616-618

deploying, into service mesh 505, 506

promoting, to production vCluster 619

**ArgoCD** 176, 281, 569, 573

**ASCII** 243

**ASP (Active Server Pages)** 524

**audit2rbac**

using, to debug policies 235-239

**auditing**

enabling, on cluster 232- 235

**audit policy** 231

creating 231, 232

**authentication** 178, 343

**authentication methods** 302

**authentication options** 180  
certificates 180, 181  
custom authentication webhooks 185  
service accounts 181-183  
TokenRequest API 183, 185

**authentication, with cloud-managed clusters**  
impersonation, to integrate 196, 197

**authorization** 343

**Authorization header** 177

**authorization policies** 489-492  
access, denying and allowing 492, 493  
actions 491  
GET methods, allowing to workload 493, 494  
reference link 491  
requests, allowing from specific source 494  
rules 491  
scope 491

**automated testing framework** 319

## B

**Backstage** 570

**backup**  
deployment, restoring from 429  
failure, simulating 429  
restoring from 428, 429  
restoring, in new cluster 434, 435  
restoring, to new cluster 433  
using, to create workloads in new cluster 431

**base64 encoded string**  
versus Configuration Data 242

**BeyondCorp**  
reference link 347

**Boutique app** 505

**buggy networks**  
handling 128, 129

**busybox deployment** 429

## C

**Capture the Flag (CTF)** 303

**Cedar** 546

**Certificate Authority (CA)** 180, 290  
certificates 180, 181

**CertificateSigningRequests** 71

**CI/CD Proxy**  
reference link 212

**Citadel** 483

**Class-B** 119

**Cloud-Managed Kubernetes**  
using 590

**cluster**  
auditing, enabling on 232-235  
backing up 431, 432  
building 433  
deleting 436  
elements 272  
interacting with 26  
Kubernetes Global Balancer (K8GB),  
deploying to 156  
privileged access to 205

**cluster, for impersonation**  
configuring 199, 200  
default groups 203  
impersonation, testing 200, 201  
impersonation, using for debugging 201, 202  
impersonation, using without OpenUnison 202  
Inbound Impersonation 203-205  
privilege, authorizing temporarily 206, 207  
privileged access to clusters 205  
Privileged User Account, using 205  
Privileged User, impersonating 206  
RBAC policies 202, 203

**cluster policies**  
enforcing 355, 356

- ClusterRoleBindings** 71, 225, 226
- ClusterRoles** 71, 224, 225
  - combining, with RoleBindings 227
  - versus Roles 222
- CNCF (Cloud Native Computing Foundation)** 31
- CNCF project** 135
- Command-Line Interface (CLI)** 174
- Common Expression Language (CEL)** 338
  - reference link 338
- Common Vulnerabilities and Exposures (CVE)** 348, 350, 564
- components, customizing with kubeadm API**
  - reference link 43
- ComponentStatus** 71
- compute requirements**
  - Cloud-Managed Kubernetes, using 591
  - fulfilling 590
  - home lab, building 591
- Confidentiality Integrity Availability (CIA)** 245
- ConfigMaps** 71-73
- Configuration Data**
  - versus Secrets 242-244
- console**
  - logs, tracing from container 466-468
- constraint violations**
  - debugging 356-360
- Container Craft Kargo** 591
- Container Essentials** 1
- container ID** 6
- containerization**
  - need for 2, 3
- container layer** 6, 7
- container logs** 464
- Container Network Interface (CNI)** 29
- Container Runtime Interface (CRI)** 3
- containers** 1
  - breakouts 345-347
- designing 347
- designing, consideration 347
- ephemeral 5, 6
- services running, accessing 8
- used, for tracing logs to console 466-468
- versus virtual machine (VM) 344, 345
- Container Storage Interface (CSI)** 80
- continuous integration and continuous deployment (CI/CD) pipelines** 376
- ControllerRevisions** 73
- control plane** 479
  - cloud-controller-manager 63
  - creating, with istiod 482
  - customizing 42, 43
  - etcd database 60, 61
  - exploring 59
  - kube-controller-manager 62
  - Kubernetes API server 59, 60
  - kube-scheduler 62
- copy-on-write** 7
- CoreDNS** 129, 135
  - DNS forwarding, testing to 150-152
  - ETCD zone, adding to 142-144
  - exposing, to external requests 149, 150
  - integrating, with enterprise DNS server 147-149
- CronJobs** 73, 74
- CSI drivers** 74
- CSI nodes** 74
- CSIStorageCapacities** 74
- custom authentication webhooks** 185
- custom Kind cluster**
  - creating 43, 44
- custom load balancer**
  - adding, for Ingress 47, 48
- Custom Resource Definition (CRD)** 74, 163, 188, 223, 266, 274, 413, 488
- custom resources**
  - used, for adding application to K8GB 163, 164

