Hands-On Kubernetes, Service Mesh and Zero-Trust

Build and manage secure applications using Kubernetes and Istio



Swapnil Dubey
Mandar J. Kulkarni

bpb

Hands-On Kubernetes, Service Mesh and Zero-Trust

Build and manage secure applications using Kubernetes and Istio

Swapnil Dubey

Mandar J. Kulkarni



www.bpbonline.com

Copyright © 2023 BPB Online

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 BPB Online 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.

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

First published: 2023

Published by BPB Online
WeWork
119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-675

www.bpbonline.com

Dedicated to

To my 'partners in Crime'
(since childhood) : ***Sneha, Shivam & Shubhanshu***
– Swapnil Dubey

****

My beloved wife:
Tejashri
&
My Daughters ***Rucha*** *and* ***Shreya***
– Mandar J. Kulkarni

About the Authors

- **Swapnil Dubey** has been working as an Architect at SLB since 2019, with a IT total experience of more that 14 years with enterpireses like Snapdeal, Pubmatic and Schlumberger. His current role at SLB involves designing and guiding technical teams implement data intensive workloads using Microservices and distributing computing architectural patterns hosted on public cloud (GCP & Azure) and On premise.

  In the past, he has served as Trainers for BigData technologies like Hadoop and Spark (Certified Trainer with Cloudera), and facilitated approximately 20 batches of people to kickstart their journey of Distributed computing. Moreover, he has spoken in multiple national & international conferences, where the key topic to talk was about containers and their management using Kubernetes.

  He completed his Masters from BITS Pilani in Data Analytics, and also holds Professional Architect Certifications in GCP and Microsoft Azure. This is his second book. Before this one, he has authored a book Scaling Google Cloud Platform with BPB Publications.

- **Mandar J. Kulkarni** has been working in software development and design for more than 16 years, and has played multiple roles such as Software Engineer, Senior Software Engineer, Technical Leader, Project manager and Software Architect. Currently, he is an architect in SLB building data products on top of Open Subsurface Data Universe (OSDU) Data Platform. He has also contributed to OSDU Data Platform with multiple architectural modifications and improvements.

  He has acquired Professional Cloud Architect certification from Google Cloud and also holds a Masters degree from BITS Pilani in Software Engineering. He has been a technical blogger for a while and this is first foray into writing a complete book.

About the Reviewer

Mahesh Chandrashekhar Erande has played the software architect role in the healthcare, telecom, and energy domains. For the past 19 years, he did end-to-end solution designing, programming and operationally supporting scalable enterprise apps. He is currently constructing the poly-cloud products for the SLB.

Acknowledgements

- Any accomplishment requires the effort of many people, and this work is no different. First and foremost, I would like to thank my family, (especially my father figure, mentor and guardian , Mr. N.R. Tiwari and My Mother – Sushma & Wife - Vartika) for continuously encouraging and supporting me in writing the book. I could have never completed this book without their support. Big thanks to the Energy which keeps pushing me everyday for my side hustles (apart from work).

  I gratefully acknowledge Mr. Mahesh Erande for his kind technical scrutiny of this book. My sincere thanks to the co author of the book, Mr. Mandar J. Kulkarni, whose constant enthusiasm and quality inspired me to bring out my best.

  My gratitude also goes to the team at BPB Publication for being supportive and patient during the editorial review of the book. A big thank you to SLB team for allowing me do this work.

  - Swapnil Dubey

- This book would not have been possible without continuous support from my family and friends. I thank them for their unconditional support and encouragement throughout this book's writing, especially my wife Tejashri and my brother Kedar.

  I am also grateful to the BPB Publications team for giving me the opportunity to author the book, and also for their support, guidance and expertise in making this book a reality. The participation and collaboration of reviewers, technical experts, and editors from team BPB has been very valuable for me as well as the book.

  Collaborating with author Mr. Swapnil Dubey has been an invaluable experience, and the learnings I gained, will guide me forever. I also want to thank Mr. Mahesh Erande for his technical reviews and feedback on the book content.

I would also like to acknowledge SLB for giving me the opportunities to work on the interesting technologies during my career and also for allowing me to write the book.

Finally, I would like to thank all the readers who keep taking interest in reading technical books. The appreciation and feedback from the readers is the biggest motivation for authors to create better content.

- Mandar J. Kulkarni

Preface

The objective of this book is to streamline the creating and operating workloads on Kubernetes. This book will guide and train software teams to run Kubernetes clusters directly (with or without EKS/GKS), use API gateways in production, and utilise Istio Service mesh, thereby having smooth, agile, and error-free delivery of business applications.

The reader masters the use of service mesh and Kubernetes, by delving into complexities and getting used to the best practices of these tools/approaches. While one runs hundreds of microservices and Kubernetes clusters, security is highly prone to be breached and that is where zero trust architecture would be kept in mind throughout the software development cycle.

The book also makes use of some of the great observability tools to provide a robust, yet clean set of monitoring metrics such as Latency, traffic, errors, and saturation to get a single performance dashboard for all microservices. After reading this book, challenges around application deployment in production, application reliability, application security and observability will be better understood, managed, and handled by the audience.

Chapter 1: Docker and Kubernetes 101 - This chapter will introduce the audience to the basics of Dockers and Kubernetes. In the docker section, the audience will get concepts to write and push images to container registries. We will give a walk through of an already developed application and package it in a docker container. There will be a discussion around practices which induce security vulnerabilities and their resolution. In the later part of the chapter, the audience will get introduced to Kubernetes, such as the why, what, and how of Kubernetes, followed by an in-depth understanding of architecture. There will be discussion around basic principles of Immutability, declarative and Self-healing way of assigning infrastructure in Kubernetes cluster.

Chapter 2: PODs – discusses the foundational block of Kubernetes called Pod. The chapter discusses the lifecycle of the pods along with health checks. The chapter also explains the resources requirements for Pod such as CPU, Memory as well as storage required for persisting data, along with security aspects like pod security standards and admissions.

Chapter 3: HTTP Load Balancing with Ingress - This chapter will discuss concepts of bringing the data in and out of an application deployed in Kubernetes. Ingress is a Kubernetes-native way to implement the "virtual hosting" pattern. This chapter will talk about exposing services deployed in Kubernetes to the outside world. AI gateways will also be discussed in this chapter taking example of open source API gateways like Gloo,Tyk and Kong. Apart from discussing the details around networking, readers will get the feel of security issues and loopholes which should be taken care of while configuring networking.

Chapter 4: Kubernetes Workload Resources – takes readers towards more practical examples of using Kubernetes in enterprise applications, by showing hands-on examples of creating workload resources such as deployments, replicasets, jobs and daemon sets. The chapter discusses the life cycle of each of these workload resources and explains which workload resource should be used for which use case while building scalable applications.

Chapter 5: ConfigMap, Secrets, and Labels - In this chapter, the concept of labels and secrets will be discussed. Labels can be used to select objects and to find collections of objects that satisfy certain conditions. In contrast, annotations are not used to identify and select objects. This chapter will help the audience to in-depth understanding of Annotations & Labels and strategies around how to use them effectively in real environments. This chapter will also help you understand the concepts of config map and a Secret better.

Chapter 6: Configuring Storage with Kubernetes – focuses on storage patterns with Kubernetes. The chapter discusses Volumes, Persistent volumes and stateful sets in details followed by a practical example of MongoDB installation. Furthermore, the chapter discusses disaster recovery of content stored using configured storage and the extesibility of Kubernetes architecture using container storage interface.

Chapter 7: Introduction to Service Discovery - Service discovery tools help solve the problem of finding which processes are listening at which addresses for which services. This chapter audience will get insight about various ways of discovering service in Kubernetes cluster. This chapter will act as a building block for section 3, where conceptual discussion will happen around how to achieve service discovery using Istio. The audience will also get insights into the various patterns of discovery and registration and the same will be showcased as hands-on exercises in the chapter.

Chapter 8: Zero Trust Using Kubernetes - This chapter will introduce the audience to the aspects of modelling and application with Zero trust principles in place. Lot of security aspects are already discussed in the previous chapters. For example, in Chapter 3, HTTP Load Balancing With Ingress, we will be talking about POD security. Similarly in Chapter 4, Kubernetes Worklad Resources, we plan to talk about security aspects when it comes to creation of networks. This chapter will give the audience a hands-on insight of how to achieve the aspects of this zero-trust security model using the individual building blocks discussed in the previous chapters.

Chapter 9: Monitoring, Logging and Observability - This chapter will talk about aspects of logging and monitoring of applications deployed in the Kubernetes cluster. This chapter will further discuss ways to implement basic SRE concepts and how the observability aspects are supported. Hands on exercises will demonstrate each of the concepts of logging, monitoring and SRE by enhancing the micro service application written and developed in earlier chapters.

Chapter 10: Effective Scaling - One of the key advantages of using Microservice deployed on Kubernetes is the power scaling mechanism. This chapter will help the audience understand the aspects of scaling in Kubernetes which includes horizontal & vertical pod scaling. Not only can we configure auto scaling on out of the box metrics, but also based on custom metric and combination of metrics. All the hands-on aspects will involve the three micro services which we created in earlier chapters. One Micro service will be planned to scale horizontally and vertically. Others will scale based on custom metrics, and third will showcase scaling based on a combination of two metrics.

Chapter 11: Introduction to Service Mesh and Istio – starts with the basics about microservices and then talks in details about the what, why and how of the service mesh concepts. The chapter discusses pros and cons of the service mesh as a concept and uses Isio as an example. The chapter then discusses Istio architecture, installation techniques and the customizations of Istio steup.

Chapter 12: Traffic Management Using Istio – is all about how to take the traffic management logic out of service code into the declarative yamls. The chapter discusses controlling ingress traffic, egress traffic and gateways. The chapter introduces Kubernetes's custom resources like VirtualService, DestinationRule, ServiceEntry and how to make use of them for achieving traffic management strategies like canary deployment, blue-green deployment. The chapter also

explains with examples how to implement design patterns like circuit breaking, timeouts, retries and fault injection using service mesh like Istio. This chapter introduces and uses a sample application to explain the traffic management patterns.

Chapter 13: Observability Using Istio – talks about how different open source observability tools like Kiali, Grafana, Prometheus, Jaeger can be used alongside Istio to improve the observability. The sample application introduced in earlier chapters is used here again to show how to manage traffic patterns between different microservices, how to observe the scalability, how to monitor and search the logs, and how and where to view and search different metrics. The chapter also explains with examples how to use distributed tracing to debug latency issues in the application.

Chapter 14: Securing Your Services Using Istio – revolves around identity management, authorization and authentication using the built-in support that Istio provides. The chapter briefly introduces what is secure communication and then explains how Istio helps with Certificate management to make the intra-cluster communication secure by default. The chapter builds on top of the existing sample application used in previous chapters to explain concepts like permissive mode of Istio, Secure naming, Peer authentication, Service authorization, End-user authorization and so on. The chapter concludes by bringing it all together by explaining security architecture of Istio.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/l14igmh

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Hands-On-Kubernetes-Service-Mesh-and-Zero-Trust**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

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

business@bpbonline.com for more details.

At **www.bpbonline.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 BPB books and eBooks.

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 **business@bpbonline.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 **www.bpbonline.com**. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights 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.