**D****DaemonSets** 75, 414**dashboard**

integrating, with OpenUnison 308-310

**dashboard security issues**

exploring 304

**dashboard security risks** 303

token, using to log in 304

unencrypted connections 304, 305

**data**

visualizing, with Grafana 457

**data log**

viewing, in Kibana 469-474

**data plane** 479**Default Restrictive policy** 358**denial-of-service attack** 128**deployment** 75

restoring, from backup 429

**destination rules** 498**development clusters**

using 26, 27

**Dikastes** 45**Disaster Recovery (DR)** 155**distroless images**

debugging 348-350

drawbacks 348

using 348-350

**DNS forwarding**

testing, to CoreDNS 150-152

**Docker** 4, 5

installation link 10

installing 8-10

installing, on Ubuntu 10, 11

preparing, to install 10

URL 10

versus Moby 4, 5

**Docker CLI**

docker attach 16, 17

docker build 20

docker exec 18

docker help 13

docker logs 18

docker ps 15

docker pull/run 20

docker rm 19

docker run 13, 14

docker start and stop 16

using 13

**Docker, guide**

reference link 13

**Docker image** 7**Docker networking** 32**Docker permissions**

granting 12, 13

**Domain Name System (DNS)** 88

used, for resolving services 100, 101

**domain-specific language (DSL)** 338**dynamic admission controllers** 314, 315

mutating 314

validating 314

**E****EDNS0** 157**endpoints** 75, 88**EndPointSlices** 75, 76**Enterprise**

Secrets, managing 244

**enterprise DNS server**

used, for integrating CoreDNS 147-149

**enterprise identities**  mapping to Kubernetes, for authorizing access  
    to resources 228, 229**Enterprise Resource Planning (ERP) system** 540**Entra** 187

**environment variables** 263  
  Kubernetes Secrets, using 263-265  
  Vault Sidecar, using 265, 266

**envoy filters** 503  
  reference link 503

**Envoy proxies** 479

**etcd backup**  
  performing 404  
  required certificates, backing up 405

**etcdctl utility** 404-406

**etcd database**  
  backing up 405-407

**etcd-io Git repository**  
  reference link 405

**ETCD zone**  
  adding, to CoreDNS 142-144

**Events resource** 76

**extended Berkley Packet Filter (eBPF)** 372

**ExternalDNS** 139-141  
  configuration options 144, 145  
  setting up 141

**ExternalDNS and CoreDNS**  
  integrating 141, 142

**ExternalDNS integration**  
  used, for creating LoadBalancer service 145-147

**external requests**  
  CoreDNS exposing to 149, 150

**External Secrets Manager** 248

**External Secrets Operator** 255, 256  
  URL 254

**External Services**  
  accessing, from vClusters 283-286

**external users**  
  authentication methods 172

## F

**feature gates** 42  
**Felix** 45  
**FlowSchemas** 76

**FooWidgets** 105, 141, 185  
  requirements 105

## G

**Galley** 483

**Gatekeeper** 313, 318, 371, 569  
  deploying 318  
  used, for enforcing node security 351  
  versus Pod Security Admission (PSA) 352, 353  
  versus Pod Security Policies (PSP) 352, 353

**Gatekeeper cache**  
  enabling 331

**gateways** 495, 496

**Git** 565

**GitLab** 569

**GitLab configuration**  
  GitLab Runner, generating 601-604  
  Personal Access Token, generating 604-607

**GitOps** 249, 565  
  bootstrapping, with OpenUnison 610-613

**GitOps strategy**  
  designing 582-585

**Global Server Load Balancing (GSLB)** 153

**GNU Privacy Guard (GnuPG)** 11

**Google Cloud Platform (GCP)** 28

**GPG public key** 11

**Grafana** 281  
  graph, creating 457  
  used, for visualizing data 457

**Grype** 348

## H

**HAProxy configuration file** 49, 50

**HAProxy traffic flow** 51, 52

**Harbor** 569

**HashiCorp Vault** 569

**headless service** 98

**Helm** 142

reference link 142

using, to install K8GB 160

**Helm chart values**

customizing 158, 159

**highly available vCluster** 286-288

creating 286

operating 286

**home lab**

building 591

**Horizontal Pod Autoscalers (HPAs)** 76

**Host Cluster Identity** 284

**Hybrid** 248

**hypervisor** 23

## I

**identity federation** 174

**Identity Provider (IdP)** 266, 280

**IDP deployment**

control panel rollout, completing 607, 608

development and production,  
integrating 609, 610

GitLab Configuration, completing 601

GitOps, bootstrapping with  
OpenUnison 610-613

harbor configuration, completing 600

initial phase 597, 598

Pulumi, setting up 596

vault, unsealing 599

**id\_token** 178-180

**image layer** 7

**image layering** 6

**images**

scanning, for known exploits 350, 351

**impersonation** 197, 198

configuring, without OpenUnison 202

RBAC policies 202, 203

security considerations 199

testing 200, 201

used, for debugging 201, 202

using, to integrate authentication with  
cloud-managed clusters 196, 197

**implicit enablement** 489

**Inbound Impersonation** 203-205

**Infrastructure as Code (IaC) tool**

controller 594

Imperative Scripting 577

remote APIs 594

State Reconciliation 577

tooling 594

using, for deployment 576-578

**Ingress** 77

custom load balancer, adding 47, 48

**Ingress annotations**

used, for adding application to K8GB 164

**IngressClasses** 77

**Ingress controllers**

names, resolving 114, 115

used, for non-HTTP traffic 115

**ingress policies**

building 333, 334

deploying 333, 334

enforcing 330, 331

Gatekeeper cache, enabling 331

test data, mocking up 332

**Ingress rules**

creating 112-114

**inline mitigation**

versus post-attack mitigation 374-376

**interactive service** 535  
**internal developer platform (IDP)** 266  
    considerations, for building 585, 586  
**Internal Developer Portal (IDP)** 266, 589  
    deploying 595, 596  
**Internet Protocol (IP)** 75  
**IPAddressPool** 118  
**IP pool scoping** 126-128  
**island mode** 87  
**Istio**  
    components 482  
    deployment methods 486  
    downloading 486  
    exposing, in Kind cluster 489  
    installing 485  
    installing, with profile 486-488  
**Istio add-ons**  
    installing 504, 505  
**Istio advantages**  
    issues, finding 481  
    security 481, 482  
    traffic management 480  
    workload observability 480  
**istiod** 482  
    used, for creating control plane 482  
**Istio documentation**  
    reference link 488  
**istiod pod** 483  
    features 483, 484  
**istio-egressgateway** 485  
**istio-ingressgateway** 484, 485  
**Istio installation configuration, customizing**  
    reference link 487  
**Istio resources** 489  
    authorization policies 489-492  
    destination rules 498  
    gateways 495, 496  
    virtual services 497

## J

**Jaeger**  
    URL 513  
**JavaScript Object Notation (JSON)** 7, 66  
**JetStack** 246  
**Jobs** 77  
**JSON Web Tokens (JWTs)** 173, 482  
**jsPolicy** 338