Reviews

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

For more information about BPB, please visit **www.bpbonline.com**.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Table of Contents

1. Docker and Kubernetes 101 1

Introduction 1

Structure 2

Objectives 2

Introduction to Docker 2

Introduction to Kubernetes 8

Kubernetes architecture 10

Kubernetes Master 11

Kubernetes Worker 14

Principles of immutability, declarative and self-healing 16

Principle of immutability 16

Declarative configurations 16

Self-healing systems 17

Installing Kubernetes 17

Installing Kubernetes locally using Minikube 18

Installing Kubernetes in Docker 19

Kubernetes client 19

Checking the version 20

Checking the status of Kubernetes Master Daemons 20

Listing all worker nodes and describing the worker node 21

Strategies to validate cluster quality 23

Cost-efficiency as measure of quality 23

Right nodes 24

Request and restrict specifications for pod CPU and memory resources 24

Persistent volumes 24

Data transfer costs and network costs 24

Security as a measure of quality 25

Conclusion 25
Points to remember 25
Multiple choice questions 26
Answers *26*

2. PODs 27
Introduction 27
Structure 28
Objectives 28
Concept of Pods 29
CRUD operations on Pods 30
Creating and running Pods *30*
Listing Pods *31*
Deleting Pods *33*
Accessing PODs 34
Accessing via port forwarding *34*
Running commands inside PODs using exec *35*
Accessing logs *36*
Managing resources 36
Resource requests: Minimum and maximum limits to PODs *36*
Data persistence 38
Internal: Using data volumes with PODs *39*
External: Data on remote disks *41*
Health checks 42
Startup probe *42*
Liveness probe *43*
Readiness probe *43*
POD security 44
Pod Security Standards *45*
Pod Security Admissions *46*

Conclusion 47
Points to remember 47
Questions 47
Answers *48*

3. HTTP Load Balancing with Ingress 49
Introduction 49
Structure 49
Objectives 50
Networking 101 50
Configuring Kubeproxy *53*
Configuring container network interfaces *54*
Ingress specifications and Ingress controller 55
Effective Ingress usage 62
Utilizing hostnames *62*
Utilizing paths *63*
Advanced Ingress 64
Running and managing multiple Ingress controllers *64*
Ingress and namespaces *64*
Path rewriting *64*
Serving TLS *65*
Alternate implementations 66
API gateways 68
Need for API gateways *68*
Routing requests *69*
Cross-cutting concerns *69*
Translating different protocols *69*
Securing network 69
Securing via network policies *69*
Securing via third-party tool *70*

Best practices for securing a network 71
Conclusion 72
Points to remember 72
Multiple choice questions 73
Answers *73*
Questions 73

4. Kubernetes Workload Resources 75
Introduction 75
Structure 76
Objectives *77*
ReplicaSets *77*
Designing ReplicaSets *77*
Creating ReplicaSets *78*
Inspecting ReplicaSets *79*
Scaling ReplicaSets *79*
Deleting ReplicaSets *81*
Deployments 81
Creating deployments *82*
Managing deployments *83*
Updating deployments *83*
Deployment strategies *86*
Monitoring deployment status *86*
Deleting deployments *87*
DaemonSets 87
Creating DaemonSets *87*
Restricting DaemonSets to specific nodes *89*
Updating DaemonSets *90*
Deleting DaemonSets *91*
Kubernetes Jobs 92

Jobs *92*
Job patterns *94*
Pod and container failures *94*
Cleaning up finished jobs automatically *94*
CronJobs *95*
Conclusion 96
Points to remember 97
Questions 98
Answers *98*

5. ConfigMap, Secrets, and Labels 99
Introduction 99
Structure 100
Objectives 100
ConfigMap 100
Creating ConfigMap *102*
Consuming ConfigMaps *104*
Consume ConfigMap in the environment variables *105*
Set command-line arguments with ConfigMap *106*
Consuming ConfigMap via volume plugin *107*
Secrets 109
Creating Secrets *109*
Consuming Secrets *111*
Consuming Secrets mounted as volume *111*
Consuming Secrets as environment variables *112*
Private docker registries *112*
Managing ConfigMaps and Secrets 113
Listing *113*
Creating *114*
Updating *114*

Applying and modifying labels 115
Labels selectors 117
Equality-based selector 117
Set-based selectors 118
Role of labels in Kubernetes architecture 118
Defining annotations 119
Conclusion 120
Points to remember 120
Questions 120
Answers 121

6. Configuring Storage with Kubernetes 123
Introduction 123
Structure 124
Objectives 124
Storage provisioning in Kubernetes 124
Volumes 124
Persistent Volumes and Persistent Volume claims 125
Storage class 130
Using StorageClass for dynamic provisioning 132
StatefulSets 133
Properties of StatefulSets 133
Volume claim templates 137
Headless service 137
Installing MongoDB on Kubernetes using StatefulSets 138
Disaster recovery 140
Container storage interface 141
Conclusion 142
Points to remember 142
Questions 143
Answers 143

7. Introduction to Service Discovery 145
Introduction 145
Structure 145
Objectives 146
What is service discovery? 146
Client-side discovery pattern 148
Server-side discovery pattern 150
Service registry 151
Registration patterns 151
Self-registration pattern 152
Third-party registration 152
Service discovery in Kubernetes 153
Service discovery using etcd 153
Service discovery in Kubernetes via Kubeproxy and DNS 157
Service objects 159
DNS 160
Readiness checks 160
Advance details 161
Endpoints 161
Manual service discovery 163
Cluster IP environment variables 164
Kubeproxy and cluster IPs 164
Conclusion 165
Points to remember 166
Questions 166
Answers 167

8. Zero Trust Using Kubernetes 169
Introduction 169
Structure 170

Objectives 170
Kubernetes security challenges 171
Role-based access control (RBAC) 173
Identity *173*
Role and role bindings 174
Managing RBAC *177*
Aggregating cluster roles *178*
User groups for bindings *179*
Introduction to Zero Trust Architecture 180
Recommendations for Kubernetes Pod security *182*
Recommendations for Kubernetes network security *185*
Recommendations for authentication and authorization *186*
Recommendations for auditing and threat detection *187*
Recommendation for application security practices *187*
Zero trust in Kubernetes 188
Identity-based service to service accesses and communication *188*
Include secret and certificate management and hardened Kubernetes encryption *189*
Enable observability with audits and logging *190*
Conclusion 191
Points to remember 192
Questions 192
Answers *193*

9. Monitoring, Logging and Observability 195
Introduction 195
Structure 196
Objectives 196
Kubernetes observability deep dive 197
Selecting metrics for SLIs *199*
Setting SLO *200*

Tracking error budgets *200*
Creating alerts *201*
Probes and uptime checks *202*
Pillars of Kubernetes observability 204
Challenges in observability 205
Exploring metrics using Prometheus and Grafana 206
Installing Prometheus and Grafana *208*
Pushing custom metrics to Prometheus *211*
Creating dashboard on the metrics using Grafana *213*
Logging and tracing 214
Logging using Fluentd *215*
Tracing with Open Telemetry using Jaeger *217*
Defining a typical SRE process 220
Responsibilities of SRE 221
Incident management *222*
Playbook maintenance *223*
Drills *223*
Selecting monitoring, metrics and visualization tools 224
Conclusion 225
Points to remember 225
Questions 226
Answers *226*

10. Effective Scaling 227
Introduction 227
Structure 228
Objectives 228
Needs of scaling microservices individually 228
Principles of scaling 229
Challenges of scaling 230

Introduction to auto scaling 231
Types of scaling in K8s 232
Horizontal pod scaling *233*
Metric threshold definition *237*
Limitations of HPA *239*
Vertical pod scaling *239*
Cluster autoscaling *242*
Standard metric scaling *244*
Custom Metric scaling *247*
Best practices of scaling 249
Conclusion 250
Points to remember 250
Questions 251
Answers *251*

11. Introduction to Service Mesh and Istio 253
Introduction 253
Structure 254
Objectives 254
Why do you need a Service Mesh? 254
Service discovery *256*
Load balancing the traffic *256*
Monitoring the traffic between services *256*
Collecting metrics *256*
Recovering from failure *256*
What is a Service Mesh? 257
What is Istio? 260
Istio architecture 261
Data plane *262*
Control plane *263*

Installing Istio 263
Installation using istioctl *264*
Cost of using a Service Mesh 267
Data plane performance and resource consumption *267*
Control plane performance and resource consumption *267*
Customizing the Istio setup 268
Conclusion 269
Points to remember 270
Questions 270
Answers *270*

12. Traffic Management Using Istio 273
Introduction 273
Structure 274
Objectives 274
Traffic management via gateways 274
Virtual service and destination rule *276*
Controlling Ingress and Egress traffic 279
Shifting traffic between versions 280
Injecting faults for testing 284
Timeouts and retries 286
Circuit breaking 288
Conclusion 290
Points to remember 291
Questions 291
Answers *291*

13. Observability Using Istio 293
Introduction 293
Structure 293
Objectives 294

Understanding the telemetry flow 294
Sample application and proxy logs 295
Visualizing Service Mesh with Kiali 297
Querying Istio Metrics with Prometheus 303
Monitoring dashboards with Grafana 305
Distributed tracing 308
Conclusion 313
Points to remember 314
Questions 314
Answers 315

14. Securing Your Services Using Istio 317
Introduction 317
Structure 318
Objectives 318
Identity Management with Istio 318
Identity verification in TLS 319
Certificate generation process in Istio 319
Authentication with Istio 321
Mutual TLS authentication 321
Secure naming 322
Peer authentication with a sample application 323
Authorization with Istio 327
Service authorization 328
End user authorization 332
Security architecture of Istio 336
Conclusion 337
Points to remember 338
Questions 338
Answers 338

Index 341-347

Chapter 1
Docker and Kubernetes 101

Introduction

Software architecture evolves with time to cater to the needs of the latest industry workloads. For example, a few years ago, when the data size was insignificant, we used to write data processing workloads using multithreading, but the processing spanned across multiple machines. After that came the wave of Big data, where distributed computing frameworks like Hadoop and Spark were used to process huge volumes of data. We are now witnessing a similar wave. Today's architects believe in breaking a use case into more minor services and orchestrating user journeys by orchestrating the calls to the microservices. Thanks to Google for donating Kubernetes to the open-source world, such an architecture is now reality. With many organizations adopting Kubernetes for their infrastructure management, it has become the platform for orchestrating and managing container-based distributed applications, both in the cloud and on-premises. No matter what role you play in the organization, be it a developer, architect, or decision maker, it is imperative to understand the challenges and features of Kubernetes to design effective workflows for the organization.

Just like Kubernetes, Docker is one of the most widely used container runtime environments. Docker has seen its growth over the last few years, and while everybody agreed to the need for a container, there have always been debates about how to manage the life cycle of a container. The way Kubernetes and docker complement

each other's needs makes them prominent partners for solving container-based workloads. Docker, the default container runtime engine, makes it easy to package an executable and push it to a remote registry from where others can later pull it.

In this chapter, you will dive deep into the concepts of Docker and Kubernetes. In the later chapters, one component discussed at a high level will be picked and discussed in further detail.

Structure

In this chapter, we will discuss the following topics:

- Introduction to Docker
- Introduction to Kubernetes
    - Kubernetes architecture
    - Principles of immutability, declarative and self-healing
- Installing Kubernetes
    - Installing Kubernetes locally
    - Installing Kubernetes in Docker
- Kubernetes client
- Strategies to validate cluster quality
    - Cost efficient
    - Security

Objectives

After studying this chapter, you should understand the basic working of Docker and Kubernetes. This chapter will also discuss some generic best practices when deploying docker and Kubernetes, and it will help you understand what factors you should keep in mind while enhancing reliability, resiliency, and efficiency better suited to the type of use cases you intend to solve. You will understand the principles of immutability, declarative, and self-healing, based on which the framework of Kubernetes stands. This chapter will also help you learn how to evaluate the quality of your cluster as per the needs of use cases.