## K

**K8GB CoreDNS servers**  
    managing, in sync 164-167  
**K8GB load balancing options** 156-158  
    failover 157  
    geoip 157  
    round robin 156  
    weighted round robin 156  
**Kaniko**  
    URL 617  
**karmor**  
    installing 390  
    logs 397-399  
    probe 390  
    profile 390-392  
    recommend command 392-397  
    uninstalling 390  
    using, to interact with KubeArmor 389, 390  
    vm 400  
**Keycloak** 187  
**Kiali** 504  
    Applications view, using 511-513  
    Graph view, using 507-511  
    homepage 506, 507  
    installing 505  
    integrating, with OpenUnison 533-535  
    Istio Config view 516-519  
    Services view, using 515, 516

- using, to observe mesh workloads 506
- Workloads view, using 513-515
- Kibana** 468, 469
  - used, for viewing data log 469-474
- KinD binary**
  - installing 36
- KinD cluster**
  - cluster config file, creating 38-40
  - creating 37
  - Istio, exposing 489
  - multi-node cluster configuration 40-42
  - reviewing 44
  - simple cluster, creating 37, 38
  - storage classes 45
  - storage driver 45
  - storage objects 44, 45
  - Storage Provisioner, using 46, 47
- KinD cluster configuration**
  - creating 48
- KinD, for OpenID Connect**
  - Active Directory groups, mapping to RBAC
    - RoleBindings 186
  - Active Directory, using with Kubernetes 185, 186
  - configuring 185
  - Dashboard, accessing 186
  - enterprise compliance requirements 187
  - implementing 187, 188
  - Kubernetes CLI access 186
  - LDAP, using with Kubernetes 185, 186
  - OIDC integration, verifying 192-194
  - OpenUnison, deploying 188-191
- KinD Kubernetes cluster**
  - working with 28-31
- Kindnet** 29
- Kopia** 414
- KubeArmor** 135, 370-374, 380
  - CI/CD pipeline integration 376
  - compliance, maintaining with standards 377
  - container security 374
  - deploying 378-380
  - deploying, in cluster 378
  - enhanced container visibility 377
  - inline mitigation versus post-attack mitigation 374-376
  - least privilege tenet adherence 377
  - Linux Security Module (LSM) 372, 373
  - multi-tenancy support 377
  - policies, creating and testing 382, 383
  - policy enforcement 377
  - policy impact testing 377
  - robust auditing and logging 377
  - zero-day vulnerability 376
- kubearmor-controller** 380
- KubeArmor logging**
  - enabling 380-382
  - events 380
- kubearmor-relay** 380
- KubeArmorSecurityPolicy**
  - creating 384-389
- KubeCon NA 2019 CTF**
  - reference link 303, 345
- kubectl**
  - commands 26
  - installing 35
  - tokens, using with 194-196
- kubelet** 63
- kubelet failure**
  - simulating 52-54
- Kubelet options**
  - customizing 42, 43
- kube\_pod\_info metric**
  - counter 444
  - gauge 444
  - histogram 445
  - summary 445
- kube-prometheus stack**
  - access, securing 462, 463

- Kubernetes** 172  
  backups 404  
  components and objects 26  
  enterprise 30  
  external users 172  
  groups 173  
  reasons, for eliminating Docker 3, 4  
  service accounts 173  
  validating admission policies, using 338-340
- Kubernetes 1.10** 31
- Kubernetes API**  
  configuring, to use OIDC 191, 192
- Kubernetes auditing** 230
- Kubernetes authorization page**  
  URL 221
- Kubernetes cluster** 58, 571
- Kubernetes components**  
  overview 58, 59
- Kubernetes Dashboard** 300  
  architecture 301  
  authentication methods 302  
  container architecture 302  
  logical architecture 301  
  reference link 188
- Kubernetes Dashboard 7.0** 310
- Kubernetes Dashboard with reverse proxy**  
  cluster-level applications 308  
  deploying 305, 306  
  local dashboards 306-308
- Kubernetes Global Balancer (K8GB)** 135, 153, 154  
  deploying, to cluster 156  
  features 154, 155  
  global load balancing, providing 164  
  installing, with Helm 160  
  requirements 155, 156  
  URL 153  
  used, for deploying highly available application 162
- Kubernetes Go SDK** 175
- Kubernetes in Docker (KinD)** 23, 27, 32  
  as nesting dolls 33  
  host, communicating via Ingress controller 34  
  installing 35  
  installing prerequisites 35  
  network flow 33  
  networking 32  
  selecting, reasons 27, 28  
  traffic flow 34
- Kubernetes, log management** 464  
  container logs 464  
  data log, viewing in Kibana 469-474  
  logs, tracing from container to console 466-468  
  OpenSearch 464, 465  
  OpenSearch, deploying 465, 466
- Kubernetes metrics** 440-447  
  managing 440
- Kubernetes resources** 67-70  
  apiservices 70  
  CertificateSigningRequests 71  
  ClusterRoleBindings 71  
  ClusterRoles 71  
  ComponentStatus 71  
  ConfigMaps 71-73  
  ControllerRevision 73  
  CronJobs 73, 74  
  CSI drivers 74  
  CSI nodes 74  
  CSIStrorageCapacities 74  
  CustomResourceDefinitions 74  
  DaemonSets 75  
  deployments 75  
  Endpoints 75  
  EndPointSlices 75  
  Events resource 76  
  FlowSchemas 76  
  HorizontalPodAutoscalers 76, 77  
  IngressClasses 77  
  Ingress resource 77

- Job 77  
LimitRanges 77  
LocalSubjectAccessReview 78  
manifests 67  
MutatingWebhookConfiguration 78  
NetworkPolicies 79  
nodes 79  
PersistentVolumeClaims 80  
PersistentVolumes 80  
Pod 80, 81  
PodDisruptionBudgets 80  
PodTemplates 81  
PriorityClasses 81  
PriorityLevelConfigurations 81  
ReplicaSets 82  
Replication controllers 82  
ResourceQuotas 82-84  
reviewing 70  
RoleBindings 84  
Roles 84  
RuntimeClasses 84  
Secrets 85  
SelfSubjectAccessReview 86  
SelfSubjectRulesReviews 86  
service accounts 86  
services 87, 88  
StatefulSets 88-90  
TokenReviews 91  
ValidatingWebhookConfiguration 91  
VolumeAttachments 92
- Kubernetes Secrets** 248  
**Kubernetes Secrets API**  
  using 266, 267  
**Kubernetes Secret Store CSI Driver**  
  URL 254  
**KubeVirt**  
  reference link 266, 400  
**Kverno** 338  
  reference link 338
- L**
- L2Advertisement resource** 118  
**layer 2** 118  
**layer 4 load balancers** 116  
  MetalLB, using as 117  
  options 116  
**layer 7 load balancers** 109  
  and name resolution 109, 111  
**least privileged access** 171  
**Let's Encrypt**  
  URL 593  
**Lightweight Directory Access Protocol (LDAP)** 180  
  using, with Kubernetes 185, 186  
**LimitRanges** 77  
**Linux Security Module (LSM)** 373  
  policies, creating and testing 382, 383  
**load balancers** 107  
  **LoadBalancer service**  
    creating 122, 123  
    creating, with ExternalDNS integration 145-147  
  **load balancing zone**  
    delegating 160-162  
**LocalSubjectAccessReview** 78  
**logs**  
  tracing, from container to console 466-468
- M**
- mandatory access controls (MACs)** 373  
**manifest** 67  
**mesh workloads**  
  observing, with Kiali 506  
**Metal as a Service (Maas)** 591  
**MetalLB** 117, 140  
  custom resources 118-120  
  installation link 118  
  using, as layer 4 load balancers 117