Introduction to Docker

Docker is the most widely used container runtime environment; it enables creating containers and running them on some infrastructure. Some infrastructure could be

physical on-premise nodes or virtual machines on any cloud platform. Developing, shipping and running applications are key terms when discussing docker. You can develop applications with each application having its binaries and libraries, and package them by creating an image. These images could be instantiated by running them as containers. Each container with separate applications can run on the same physical or virtual machine without impacting the other.

Consider *Figure 1.1*, which demonstrates the preceding discussion in detail.

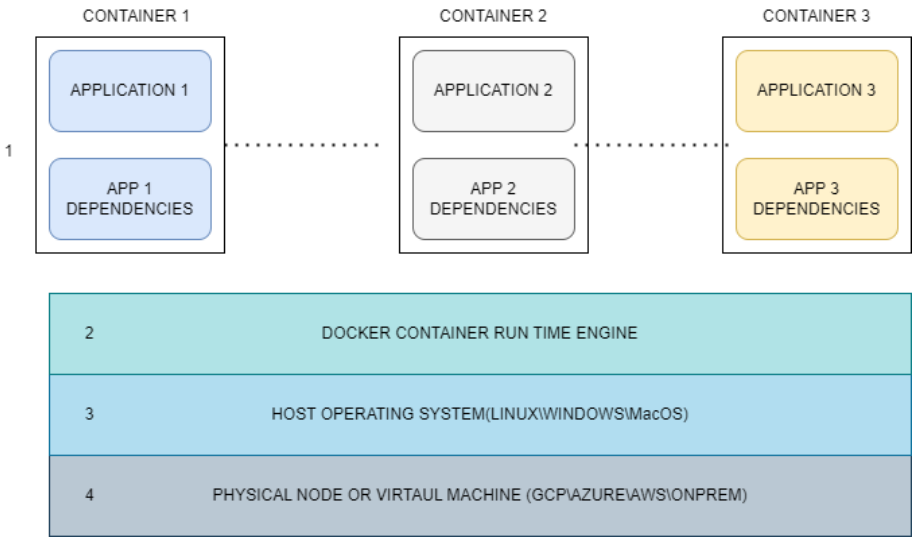


Figure 1.1: *Docker 101*

Refer to the numerical labelling in *Figure 1.1* with the following corresponding numerical explanations:

1. Multiple applications with completely different technology stacks could be developed, and their complete dependencies could be packaged as container images. These container images, when instantiated, become containers.

2. These container images need a container runtime environment. The container runtime environment provides all the features to launch , execute, and delete an image. Multiple runtime environments are available in the industry, such as runC, containerd, Docker, and Windows Containers.

3. These container runtime environments run on top of any operating system. For example, a docker runtime could be installed on Linux, Windows, or macOS, unless the container runtime is installed successfully and no other host operating system restrictions apply.

4. The mentioned setup can run on a physical or virtual machine, and on-premise machines on public cloud providers like GCP, AWS, or Azure. In fact,

the setup can run on a device in which an operating system and **Container Runtime Environment** (**CRE**) could be installed.

With this basic introduction to how containers, container run time, and physical infrastructure align, let's now look at Docker precisely and understand the game's rules with Docker as a container runtime environment. *Figure 1.2* demonstrates the docker process of building and using images:

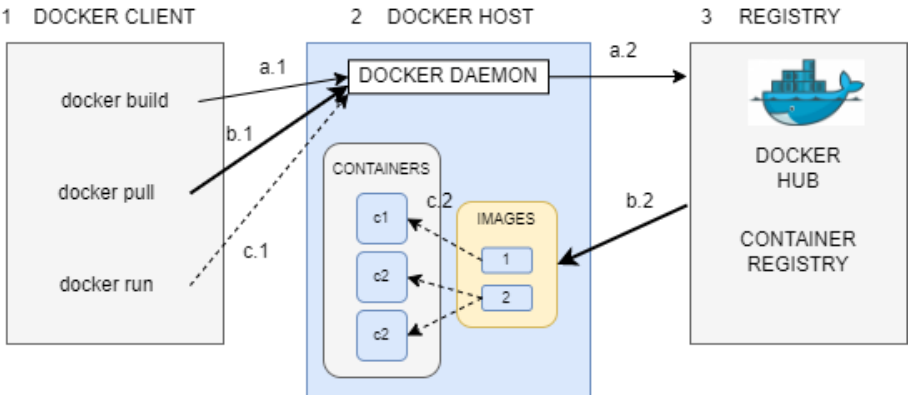


***Figure 1.2**: Docker Process*

Refer to the numerical labelling in *Figure 1.2* with the following corresponding numerical explanations:

1. Docker client is a CLI utility to execute docker commands. In this section, there are three main commands that everybody should know about.
2. The docker daemon interprets the CLI commands, and the action is performed.
3. A registry is a repo where you build and upload the image. A few standard registries are docker hub, quay.io, and registries with cloud providers, such as Google Container registry in the Google cloud platform.

Let us talk about the three docker commands shown in *Figure 1.2*:

- **`Docker builds <docker-file-path>`**: You specify the contents of your repo in a plaintext file (which are written as per the construct suggested by Docker). This command creates a local image using the plaintext file(created above). Look at the arrows labeled with **`a`**.
    - **`a.1`**: Docker build command is interpreted by the docker daemon.
    - **`a.2`**: The image created by the docker build command can be pushed to container registries.

- **`Docker pulls <container registry link for image>`**: This command pulls the image from the registry to a local machine. Look at the thick solid lines and follow the flow labelled as **`b`**.
    - **`b.1`**: The docker daemon interprets the docker pull command, and a pull call is made to a remote repository.
    - **`b.2`**: Docker image is pulled to a local system.
- **`Docker run <image name>`**: Create a container using one of the docker images available locally in the systems.
    - **`c.1`**: Docker run command interpreted by docker daemon.
    - **`c.2`**: Containers are created using images. Image labeled as one is used in creating container **`c1`**, and image two is used in creating containers **`c2`** and **`c3`**.

Now is the time to investigate the complete preceding defined process. For this exercise, refer to the **`docker-demo-python-app`** folder in the code base attached to this chapter. It is a simple hello world Python application. If you look at the folder's contents, there are python-related files and a file named **Dockerfile**.

You will use docker hub, an openly available container registry, for this exercise. Follow the given steps:

1. **Log in to the docker hub**

   Type the following command and enter your username and password for the docker hub. To create this username and password, get yourself registered at **https://hub.docker.com/signup**.

   `$ docker login`

   Refer to *Figure 1.3*:

```
Login with your Docker ID to push and pull images from Docker Hub. If you don't have a D
Username: swapnildubey1984
Password:
WARNING! Your password will be stored unencrypted in /home/sdubey7/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

Figure 1.3: *Docker hub Login*

2. **Build an application image**

   In this step, you will build the docker image in local. We will discuss how a docker file looks in the next step.

   `$ docker build -t demo-python-app:1.0.1 .`

Once the preceding command completes, run the following command if your docker image is present locally:

```
$ docker images|grep 'demo-python'
```

3. **Build multistage images**

It is time to investigate the docker file you used to create an image.

```
FROM python

# Creating Application Source Code Directory
RUN mkdir -p /usr/src/app

# Setting Home Directory for containers
WORKDIR /usr/src/app

# Installing Python dependencies
COPY requirements.txt /usr/src/app/
RUN pip install --no-cache-dir -r requirements.txt

# Copying src code to Container
COPY . /usr/src/app


# Application Environment variables
#ENV APP_ENV development
ENV PORT 8080


# Exposing Ports
EXPOSE $PORT


# Setting Persistent data
VOLUME ["/app-data"]


# Running Python Application
CMD gunicorn -b :$PORT -c gunicorn.conf.py main:app
```

In the preceding file, you can see the first line, **`FROM python`**, meaning that this image, when built, will first pull the Python image and then prepare a new image by adding the following details in the **`Dockerfile`**.

This is known as multistage pipelines, and there are obvious advantages. You can build an image once and then reuse and share the same image as sub images in across multiple images. For example, in your Enterprise, there could be one hardened image by security team for Python, and all teams could use the hardened Python image and use it to create application code specific image.. This makes the **`Dockerfile`** creation simple and more straightforward.

Also, note the constructs like **`RUN`**, **`COPY`**, **`EXPOSE`**, and so on. These are docker-specific constructs and have a special meaning in the docker container runtime environment.

4. **Store images in registries**

   The image **`demo-python-app:1.0.1`**, which you built in step 2, is still available locally, meaning that no other machine can create a container using that image. For this, you have to share the image with the container registry. You will be using the docker hub for this. Once you have an account created and have logged in to docker hub, you can trigger the following two-step process to push the image:

   i. **Step 1**: Tag the image

      ```
      $ docker tag demo-python-app:1.0.1 <dockerhub-username>/
      demo-python-app-dockerhub:1.0.
      ```

   ii. **Step 2**: Push the image to docker hub

      ```
      $ docker push <dockerhub-username>/demo-python-app-
      dockerhub:1.0.
      ```

   On docker hub web page, you can see if the image is pushed or not. Refer to *Figure 1.4*:

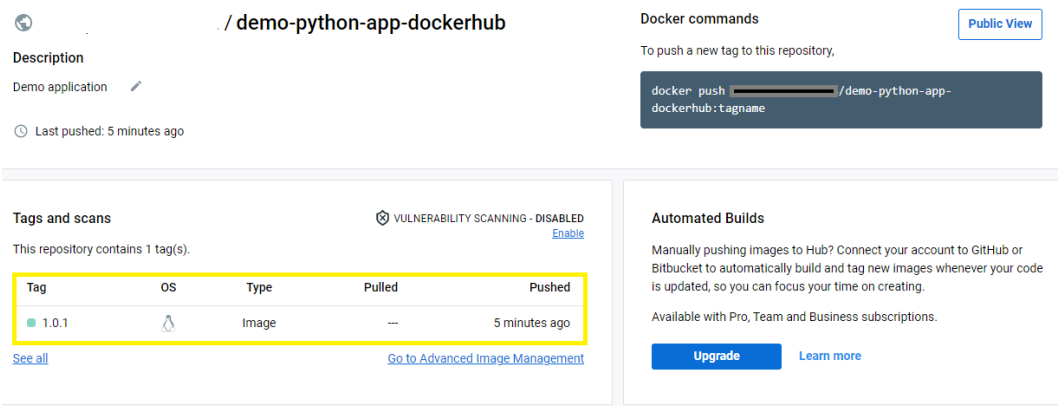


Figure 1.4: *Docker Hub Repo*

5. **Container runtime**

   The docker image pushed to the docker hub can now be pulled into any machine having docker installed. The pull of the image will make a copy from the remote repo to the local machine, and then the docker image can be executed.

   ```
   $ docker pull <dockerhub-username>/demo-python-app-
   dockerhub:1.0.1
   ```

   ```
   $ docker images| grep "demo-python"
   ```

   The preceding docker image command will now show two results: the local one, that is, **`demo-python-app:1.0.1`**, and **`demo-python-app-dockerhub:1.0.1`**. Refer to *Figure 1.5*:

```
demo-python-app                                  1.0.1     aafe172b02ee   24 minutes ago   951MB
[redacted]/demo-python-app-dockerhub             1.0.1     aafe172b02ee   24 minutes ago   951MB
```

Figure 1.5: *Local Docker Images*

   As the last step, you can create a docker container using the following command:

   ```
   $ docker run -d -p 8080:8080 <dockerhub-username>/demo-python-app-
   dockerhub:1.0.1
   ```

Open the web browser and feed the URL **`localhost:8080`**; a web page will open, and this will show that the container is created and exposed at port **`8080`** of the machine.

Introduction to Kubernetes

Kubernetes is an open-source container orchestrator for efficiently deploying and hosting containerized applications. Google initially developed it as an internal project to host scalable distributed applications reliably. In modern software systems, services are delivered over the network via APIs. The hosting and running of the APIs generally happen over multiple machines present geographically at the same or different locations. Also, since the data is growing every day, the scalability aspect of such services has started taking center stage, with no point in service-delivering responses breaching **Service Level Agreements** (**SLA**). Your application should use the optimal infrastructure to keep costs in check. Both the aspects of applications, that is, being scalable (up and down) and distributed, make sense only when the system is reliable. An algorithm running on such modern systems should produce the same results in multiple runs without any dependence on where and how the application is hosted.

Since Kubernetes was made open-source in 2014, it has become one of the most popular open-source projects in the world. Kubernetes APIs have become the de facto standard for building cloud-native applications. Kubernetes is a managed offering from almost all cloud providers: Google cloud platform, Amazon Web Services, and Microsoft Azure. Kubernetes, also known as K8S, automates containerized applications' deployment, scaling, and management. It provides planet-scale infra; if you keep supplying physical infrastructure, Kubernetes can scale up your application to significant levels. The larger the deployment, the greater the chance of parts of the infrastructure failing; Kubernetes has auto-healing properties, enabling automated recovery from failures.

Kubernetes also has some extremely mature features apart from the ones already mentioned. A few of the handy ones are as follows:

- **Capability to scale**: The application deployed in Kubernetes can scale horizontally (scaling up and down) and vertically (scaling in and out).
- **Security**: Kubernetes provides a platform for secured communications between multiple services. The extent depends on the type of application, for example, applying authentication and authorization on the services accepting internet data (external, front facing) to user authentication and consent to all services (internal and external)
- **Extensibility**: This refers to adding more features to the Kubernetes cluster without impacting the already present applications. For example, you can integrate plugins that will produce metrics about your application that are needed to perform SRE activities.
- **Support for batch executions**: We have only discussed services so far; however, Kubernetes provides support for executing batch jobs and also provides the ability to trigger cron jobs.
- **Rollbacks and roll-outs**: Kubernetes support features to roll back and roll out your application in stages, meaning that you can choose to deploy a new version of the service by just allowing it to serve 10% of users and then allow it for all.
- **Storage and config management**: Kubernetes provides the capability to use various storage solutions – SSD or HDD, Google Cloud Storage, AWS S3, or Azure Storage. In addition, Kubernetes has support for effectively managing general and secret configurations.

In the current book, you will see the preceding features being described and explained in depth, with special attention to security aspects and production readiness of the Kubernetes platform and applications.

Kubernetes architecture

Kubernetes is a complete and competent framework for modern workloads, but it is also very complex. When they read the documentation, many people get overwhelmed and get lost in the amount of information it provides. In this section, you will see the architecture of Kubernetes, and we will talk about the basics of the architecture, its components, and the roles each component plays in how Kubernetes does what it does.

Kubernetes follows a master-worker architecture. Consider *Figure 1.6*; you will see components - worker nodes and master nodes. As the name suggests, worker nodes are where actual work happens, and the master node is where we control and synchronize the working between worker nodes:

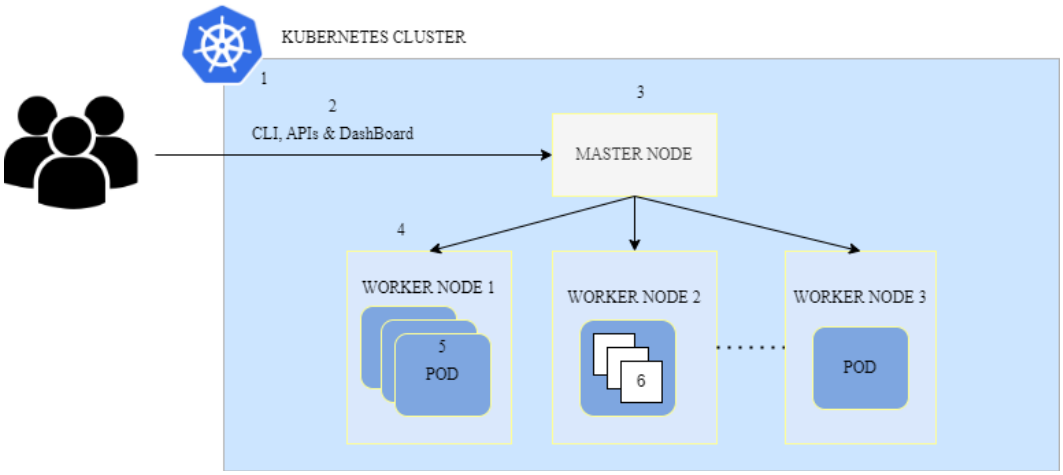


Figure 1.6: *Kubernetes 101*

Refer to the numerical labeling in *Figure 1.6* with the following corresponding numerical explanations:

1. Label 1 represents a complete Kubernetes cluster. A Kubernetes cluster is a collection of physical machines/nodes or virtual machines, with an official limit of a max of 5000 nodes. The control plane (Master) and workload execution plane (Worker) are deployed on these nodes. The expectation from the nodes comes from expectations from the Kubernetes components. For example, your master machine could only be a Linux box, while the worker nodes can be windows boxes too.

    a. Kubernetes is responsible for identifying and keeping track of which nodes are available in the cluster. Still, Kubernetes does not manage the node, which includes things like managing the file system,

updating the operating system security patches, and so on, inside the node. The management of the node becomes the responsibility of a separate components/team.

2. Any system/entity or person (developer or admin) can interact with the Kubernetes cluster via CLI, APIs, and Dashboard. All these interactions happen only via Master nodes.

3. The master node manages and controls the Kubernetes cluster and is the entry point for all admin tasks. Since the master node is responsible for maintaining the entire infrastructure of the Kubernetes cluster, when master nodes are offline or degraded, the cluster ceases to be a cluster. The nodes are just a bunch of ad hoc nodes for the period, and the cluster does not respond to the creation of new resources(pods), node failures, and so on. No new workloads can be launched on the cluster.

4. Worker nodes are the workhorses of the cluster, which perform the actual processing of data. They only take instructions from the Master and revert to the Master. If they do not receive any signal from the Master, they will keep waiting for the following instructions. For example, in the scenario of Master being down, the worker node will finish the work running on them and will keep waiting. If a worker node is down, it results in the low processing capability of the cluster.

5. Kubernetes Pods host workloads. A workload and all its supporting needs, like exposing ports, infrastructure and other networking requirements, and so on, are specified in a YAML file. This file is used to spin up a container or multiple containers in a Pod. Since you define one YAML per pod and each pod can have multiple containers, all containers share the resources inside a pod. For example, if your workload creates two containers inside the pod and your YAML file assigns them, both pods will share this one core of the CPU.

6. Containers represent the containerized version of your application code. Containers inside one pod are co-located and co-scheduled to run on the same Kubernetes work node. Kubernetes support multiple container runtime environments like containerd, CRI-O, docker, and several others. You will see docker being used throughout the book.

With the preceding behavioral concepts in mind, let us look at the components and internal working of the Master and worker nodes.

Kubernetes Master

Kubernetes Master, or the control plane of the Kubernetes cluster comes to life when a Kubernetes cluster is created and dies when a cluster is deleted. Generally, the

responsibility of managing and maintaining the control plane lies with the team that creates the cluster. This component is one unified endpoint of the cluster using which all the administrative calls are made to the cluster. The role played by the master is crucial, as, without a master, a cluster is just an incapable collection of unattached nodes. Generally, for fault tolerance, multiple Masters are set up, with one master being active and serving, and others following the active one. When an active master node goes down, a follower master node becomes active.

Consider *Figure 1.7*. As we can see, Kubernetes master has four components: etcd, scheduler, controller, and API server.

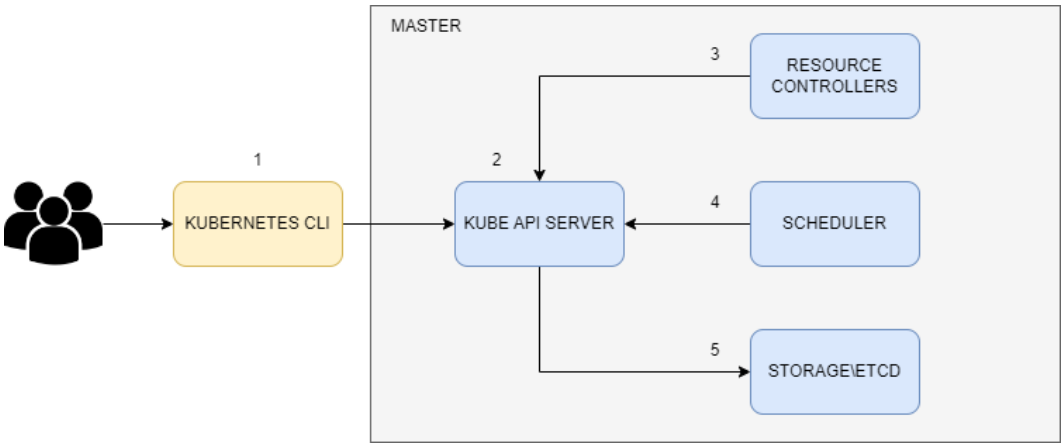


Figure 1.7: *Kubernetes Master*

Refer to the numerical labelling in *Figure 1.7* with the following corresponding numerical explanations:

1. An outside entity, like a developer, admin, or another system process, can interact with the Kubernetes cluster directly using HTTP/gRPC or indirectly using Kubernetes command-line Interface - Kubectl and UI.

2. All communication coming from the outside world goes via the API server. When a request comes in, the API server validates the request and then processes and executes them.

3. The resource controller does resource consumption and allocation checks in an infinite loop (continuously without stopping). It knows the desired state of resources and compares that with the current state of the resources in the cluster, and if they are not the same, it takes corrective actions to minimalize the delta. In short, the controller works continuously to make the current state of resources a desirable state.

4. The scheduler schedules the work of different nodes. It has resource consumption statistics of each cluster's worker node. Based on sufficient

infra bandwidth on a node superimposed with factors like quality of service, data locality, and other parameters, the scheduler selects a worker node to schedule the work in terms of services and pods.

5. Storage/**`etcd`** is a distributed key-value store that stores the cluster state. **`Etcd`** is written in Golang and is based on the RAFT consensus algorithm. The Raft algorithm allows a group of machines to behave coherently. Even if a few members fail, the algorithm keeps working. There is one master and others who follow the master.

Apart from maintaining state, **`etcd`** also stores subnet information and ConfigMaps (Kubernetes construct to manage configuration).

Since you understand the behavioral need of each component in Kubernetes master now, let us cover some best practices around setting up Kubernetes Master. These best practices will help your cluster be resilient, reliable, and easily recoverable in case of catastrophic failures:

- Kubernetes master should be replicated across three nodes (possibly distributed across multiple zones) at least for a highly available Kubernetes cluster. In case of failures, **`etcd`** needs the most master nodes to form a quorum and continue functioning. For this, an odd number of masters should be created.

- **`Etcd`** is a component responsible for storing and replicating the state of the Kubernetes cluster. Because of the type of role played, **`etcd`** has high resource requirements (memory to hold data and CPU to serve requests with minimal latency). It is a best practice to separate **`etcd`** deployment (from other Kubernetes daemons) and place them on dedicated nodes.

- **`Etcd`** stores the cluster state. Hence, it is advised to set up a backup for **`etcd`** data, generally on a separate host. In the case of on-prem deployments, a snapshot could be taken by running the '**`etcdctl`** snapshot save' command. If the Kubernetes setup is on the cloud using storage volumes, a snapshot of the storage volume could be taken and saved on the blob store.

- Replicate two replicas across zones in active and passive setup for the controller manager and scheduler, setting up two controller managers and schedulers, one active and one passive. The active one will serve the request, and the passive one will follow the active one. Once the active one goes down for any reason, the passive one will become active. This can be done by passing the **`--leader-elect`** flag to the Kube scheduler.

- It is recommended to set up automated monitoring and alerting for the Kubernetes master. Kubernetes master components can emit metrics in a particular format that is very easily configurable with several tools available in the market, for example, Dynatrace, Datadog, Sysdig, and so on.

The list of best practices keeps evolving, and the above is by no means the complete list of possible approaches. But it makes sense to identify the definition of resiliency and reliability if your team owns Kubernetes cluster deployment and maintenance.

Kubernetes Worker

Kubernetes worker is a physical or a virtual node that runs the workloads and is controlled by the Kubernetes master. Pods (a collection of containers that will be covered in depth in *Chapter 2: PODs*) are scheduled to run on worker nodes, which have the necessary tools to connect them and execute real-world use cases.

Any interaction of the application running inside the pod happens via connection to worker nodes and not the master nodes.

Consider *Figure 1.8*; Kubernetes Worker comprises of three components – pods, Kubelet and Kubeproxy:

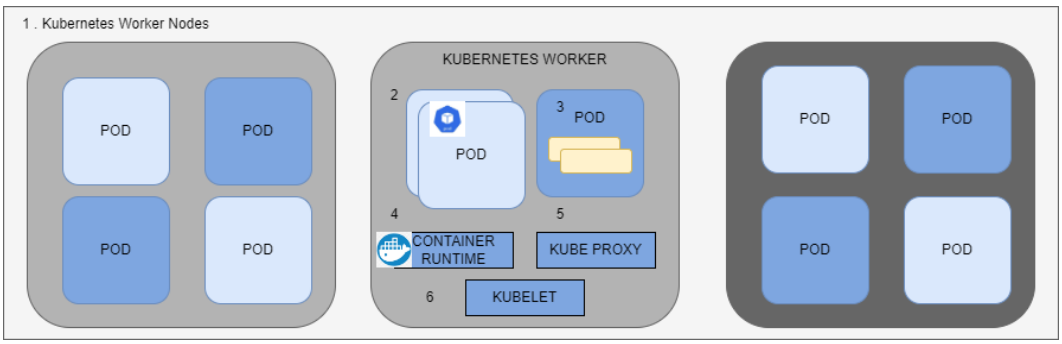


Figure 1.8: *Kubernetes Worker*

Refer to the numerical labelling in *Figure 1.8* with the following corresponding numerical explanations:

1. A Kubernetes cluster has multiple worker nodes (physical or virtual), with a maximum limit of 5000 nodes.
2. Each worker node has a collection of pods. The number of pods depends on the total size of worker divided by how much resource a pod will consume. There is an upper limit to the number of pods that can be created, i.e., no more than 110 pods per node and no more than 150,000 total pods.
3. A *Pod* is a group containers with shared storage and network resources and a specification of how to run the containers.
4. Container runtime is a software component that can run containers on a node. Also known as a container engine, it is responsible for loading container images in repositories, monitoring local system resources, isolating system resources for a container, and managing the container life cycle.

   a. Examples of container runtimes are runC, containerd, Docker, and Windows Containers. In *Figure 1.8*, we see the icon of Docker, and you will see this whole book using Docker as the container runtime.

5. Kubeproxy manages and maintains the network rules on each worker node. These network rules facilitate communication across your pods, as well as, from a client inside or outside of the Kubernetes cluster.

6. Kubelet is responsible for starting a pod with a container inside it. It is a process of Kubernetes that has an interface with the container runtime and the node. Kubelet takes as input pod specifications and ensures that the actual running pod meets the container definitions inside the specifications (quantitative and qualitative). It will not manage any containers that are not created via Kubernetes.

Light-colored pods are logically separated from dark-colored pods; this logical segregation is known as namespaces. Let us now look at some general best practices around Kubernetes workers:

- Kubernetes worker nodes are distributed into logical segregation known as namespaces. It is recommended to set limits to these namespaces; else, one namespace might affect the others by expanding too much. Resource Quotas limit the total amount of CPU, memory, and storage resources consumed by all containers running in a namespace.
- You can limit the number of pods that can spin up inside the namespace and the number of pods that can be created inside a node.
- It is recommended to use Horizontal Pod scaling and increase the number of pods when the need arises. This will result in optimal use of the cluster by applications that need the resources for processing.
- Generally, some resources on each node are configured for the system process. These system processes primarily are operating system-level daemons and Kubernetes daemons.
    - **`--Kube-reserved`**: Allows you to reserve resources for Kubernetes system daemons, such as the kubelet, container runtime, and node problem detector
    - **`--system-reserved`**: Allows you to reserve resources for OS system daemons, such as sshd and udev
- In case of cloud deployments, configure cluster Autoscalar, which can scale up your cluster based on two signals: utilization of nodes and whether there is a need to scale up. The CA will also downscale the cluster and remove idle nodes if they are underutilized.

As mention earlier, these guidelines keep evolving and improving, and they differ from use case to use case. For example, if you have an application that cannot be modelled into scalable multi-pod applications, keep static infrastructure.

Principles of immutability, declarative and self-healing

In today's software world, users not only expect new features and improvement to existing features iteratively, but there is also an unsaid rule of delivering new features with a lot of resiliency, reliability, and stability. Kubernetes goes by the following principles to support rapid and reliable software development and release.

Principle of immutability

On our laptops, whenever an OS update comes, the update might be allowed to install or may be halted for some time. This results in different OS flavors running on two other laptops even though they are running the same operating system version. Along similar lines, in software systems where we have a tomcat server running with version A, if there is a need to update the version, that could be updated by manually taking actions. If you want to perform a similar update on multiple servers, versions of tomcat will be updated only when done externally. There are high chances of two servers running different versions. Likewise, there needs to be a comprehensive way to keep note of the changes that were done. The preceding is an example of mutable infrastructure, as you modified the same infrastructure with new packages or libraries. You also realized that there could be a risk of software drifts with the added risk of no efficient logging mechanism of changes. This could result in rolling back complex changes.

Kubernetes works on immutability, that is, you build the complete infrastructure at the start and launch it. If you have a use case of updating the existing infrastructure, create a new version and relaunch. Delete the old infrastructure component. One might debate that a similar thing could be achieved by logging in to the container and doing it manually. However, the immutability approach brings some advantages, and the first key advantage is that the artifact used to generate new infrastructure is a document of what has changed. It could be easily identified if something goes wrong, and fixes can be applied. In addition, this approach gives an option of risk-free rollback in case of issues with the new setup.

Declarative configurations

Kubernetes uses the concept of declarative configuration extensively. One can define all the objects in Kubernetes, which represents the system's desired state. It is the responsibility of the actors in Kubernetes to bring the system up to the declared

desired configuration. The approach contrary to declarative configuration is imperative systems. In imperative systems, rather than defining the end state of the system, a series of steps are declared upfront. For example, if you want your four replicas of an application to be deployed, then in the case of imperative systems, you will give instructions to create replicas one, two, three, and four. In the declarative configuration, the system will be told to create four replicas, and the system thread will do the work.

Declarative configurations, combined with a version control system and the capability of Kubernetes to achieve declared configurations, make the rollback of a change easy. It is simply reverting the configuration file to one version older. This was impossible in imperative systems, as they would tell how to move from state one to state two without giving the rollback instructions.

Self-healing systems

Another vital principle of the Kubernetes world is self-healing. Assume that you want to deploy three replicas of your application, and due to a resource crunch, only two replicas were successfully created. Kubernetes will continuously keep on trying to create the three replicas. Not just this, let us assume that Kubernetes was initially able to create three replicas, but due to node failure, one of the replicas got removed. Kubernetes daemon will continuously keep on trying to create the third replica.

On the other side of the self-healing system, in the case of imperative systems, you generally configure the failure stats to raise some signal, and this signal is captured, and then actions associated with this event (captured signal) is triggered. . For example, if the number of replicas is less than what is needed, trigger an alarm and send emails to the IT team so that they can manually create one more replica. This form of system generally takes more time to rectify, requires more effort and is less reliable.

Installing Kubernetes

There are three general environments where teams deploy the Kubernetes setup. These are Kubernetes deployment on-prem, Kubernetes deployment on the cloud, and local deployment. The first two environments, that is, on-prem and cloud, are where an IT team is involved, which takes care of creating and managing the infrastructure case of on-prem; it will be your in-house IT team and the cloud IT team on the cloud.

Creating Kubernetes Infrastructure is easy on cloud providers as all major public cloud providers like Google Cloud Platform, Amazon Web Services, and Microsoft Azure support Kubernetes as a managed offering, and it takes a few minutes and just one command to spin up a Kubernetes cluster.

Let's take an example of GCP:

```
gcloud container clusters create dummy-cluster --zone us-central1-a
--node-locations us-central1-a,us-central1-b,us-central1-c
```

The preceding command creates a Kubernetes cluster named **`dummy-cluster`** in zone **`us-central1-a`**.

Similar kinds of commands are available in the other two cloud providers as well. These are simple and straight, and if you have access, run a command and get a production-grade battle-tested Kubernetes cluster ready.

In this section, you will investigate two approaches for setting up Kubernetes clusters locally. Keep in mind that local cluster setup is only for development purposes and should not be used to investigate production-grade issues, especially related to networking and scaling, that is, where there is communication across two nodes involved. After completing the development and testing locally, the workflow should ideally be tested once on full-fledged Kubernetes cluster as well.

There are two ways to install Kubernetes locally; let's look at both of these.

Installing Kubernetes locally using Minikube

Minikube setup is a simple one-node Kubernetes cluster. A few features will be discussed in the book, which are either unavailable or have minimal availability with Minikube. We will use the Google Cloud Platform to showcase and demonstrate the concept in those cases.

Make sure you have a hypervisor installed on your local machine to use Minikube. For Linux-based systems, this is generally VirtualBox. On Windows, the Hyper-V hypervisor is the default option.

Docker For Desktop comes with an integrated Minikube setup. However, if you want to install libraries yourself, you can refer to **https://github.com/kubernetes/minikube**, which has libraries of Minikube for Linux, macOS, and Windows.

The following command is used for installing Minikube in Ubuntu machine:

```
$ sudo apt install -y curl wget apt-transport-https

$ wget https://storage.googleapis.com/minikube/releases/latest/minikube-
linux-amd64

$ sudo cp minikube-linux-amd64 /usr/local/bin/minikube

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

$ minikube version
```

Once Minikube library is installed, you can create a local cluster using the following command:

```
$ minikube start
```

You can stop the cluster using the following command:

```
$ minikube stop
```

For deleting the entire cluster, you can use the following command:

```
$ minikube delete
```

Installing Kubernetes in Docker

In this approach, docker containers are used to simulate multi node Kubernetes cluster instead of virtual machines. This kind of project (Kubernetes IN Docker - **https://kind.sigs.k8s.io/**) is widely used by developers for testing applications quickly and easily.