**MetalLB components** 120

controller 120, 121

speaker 121, 122

**MetalLB's BGP mode**

reference link 117

**microbot** 151**microservice building** 535, 536

access, authorizing 540-542

authentication, integrating 537-540

Hello World, deploying 536

other services, calling 549

simple impersonation, using 559

tokens, passing between services 559

user entitlements, authorizing 545

users, determining 542-544

**microservices**

managing, with Istio 528, 529

versus monolithic architecture 524

**microservices design** 526-528

benefits 528

**MinIO** 410

console 410, 411

deploying 410

reference link 410

**Moby**

versus Docker 4, 5

**Monitoring**

Applicatons 458

Integrating 621

Stack 462-464

**monolith deployment** 529

Kiali and OpenUnison integration 533-535

outside cluster 530-532

sticky sessions, configuring 532, 533

**monoliths** 528**monolithic application design** 525, 526**MTS (Microsoft Transaction Server)** 524**multi-node cluster configuration** 40-42**multi-platform image** 9

reference link 10

**multiple address pools**

using 125-127

**multiple clusters**

load balancing between 152-154

**multiple protocols**

using 129, 130

**multitenancy**

benefits, exploring 272, 273

**multi-tenant clusters** 82

policy deployment, scaling 360-365

**Multitenant Cluster, with self-service platform**

building 288, 289

requirements analyzing 289

**multitenant Kubernetes**

implementing, challenges 273-275

**multitenant platform**

components, replacing 622

deploying 292-297

designing 290-292

expanding 620

monitoring, integration with logging 621

policy management, integrating 621

requirements building 566-568

sources of identity 621

technology stack, selecting 568, 569

**mutating admission controllers** 314**mutating webhook** 334-337**MutatingWebhookConfiguration** 78**mutual Transport Layer Security**

(mTLS) 481, 482

**N****name resolution**

nip.io, using for 111, 112

**namespace** 78

restoring 430, 431

**namespace multi-tenancy**  
implementing 229, 230

**Negative Roles** 223

**Netshoot** 146

**network address translation (NAT)** 8

**Network Attached Storage (NAS)** 592

**Network File System (NFS)** 31

**network policy** 79, 130  
creating 132-135  
creating, tools 135

**network policy object**  
overview 131  
podSelector 131  
PolicyTypes 131, 132

**nginx-ingress.sh** 114

**NIST-800-53** 249

**node image** 28, 32

**nodes**  
customizing 592  
customizing, drawbacks 592  
services, accessing 593

**node security** 344  
enforcing, with Gatekeeper 351  
enforcing, with Pod Security Standards 365, 366

**node security policies**  
authorizing 353, 354  
cluster policies, enforcing 355, 356  
constraint violations, debugging 356-360  
debugging 354  
deploying 354  
policy deployment, scaling in multi-tenant clusters 360-365  
security context defaults, generating 355

**non-HTTP traffic**  
Ingress controllers, using 115

**Non-Volatile Memory Express (NVMe)** 90

**O**

**OAuth2** 174

**OAuth2 Token Exchange**  
check-writing service, deploying 552, 553  
check-writing service, running 552, 553  
delegation, using 557-559  
Impersonation, using 554-556  
service authentication 551, 552  
using 549, 550

**object** 315

**object identifiers** 314

**OCSP protocol** 180

**OIDC Identity Provider (IdP)** 175

**OIDC integration**  
verifying 192-194

**OIDC login process**  
tokens, generated from 175

**Okta** 187

**OPA authorization rule**  
creating 547-549

**Open Container Initiative (OCI)** 3

**OpenFGA** 546

**OpenID Connect (OIDC)** 42, 173, 174  
benefits 174  
Kubernetes API, configuring to use 191, 192

**OpenID Connect protocol** 171-176

**OpenID Connect Tokens**  
reference link 177

**Open Policy Agent (OPA)** 179, 313-315, 344, 545  
architecture 315-317  
automated testing framework 319  
Gatekeeper 318  
policy language 317

**OpenSearch** 464, 465  
deploying 465, 466

**OpenShift Container Platform (OCP)** 565

**Open Systems Interconnection (OSI) model** 107, 108

**OpenUnison** 187, 232, 241, 569  
access, securing to metrics endpoint 461, 462  
deploying 188-191  
GitOps, bootstrapping with 610-613  
integrating, with Kiali 533-535  
metrics, adding 459-461  
metrics, need for 458  
monitoring 458  
reference link 210  
used, integrating dashboard 308-310

**opinionated platforms** 565, 566

**orchestration** 290, 403, 479

**out of memory (OOM) event** 503

## P

**peer authentication** 499, 500

**persistent data** 8

**Persistent Volume Claims (PVCs)** 80, 410

**persistent volume (PV)** 80

**phishing** 310

**pipelines**  
authenticating from 207  
designing 564, 565  
securing 566

**pipelines authentication**  
anti-patterns, avoiding 215  
certificates, using 210-212  
identity, using 212-215  
tokens, using 208-210

**PKCS11** 180

**platform architecture**  
designing 570, 571  
images, pushing and pulling securely 576  
remote Kubernetes cluster,  
managing securely 571-576

**Platform as a Service (PaaS)** 274

**PodDisruptionBudget (PDB)** 80

**Podman** 32

**Pod resource** 80

**Pod Security Admission (PSA)** 352  
versus Gatekeeper 352, 353

**Pod Security Policies (PSP)** 351, 352  
versus Gatekeeper 352, 353  
versus Pod Security Admission (PSA) 352, 353

**Pod Security Standards** 352  
reference link 365  
using, to enforce node security 365, 366

**PodTemplates** 81

**policies**  
creating, without Rego 337  
debugging, with audit2rbac 235-239

**policy file** 231

**Policy Spec for Containers**  
reference link 384

**post-attack mitigation**  
versus inline mitigation 374-376

**primary DNS server**  
configuring 150

**PriorityClasses** 81

**PriorityLevelConfigurations** 81

**private key** 11

**Privilege Access Manager (PAM)** 205

**Privileged User**  
impersonating 206

**Privileged User Account**  
using 205

**profile**  
used, for installing Istio 486-488

**Prometheus** 442, 443  
metrics collection 444, 445  
querying, with PromQL 447-450

**Prometheus Stack**  
deploying 441

- PromQL** 447  
used, for querying Prometheus 447-450
- Proof Key for Code Exchange (PKCE) protocol** 176
- protocol transition** 560
- public key** 11
- Pulumi** 292  
deploying 594  
installation link 595  
URL 577
- R**
- recursive query** 148
- Rego** 317  
debugging 329, 330  
dynamic policies, building 326-329  
existing policies, using 330  
OPA policy, developing 320  
OPA policy, testing 321, 322  
policies, deploying to Gatekeeper 323-325  
using, to write policies 319, 320
- ReplicaSets** 82
- replication controllers** 82
- RequestAuthentication policy** 500, 501
- RequestAuthorization policy** 500
- ResourceQuotas** 82- 84
- Restic** 414
- reverse proxy**  
configuring 306
- Role-Based Access Control (RBAC)** 220  
versus entitlement-based access control 220
- RoleBindings** 84, 225, 226  
ClusterRoles, combining with 227
- Roles** 84, 220  
identifying 221, 222  
Negative Roles 223  
resources 220  
verbs 220
- versus ClusterRoles 222
- Rollout** 565
- RuntimeClasses** 84
- runtime security** 370, 371
- S**
- Sealed Secrets** 248, 249
- Secrets** 85, 241  
integrating, into deployments 256, 257  
managing, in Enterprise 244  
protecting, in application 247  
storing, as Secret Objects 248, 249  
versus Configuration Data 242-244
- Secrets at rest**  
threats 245, 246
- Secrets, integration into deployments**  
Environment Variables 263  
Kubernetes Secrets API, using 266, 267  
Vault API, using 267  
volume mounts 257  
workload, consuming secret data 257
- Secrets in transit** 246
- Secrets Managers** 248
- Secrets, storing as Secret Objects** 248, 249  
external Secrets Managers 250-253  
hybrid of external secrets management and  
secret objects, using 254-256  
Sealed Secrets 249, 250
- Security Context Constraints (SCCs)** 351
- Security Information and Event Manager (SIEM)** 308
- SelfSubjectAccessReviews** 86
- SelfSubjectRulesReviews** 86
- Server Name Indication(SNI)** 115
- service accounts** 86, 87, 172, 181-183
- service entries** 501, 502
- service mesh** 479  
application, deploying into 505, 506