```
$ kind create cluster --wait 2m
$ export KUBECONFIG="$(kind get kubeconfig-path)"
$ kubectl cluster-info
$ kind delete cluster
```

You can deploy the docker hub image now directly on the Minikube installation. Use the following command to deploy the image as Pod in the local Kubernetes:

```
$ kubectl create deployment demo-python-app --image= <dockerhub-username>/
demo-python-app-dockerhub:1.0.1 -- /agnhost netexec --http-port=8080
```

You can check the deployment using the following command:

```
$ kubectl get deployments
```

Figure 1.9 is the output of the preceding command, showing that a deployment is created in Kubernetes:

```
NAME              READY   UP-TO-DATE   AVAILABLE   AGE
demo-python-app   0/1     1            0           18s
```

Figure 1.9: *Kubernetes Deployments*

Kubernetes client

Kubectl is the official command-line interface for Kubernetes for interacting with Kubernetes API. Kubectl could manage almost every object in Kubernetes, such

as ReplicaSet, Pods, and Services. In this section, you will look into the common commands. This will be an extremely important section as this book will use the CLI interface a lot for showcasing the hands-on exercises. In this part, you will see **`kubectl`** commands, which will display critical information related to Kubernetes Master and Worker node.

Checking the version

To check the version, use the following command:

```
$ kubectl version
```

The output of the preceding command is as follows:

```
Client Version: version.Info{Major:"1",
Minor:"25", GitVersion:"v1.25.4",
GitCommit:"872a965c6c6526caa949f0c6ac028ef7aff3fb78",
GitTreeState:"clean", BuildDate:"2022-11-09T13:36:36Z",
GoVersion:"go1.19.3", Compiler:"gc", Platform:"linux/amd64"}

Kustomize Version: v4.5.7

Server Version: version.Info{Major:"1",
Minor:"21", GitVersion:"v1.21.14-gke.4300",
GitCommit:"348bdc1040d273677ca07c0862de867332eeb3a1",
GitTreeState:"clean", BuildDate:"2022-08-17T09:22:54Z",
GoVersion:"go1.16.15b7", Compiler:"gc", Platform:"linux/amd64"}
```

The command output shows two different versions: the local **`kubectl`** version and Kubernetes API server.

Checking the status of Kubernetes Master Daemons

To check the status of Kubernetes Master Daemons, use the following command:

```
$ kubectl get componentstatuses
```

The output of the preceding command is as follows:

```
NAME                              STATUS    MESSAGE             ERROR
scheduler                         Healthy    ok
etcd-1                            Healthy    {"health":"true"}
```

In the preceding output, you can see the status of controller-manager, scheduler, and etcd.

Listing all worker nodes and describing the worker node

To list all worker nodes, use the following code:

```
$ kubectl get nodes

NAME                                                STATUS
ROLES    AGE       VERSION

node-2020-p-data-services-460b17e5-ur74       Ready,Master        <none>
3h13m    v1.21.14

node-2020-p-pool-8574b14d-1t00                          Ready
<none>    417d       v1.18.20

node-2020-p-pool-8574b14d-29kd                  Ready
<none>    31h         v1.18.201

node-2020-p-pool-8574b14d-5b51                   Ready
<none>    417d       v1.18.20

node-2020-p-pool-8574b14d-7q9m                 Ready
<none>    270d        v1.18.20
```

The preceding output represents a Kubernetes cluster with five nodes. The very first node represents the master. You can see the age column; it contains multiple values, demonstrating that nodes come and go. Observe the version section of the output, which indicates that different node with different versions of Kubernetes run well together.

Using the following command, you can check the details of a particular node:

```
$ kubectl describe nodes node-2020-p-pool-8574b14d-1t00
```

The preceding command gives a lot of information about the node. First, it gives information about the OS running, processor, and the type of machine:

```
Name:               node-2020-p-pool-8574b14d-1t00
Roles:              <none>
Labels:              beta.kubernetes.io/arch=amd64
                          beta.kubernetes.io/instance-type=n1-standard-1
                          beta.kubernetes.io/os=linux
```

Next, you will see details about operations on nodes and conditions, the status and their time of occurrence and resolution. Refer to *Figure 1.10:*

Type	Status	LastHeartbeatTime	LastTransitionTime	Reason	Message
FrequentContainerdRestart	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	NoFrequentContainerdRestart	containerd is functioning properly
KernelDeadlock	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	KernelHasNoDeadlock	kernel has no deadlock
ReadonlyFilesystem	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	FilesystemIsNotReadOnly	Filesystem is not read-only
CorruptDockerOverlay2	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	NoCorruptDockerOverlay2	docker overlay2 is functioning properly
FrequentUnregisterNetDevice	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	NoFrequentUnregisterNetDevice	node is functioning properly
FrequentKubeletRestart	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	NoFrequentKubeletRestart	kubelet is functioning properly
FrequentDockerRestart	False	Mon, 12 Dec 2022 05:32:21 +0000	Wed, 20 Oct 2021 11:12:34 +0000	NoFrequentDockerRestart	docker is functioning properly
NetworkUnavailable	False	Wed, 20 Oct 2021 11:12:42 +0000	Wed, 20 Oct 2021 11:12:42 +0000	RouteCreated	RouteController created a route
MemoryPressure	False	Mon, 12 Dec 2022 05:30:15 +0000	Sun, 23 Oct 2022 01:35:51 +0000	KubeletHasSufficientMemory	kubelet has sufficient memory available
DiskPressure	False	Mon, 12 Dec 2022 05:30:15 +0000	Wed, 20 Oct 2021 11:12:29 +0000	KubeletHasNoDiskPressure	kubelet has no disk pressure
PIDPressure	False	Mon, 12 Dec 2022 05:30:15 +0000	Wed, 20 Oct 2021 11:12:29 +0000	KubeletHasSufficientPID	kubelet has sufficient PID available
Ready	True	Mon, 12 Dec 2022 05:30:15 +0000	Wed, 20 Oct 2021 11:12:40 +0000	KubeletReady	kubelet is posting ready status. AppArmor enabled

Figure 1.10: *Node Operations & Conditions*

Another important information produced as outcome is the details of the capacity and allocation of resources:

```
Capacity:
    attachable-volumes -pd:  127
    cpu:                        1
    ephemeral-storage:          98868448Ki
    hugepages-2Mi:              0
    memory:                     3773756Ki
    pods:                       110
Allocatable:
    attachable-volumes-gce-pd:  127
    cpu:                        940m
    ephemeral-storage:          47093746742
    hugepages-2Mi:              0
    memory:                     2688316Ki
    pods:                       110
```

The following is the critical system information, such as OS version, container runtime used, Kubelet and Kub proxy version:

```
System Info:
  Machine ID:                 7ad08de7d10bf9ffe6ced83df79f6248
  System UUID:                7ad08de7-d10b-f9ff-e6ce-d83df79f6248
  Boot ID:                    1d9b96b5-7445-48f7-b341-da53fed2ad14
  Kernel Version:             5.4.109+
  OS Image:                   Container-Optimized OS
  Operating System:           linux
```

```
  Architecture:                amd64
  Container Runtime Version:   docker://19.3.15
  Kubelet Version:             v1.18.20
  Kube-Proxy Version:          v1.18.20
```

Figure 1.11 features some information about the pods running on the node:

```
Non-terminated Pods:          (12 in total)
  Namespace                   Name                                          CPU Requests  CPU Limits  Memory Requests  Memory Limits  Age
  ---------                   ----                                          ------------  ----------  ---------------  -------------  ---
  default                     falco-bdxct                                   100m (10%)    1 (106%)    512Mi (19%)      1Gi (39%)      409d
  default                     falco-falcosidekick-5cdb67fc48-xrdrq          100m (10%)    0 (0%)      0 (0%)           0 (0%)         318d
  default                     historical-ingestion-8ddc75d79-tjf8g          100m (10%)    0 (0%)      256Mi (9%)       0 (0%)         335d
  default                     pipeline-ingestion-5b7f78ff9b-jklwr           100m (10%)    0 (0%)      0 (0%)           0 (0%)         2d11h
  default                     piq-provisioning-automation-76479f7df6-bhn46  100m (10%)    0 (0%)      0 (0%)           0 (0%)         164d
  default                     service-vmg-5678dd9b74-dfdzp                  100m (10%)    0 (0%)      512Mi (19%)      0 (0%)         135d
  kube-system                 container-watcher-qg8jm                       25m (2%)      125m (13%)  50Mi (1%)        300Mi (11%)    25d
  kube-system                 fluentbit-gke-gtgv4                           100m (10%)    0 (0%)      200Mi (7%)       500Mi (19%)    62d
  kube-system                 metrics-agent-kh7k5                           3m (0%)       0 (0%)      50Mi (1%)        50Mi (1%)      105d
  kube-system                 kube-proxy-p-pool-8574b14d-1t00               100m (10%)    0 (0%)      0 (0%)           0 (0%)         417d
  kube-system                 pdcsi-node-fgkqn                              10m (1%)      0 (0%)      20Mi (0%)        100Mi (3%)     47d
  prometheus                  prometheus-prometheus-node-exporter-s84fn     0 (0%)        0 (0%)      0 (0%)           0 (0%)         417d
Allocated resources:
```

Figure 1.11: *Pod Information*

The number of options available in Kubectl are many. Those options will be covered as and when the book talks about a construct. However, for now, using the preceding commands, anybody can gather some key nature of the Kubernetes cluster, such as the type of infra, stability, and various conditions that could make the situation for a node worse.

Strategies to validate cluster quality

Till now, we investigated what a Kubernetes cluster looks like, what daemons do, and what are some of the best practices while configuring Kubernetes-owned threads. In this section, we will look at cost-efficiency and security as a strategy/heuristic to define the quality of the cluster.

Each use case is different. Creating the right flavor of a production-grade cluster depends on the non-functional requirements of the project. For example, ideally, you should configure the most resilient, reliable, and secure system, but high resiliency, high reliability, and high security come with added cost and effort to create and manage. Hence, assessing the right needs of the use case and taking appropriate decisions on switching the right features and their levels in Kubernetes becomes vital.

Cost-efficiency as measure of quality

In this section, we will look at aspects of Kubernetes that can result in high costs and hence, need a decision to be taken upfront. A cluster meeting SLA with the lowest price possible is a heuristic to measure the quality of the cluster.

Right nodes

Parameters that impact a node's price on-premises or cloud include the processor vendor (Intel, AMD, or AWS), **OS**, instance generation, CPU and memory capacity, processor architecture (x86, Arm64), and ratio, and the pricing model (specific to cloud on-demand, reserved instances, savings plans, or spot instances).

You can target for maximum utilization of nodes. Still, maximizing utilization without negatively impacting workload performance is challenging. As a result, most enterprises find that they are heavily overprovisioned with generally less utilization across their Kubernetes nodes.

Request and restrict specifications for pod CPU and memory resources

It would help if you always define the right resource needs for your application running in a pod. For example, a single-threaded application defined to use four cores does not make sense. Similar is the case with memory. At the minimum, each application should be segregated into two groups: high memory consuming and high CPU consuming; and each should use appropriate infrastructure. The number of pods will eventually require nodes to accommodate them, and inflated resource needs imply an excessive number of node needs.

Similarly, choose pod CPU cores as factors of the number of cores available on a machine. For example, if the number of cores per machine is 4, choose a pod with CPU needs of 1,2, and 4. Choosing a pod size of three might result in 1 core going waste.

Persistent volumes

Any volume attached to your cluster has a cost. You can connect both SSD and HDD volumes, but which storage class it should be must be decided while designing the workloads. An SSD mount will burn more money. Also, try to avoid spinning this volume for the next 5 years upfront; instead, make smaller targets and add space as and when more space is needed. If your 5-year needs, say, the volume of 10 GB to be mounted, a yearly estimate might mean just 2 GB. In this case, plan for migration of upgraded volume every year.

Data transfer costs and network costs

There are a few widespread and prominent data transfer scenarios. For example, data transfer happens when pods communicate with each zone and region, with the control plane, and with load balancers, and pods interact with external services like databases, and data is replicated across regions to support disaster recovery.

High data transfer eventually results in increased costs. It is recommended that you strategize data transfer architecturally and make applications accordingly.

Along similar lines, you need to decide the entry point in the application and provide public IP for only that point instead of multiple Public IPs.

The preceding aspects are just a few pointers to be taken care of, but the idea is that these aspects, which incur cost on Kubernetes deployments, need to be understood. Proper governance is necessary so that the development team develops applications suiting your governance policies. If all the rules are met well, low cost with a breach of SLA becomes a suitable parameter for defining quality.

Security as a measure of quality

There are various levels of security across all the components of Kubernetes. One can opt for options to apply zero protection or can decide to use an extreme level of security (Zero Trust will be discussed in further detail in the later chapters).

A more secure cluster requires the effort of creation and maintenance. Designing the compliances of security for a use case is key to measuring the quality of the cluster for a use case. Security is one of the key fundamentals you will see being discussed throughout the book. But the rule of thumb is that high security means high cost and effort. Choose the level of security needed wisely and optimize the cluster accordingly.

Conclusion

Kubernetes is the most commonly used orchestration manager for container-based workloads. Kubernetes can be deployed on-premises and on all major cloud providers, so it has become the obvious choice for an orchestration platform. In addition to being obvious, Kubernetes provides a mature set of features to handle all kinds and patterns of modern workloads. Also, since Kubernetes provides many touch points during deployment, it could be optimized to work as per the use case handled. Take decisions and set up governance for your applications to achieve optimal throughput.

Points to remember

- Kubernetes is an orchestration platform that was initially developed by Google and then open sourced.
- Kubernetes is the go-to platform for your cloud-native applications, as Kubernetes can be deployed on-premises and on the cloud.

- Kubernetes allows running multiple container runtimes, the most famous being docker.
- It is easy to build an application of Kubernetes, but it is equally difficult to build an optimal application of Kubernetes.
- Do take a decision and set up governance about the deployment topology of nodes and the level of security.

Multiple choice questions

1. **Which of the following is a datastore for saving the cluster state of Kubernetes?**
    a. Pod
    b. Node
    c. Etcd
    d. None of the above
2. **Which of the following runs on each worker node?**
    a. Kubeproxy
    b. Kubelet
    c. Container runtime
    d. All of the above
3. What is the need for Container Orchestration?

Answers

1. **c**
2. **d**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 2
PODs

Introduction

Now that you have entered the world of Containers and Kubernetes, let us get to know it better. We will start with the basic building block in the Kubernetes, which is POD. The Pods provide running environment for containers. In other words, pods mean the world to containers, and containers cannot exist without pods in Kubernetes. Kubernetes is defined and widely known as the orchestrator for containers, and no doubt it does that, but it does it through pods.

Pod is an object in Kubernetes, just like there are many others, and we will get to know them in the upcoming chapters. In this chapter, we will talk about pods and their life cycle. Knowing about pods will help you not only identify and fix the issues in a Kubernetes cluster but will also make you ready for the subsequent chapters as we dig deeper. We will learn how to access the pods to debug the issues, what probes you can put to identify whether or not the pod is up and running, and how to get the best out of pods by setting some resource requests and limits.

We will discuss Pod security at the end of this chapter, not because it is of least importance, but because we first need to know some basics related to Pods to make them more secure. Securing pods in a Kubernetes cluster is a must when you are looking at Kubernetes cluster's security spectrum.

Structure

In this chapter, we will discuss the following topics:

- Concept of Pods
- CRUD operations of Pods
    - Creating and running Pods
    - Listing Pods
    - Deleting Pods
- Accessing Pods
    - Accessing via port forwarding
    - Running commands inside Pods using exec
    - Accessing Logs
- Managing resources
    - Resource requests: minimum and maximum limits to Pods
- Data persistence
    - Internal: Using data volumes with Pods
    - External: Data on remote disks
- Health checks
    - Startup probe
    - Liveness probe
    - Readiness probe
- Pod security
    - Pod Security Standards
    - Pod Security Admissions

Objectives

The aim of this chapter is to introduce readers to the foundation of Kubernetes, that is, Pods. This is the first chapter where users will start dealing heavily with Kubernetes objects and command-line tool **`kubectl`**. This chapter will help readers get familiar with the life cycle of Pods, how to access and manage them and how to check a pod's health. Examples and images in the chapter will ensure that readers get near-hands-on experience of playing with pods through **`kubectl`** commands.

Thus, this chapter will familiarize users with the world of Kubernetes and prepare them for the upcoming chapters in this book.

Concept of Pods

Containers contain the source code and the dependencies required for a software unit to run on its own. But what contains containers? The answer is Pod. The Pod contains containers, mostly one at a time, but sometimes a pod contains more than one container. If you think of Kubernetes as a house built from walls of bricks, then a pod is like an individual brick. Of course, that brick has its own ingredients, but when you start building a house, you start with the brick. Same is the case with pods: you use them to build an environment for containers of your application. And these pods ultimately make up the cluster.

Let us say you have an application that is very basic and provides 2-3 REST APIs. You can create a container with the image for the application and create a pod containing that container with just one command once you have the image in an image repository. *Figure 2.1* features a container in a pod in a node:

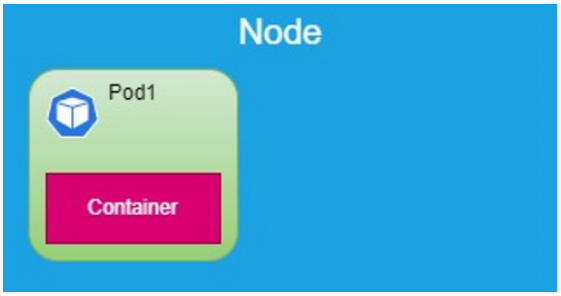


Figure 2.1: *A container in a pod in a Node*

Pods are the atomic scaling unit in a Kubernetes cluster, not the containers. That means when you scale up your application to handle more incoming requests, you scale the number of pods required for your application. *Figure 2.2* features a node containing 2 Pods:

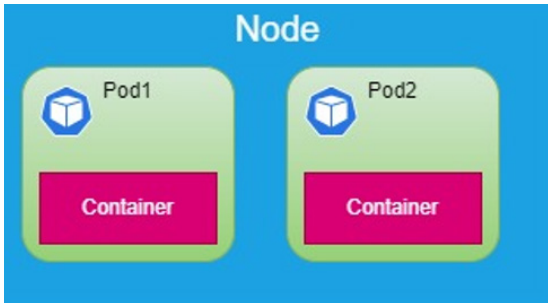


Figure 2.2: *A node containing 2 Pods*

CRUD operations on Pods

Let us discuss the life cycle of pods from creation to deletion. We will be using **kubectl** commands here.

Be assured that the world of Kubernetes is a yamlverse, where lots of yamls are used for different purposes. In this chapter as well, we will be looking at sample yamls to understand the concepts in more detail. Do note that Kubernetes supports JSON format as well, but we are mostly going to use yamls in this book; the use of JSONs for Kubernetes commands is not very popular.

Creating and running Pods

The **kubectl run** command is the quickest way to create a pod with a container image. The following command creates a running pod by pulling a nginx image from image registry:

```
kubectl run my-nginx  --image=nginx
```

Figure 2.3 shows how the output looks:

```
D:\>kubectl run my-nginx --image=nginx
pod/my-nginx created
```

Figure 2.3: *Create a Pod from container image*

You can also create pods with a **yaml** file having content as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.22.1
  ports:
  - containerPort: 80
```

Then you run the **kubectl apply** or **kubectl create** commands to create the pod:

```
kubectl apply -f nginx-pod.yaml
```

Here, **`nginx-pod.yaml`** is the file that has the content shown.

Let us take a note of key attributes in this yaml:

- **`spec.containers.image`** specifies which image to pull from the image registry.
- **`ports.containerPort`** specifies which port from the pod is used by the container.

When building real-world applications, you will not be creating pods through yaml files like these, but pods will be created through deployments. Deployments are another important object in the world of Kubernetes, and we will be discussing it in detail in one of the upcoming chapters.

Listing Pods

The command to list the pods in your namespace in cluster is independent of the nature of their creation. You can list the pods using the **`kubectl get pods`** command.

The **`get pods`** command lists the pods available in your cluster and in the specified namespace if any. *Figure 2.4* provides a sample output when only one pod is in the **`ContainerCreating`** state in the default namespace:

```
D:\>kubectl get pods
NAME       READY   STATUS              RESTARTS   AGE
my-nginx   0/1     ContainerCreating   0          6s

D:\>kubectl get pods
NAME       READY   STATUS              RESTARTS   AGE
my-nginx   0/1     ContainerCreating   0          9s
```

Figure 2.4: *Listing Pods*

It is obvious that the status or phase of a pod depends on the state of containers within it.

The pod goes through different phases in its life cycle, namely, **Pending**, **Running**, **Succeeded**, **Failed** and **Unknown**, as shown in *Figure 2.5:*

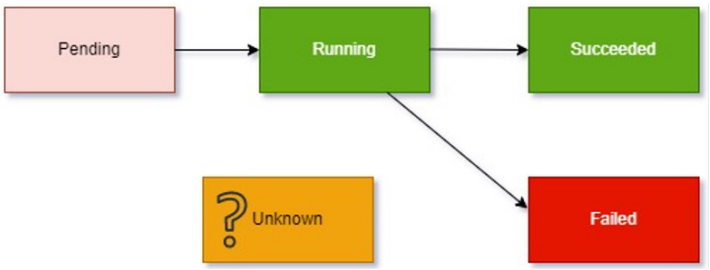


Figure 2.5: *Pod phases*

The Pod phases are:

- The **Pending** phase indicates that the pod is either pulling the image from the container or the pod itself is yet to be scheduled by the Kubernetes cluster.
- The **Running** phase indicates that **at least one** of the containers in the pod is running.
- The **Succeeded** phase can be misleading. Success here means success in terminating the containers and does not indicate the success in handling incoming requests or anything. A pod in the **Succeeded** state has one or more containers that are successfully terminated.
- Similarly, the **Failed** phase indicates that one (or more) of the containers in the pod was terminated ungracefully or unsuccessfully.
- The **Unknown** phase is Kubernetes' way of saying *phase not found*. This is mostly in the case of communication failure with the node containing the pod.

Containers in the pods have only three states, as shown in *Figure 2.6*:

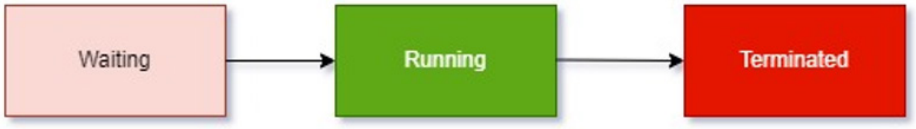


Figure 2.6: *Container states*

The Container states are**:**

- **Waiting:** This is the state that indicates that the container is not ready yet and is either pulling the image from registry or is busy with some prerequisite tasks before the start-up. If a container is in this phase, the pod will mostly be in the pending phase.
- **Running:** This can be termed as the *ideal* or *expected* state for a container and its associated pod, the Kubernetes control plane and application developer, SREs and so on. A running state for a pod makes everyone happy, barring exceptional cases when the pods are not getting terminated as expected. It indicates that the container is running fine, and there are no issues to worry about.
- **Terminated:** This phase indicates that the container has either completed its task and hence, terminated successfully or maybe it failed somewhere in-between. The **`kubectl describe pod`** command tells the reason for termination of the pod along with exit code.