- service names**  
making, available externally 140, 141
- Services** 87, 88  
creating 98-100  
resolving, with DNS 100, 101  
working 96, 97
- services running**  
accessing, in containers 8
- service types** 101  
ClusterIP service 101, 102  
ExternalName service 105, 106  
LoadBalancer service 105  
NodePort service 102-104
- sidecars** 260, 502, 503
- signatures** 246
- simple storage service (S3)** 31
- Single Sign-On (SSO)** 569
- Smurf attack** 128
- socket** 116
- speaker** 120, 121  
responsibilities 121
- Special Interest Group (SIG)** 27
- StatefulSets** 88, 89
- static pod** 192
- storage classes** 90
- storage driver** 6
- SubjectAccessReviews** 91
- subjects** 225
- submitter identifiers** 314
- sync**  
K8GB CoreDNS servers, managing 164-167
- system assigned groups** 173
- T**
- Talos Linux from Sidero**  
reference link 592
- Tanzu Application Platform (TAP)** 407
- Tape Archive** 345
- tarball** 345
- TektonCD dashboard** 308
- tenant, IDP**  
onboarding 613-616  
users, adding to 620
- tenant onboarding**  
automating 579-582
- tenants**  
vClusters, using for 275, 276
- Terraform** 292
- ThinApp** 2
- threat modeling** 244
- TLS passthrough** 115
- token**  
using, to log in 304  
using, with kubectl 194-196
- TokenRequest API** 183, 185
- TokenReviews** 91
- Topaz** 546
- traffic management** 480  
blue/green deployments 480  
canary deployments 481
- Trivy** 348
- U**
- Ubuntu**  
Docker, installing on 10, 11
- Ubuntu 22.04** 28
- user-asserted groups** 173
- user entitlements authorization, microservice** 545
- OPA authorization rule, creating 547-549  
OPA, using with Istio 545, 546  
service authorization 545

**user experience (UX)** 3, 280

**UTF-8** 243

**UTF-16** 243

## V

**validating admission controllers** 314

**validating admission policies**

  using, in Kubernetes 338-340

**validating admission policies, components**

  ValidatingAdmissionPolicy 338

  ValidatingAdmissionPolicyBinding 338

**ValidatingWebhookConfigurations** 91

**Vault**

  sidecar injector, using 260-262

**Vault API**

  using 267

**vCluster Identity** 284

**vClusters** 569

  accessing, securely 280-283

  components 275

  deploying 277-280

  External Services, accessing from 283-286

  issues 285

  upgrading 288

  using, for tenants 275, 276

**Velero** 403, 407

  backup flags 423

  backup storage location 409

  cluster backup, scheduling 421-423

  commands 426-428

  custom backup, creating 423

  installing 412-414

  installing, in new cluster 434

  limitations, of backing up data 416

  managing, with CLI 424

  one-time cluster backup, running 416-421

  requirements 408

  storage provider plug-ins 408

  system requirements 408

  used, for restoring from backup 428, 429

**Velero CLI** 408

  installing 408

**Velero objects**

  creating 426

  deleting 426

  details, retrieving for 426

  listing 425

**Velero server** 408

**Velero, to back up PVCs** 414

  opt-in approach 415

  opt-out approach 415

**virtual IP (VIP) address** 111

**virtual machine (VM)** 344

  versus containers 344, 345

**virtual services** 497

**volume mounts** 257

  Kubernetes Secrets, using 258, 259

  Vault's Sidecar Injector, using 260-262

## W

**WasmPlugins** 503, 504

**web application filters (WAFs)** 124

**web frontend** 101

**worker node components** 63

  container runtime 63

  kubelet 63

  kube-proxy 63

**workloads**

  exposing, to requests 96

## Y

**YAML Ain't Markup Language (YAML)** 66

## Z

**zero trust** 347

# Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily.

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below:



<https://packt.link/free-ebook/9781835086957>

2. Submit your proof of purchase.
3. That's it! We'll send your free PDF and other benefits to your email directly.