The **kubectl describe** command can be used to get more information about specific pods, and you will get similar yaml response as shown previously, with some additional information:

```
kubectl describe pod my-nginx-pod
```

The events related to a pod can be known through the **kubectl describe pods** command, as shown in *Figure 2.7*:

```
Events:
  Type    Reason     Age    From               Message
  ----    ------     ----   ----               -------
  Normal  Scheduled  3m16s  default-scheduler  Successfully assigned default/my-nginx to docker-desktop
  Normal  Pulling    3m15s  kubelet            Pulling image "nginx"
  Normal  Pulled     3m2s   kubelet            Successfully pulled image "nginx" in 12.2351517s
  Normal  Created    3m2s   kubelet            Created container my-nginx
  Normal  Started    3m2s   kubelet            Started container my-nginx
```

Figure 2.7: *Events in a Pod*

One of the specifications that you should observe while looking at pod yamls is **restartPolicy**, which is specified for a pod and applies to all the containers in the pod. The three possible values for **restartPolicy** are as follows:

- **Always**: Always restart the container when it is terminated; this is the default value
- **OnFailure**: Restart the container only in case of failures such as liveness probe failure or a non-zero exit code from a process in the container
- **Never**: Never restart the containers automatically

Deleting Pods

Just like creation, deletion of pods is usually controlled through deployment objects. Direct deletion of pods via command line is not advised and can cause undefined behaviors for certain applications. Of course, there is a **kubectl** command to delete an individual pod as well, and you get no points for guessing this:

```
kubectl delete pod my-nginx
```

Figure 2.8 provides a sample output of the **delete** command, followed by **get pods** command:

```
D:\>kubectl delete pod my-nginx
pod "my-nginx" deleted

D:\>kubectl get pods
No resources found in default namespace.
```

Figure 2.8: *Deleting pods*

It should be noted that the **`kubectl delete`** command has an optional argument called grace-period, which defaults to 30 seconds. This means that all deletes are given 30 seconds to perform clean-up activities, if any. You can pass your value to **`--grace-periodargument`** if you want to increase or reduce the period of 30 seconds for deletion.

Accessing PODs

As the services in your cluster grow, you may find yourself in a situation where you must access individual pods of the services for troubleshooting. Let me warn you, it may sound scary initially, but let me also assure you that it makes you feel very good once you get comfortable with it.

Containers in the pod share the network resources like namespace, IP address and network ports. When there are multiple containers in the same pod, they can communicate with each other through localhost, because the IP address is shared. When containers want to communicate to the resources outside the pod, they must use IP networking.

Accessing via port forwarding

You can access a running pod via the **`kubectl port-forward`** command. Let us break that into parts to understand it better. As we know, a pod provides a running environment for the container, and it opens a window to the outside world for the container through a port.

The command that you should know is as follows:

```
kubectl port-forward my-nginx 8080:80
```

This command creates a secure tunnel from your machine to the pod called **`my-nginx`** via Kubernetes master node on the Kubernetes cluster. You need to be connected to a Kubernetes cluster beforehand. If you are using a Kubernetes cluster deployed on a public cloud like Google Cloud, Microsoft Azure or Amazon AWS, this means logging in to the cloud provider's consoles with credentials. The requests on port 8080 on your machine are forwarded to port 80 on the specified container. This is a blocking command, and while this command is running, you can access the pod on your localhost:8080. Refer to *Figure 2.9:*

```
D:\>kubectl port-forward my-nginx 8080:80
Forwarding from 127.0.0.1:8080 -> 80
Forwarding from [::1]:8080 -> 80
Handling connection for 8080
Handling connection for 8080
```

Figure 2.9: *port-forward on a pod*

If you have a pod running with nginx image and then you open **`localhost:8080`** in your browser while port-forward is happening, you will see a **Welcome to nginx!** page.

Most times, you will not need to port-forward for a pod, but you may run a similar command for service objects in Kubernetes. But if you know which of the pods within your service is faulty, you can use **`port-forward`** and a few more commands that we are going to look at to debug the issue.

Running commands inside PODs using exec

To run the commands inside pod, you need to use the **`exec`** command, as shown in *Figure 2.10:*

```
D:\>kubectl exec my-redis -- pwd
/data

D:\>kubectl exec my-nginx -- date
Sat Dec 10 10:15:56 UTC 2022
```

Figure 2.10: *Executing commands inside pod*

To actually peep into the pod and see the world inside, you can use **`-it`** flags, which give an interactive session. *Figure 2.11* is a snippet showing how can you get into a pod containing an nginx container and see the contents of the **`index.html`**:

```
D:\>kubectl exec my-nginx -it -- bash
root@my-nginx:/# ls
bin  boot  dev  docker-entrypoint.d  docker-entrypoint.sh  etc  home  lib  lib64  media
root@my-nginx:/# cd /usr/share/nginx/
root@my-nginx:/usr/share/nginx# ls
html
root@my-nginx:/usr/share/nginx# cd html
root@my-nginx:/usr/share/nginx/html# ls
50x.html  index.html
root@my-nginx:/usr/share/nginx/html# cat index.html
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
```

Figure 2.11: *Interactive session inside pod*

The **`bash`** command, along with the **`-it`** flag opens a bash prompt inside pod, and you can see the files inside the pod. This can be very useful in certain cases, but its misuse should be avoided.

Accessing logs

The **kubectl logs** command can be used to access logs for the container application running inside pod:

```
kubectl logs <pod-name> -c data-pod -n namespace1
```

The **-c** flag is used to select the container for which you want to see the logs. Here, **data-pod** is the container name. In case your pod has only one container, the **-c** flag is not required.

The **logs** command shows the logs of the current instance of the container by default. In case your application is crashing repeatedly and causing container restarts, and you want to see the logs of the terminated instance of the container, you can use the **-previous** flag.

In most cases, you will not need to see the logs at an individual pod level. Cloud providers support combining logs from all pods in a Kubernetes cluster, and there is good support to configure the retention of logs for longer period. Not only storage but searching and filtering across logs is also easier with the help of tools from cloud providers.

Managing resources

Containers within the same pod share resources, and when you define specifications for a pod, you can optionally specify the resource limits for the containers that it would contain.

Memory and CPU are the most common resources to specify for containers.

Resource requests: Minimum and maximum limits to PODs

Resources are requested per container, not per Pod. The total resources requested by the Pod is the sum of all resources requested by all containers in the Pod. This makes sense considering that different containers within a pod may have different resource requirements.

Following is a modified version of the yaml that we used for pod creation. It has additional fields that specify the limits for its containers:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-nginx
```

```
spec:
  containers:
  - name: nginx
    image: nginx:1.22.1
    resources:
      requests:
         memory: 100Mi
         cpu: “0.5”
      limits:
        memory: 200Mi
        cpu: “1”
  ports:
  - containerPort: 80
```

As the name indicates, limits specify the maximum memory or CPU that can be allocated to the container. Similarly, requests specify how much of the memory and the CPU are requested for the container.

The unit used here for memory is Mi. 1 Mi Bytes, also called as Mebibytes, is equal to 2^20 bytes or 1024 kilobytes. Similarly, 1 Gi Bytes is equal to 2^30 bytes or 1024 Mebibytes.

The unit used for CPU here is just a number passed as string, and it indicates the number of CPU units requested by the container or limited by the specification. You can also specify requests and limits in the terms of miliCPUs. Following is an example where 500 miliCPU is specified both as a request and limit:

```
resources:
  limits:
    cpu: 500m
  requests:
    cpu: 500m
```

You can use the **`kubectl top`** command to find the memory and CPU usage of your pods. *Figure 2.12* shows how the sample output looks like for the `top` command:

```
D:\>kubectl top pod my-redis
NAME       CPU(cores)   MEMORY(bytes)
my-redis   2m           7Mi

D:\>kubectl top pod my-nginx
NAME       CPU(cores)   MEMORY(bytes)
my-nginx   0m           9Mi
```

Figure 2.12: *Resource usage of pods*

These usage numbers are collected from metrics-server deployment running in your cluster. Metrics Server is another object in Kubernetes world that is used specifically for autoscaling pods in the cluster.

It is the source of container resource metrics for autoscaling pipelines in Kubernetes.

Metrics Server collects resource metrics from Kubelets and exposes them in Kubernetes apiserver through Metrics API. These metrics are then used by Horizontal Pod Autoscaler and Vertical Pod Autoscaler objects in Kubernetes. Kubelet, which was introduced in *Chapter 1, Docker and Kubernetes 101*, is an agent that runs on every node in Kubernetes cluster and helps run the system.

The quickest way to access metrics API is using the **`kubectl top`** command, as shown earlier. This gives you enough details to debug and determine whether the autoscaling for your application is working as expected. If the metrics-server is not installed or running in your cluster, the **`kubectl top`** command will not work as expected, and you need to make sure the metrics-server is running in your cluster to fetch these resource details through the **`kubectl`** command.

What do you think happens when a container tries to request more memory than the limit specified in the yaml specification? If you think that the pod creation fails, you are partially wrong. The pod creation succeeds if the memory allocated is sufficient to boot the pod, and the container in the pod runs fine as long as it manages with the memory less than the limit specified. If the process in the running container tries to consume more memory than the specified limit, then the process is terminated with an **`Out-of-Memory`** error. Thus, the memory allocated to the container is throttled, and the limit specified cannot be crossed.

Similarly, for the CPU requests, if the CPU required for the container application is more than requests limit but below the specified limit, and if the CPU is available, then it is made available for the container application. The difference between CPU allocation and memory allocation is that the memory once allocated cannot be pulled back from the application. However, CPU allocation to a container application can be reduced in case a new container comes up and the existing CPU needs to be shared. For example, consider a scenario where there are two pods running on a node, and each of them is using one CPU. When a third pod comes up, which has similar resource requests, then the two available CPU units will be equally distributed between 3 pods. After some time, if the two pods are successfully terminated, then the remaining pod can use the two CPU units completely for its container(s).

Data persistence

If you have an application where the container writes the data to its local filesystem, then it works fine as long as the container is running. But if the container crashes or pod crashes, or the node on which the pod is running dies, then all the data is gone.

This is because the container and the storage it uses are tightly coupled; in other words, they are one and the same.

This has an advantage that the application's data is not left around when the application itself is gone. *Figure 2.13* features data inside a Pod:

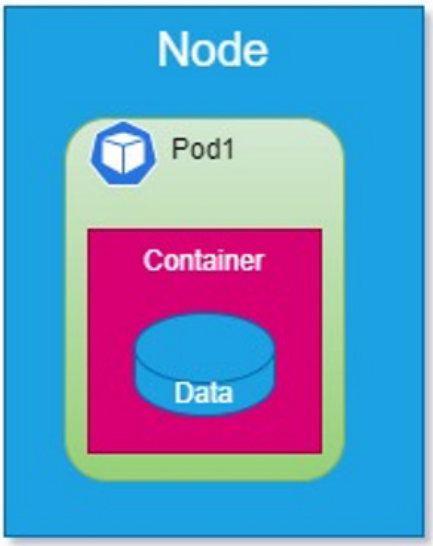


Figure 2.13: *Data inside Pod*

However, because containers can crash, you do not want to lose the data due to a bug in your software application. When **`kubelet`** restarts the container, it starts the container with an empty slate. So, the better way is to make sure the life cycle of the data is not tied to the life cycle of the container. We need to ensure that the data persists even when the container is dead. Let us look at couple of ways to do it.

Internal: Using data volumes with PODs

You can add an **`emptyDir`** volume to the pod in its specification itself. This type of volume is created when a pod is allocated in a Node and the volume is retained along the life cycle of the pod. When the pod containing the volume is removed from the node, the data in the **`emptyDir`** is cleaned.

This volume is empty upon creation, hence the name. All containers in the pod can use the volume and share the data. Let us look at a sample yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-redis
```

```
spec:
  volumes:
  - name: cache-volume
    emptyDir:
      sizeLimit: 500Mi
  containers:
  - image: redis
    name: redis-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
```

Here, **spec.volumes** indicates that the **emptyDir** volume of size 500 Mebibytes is created with the name **cache-volume**. The **redis-container** uses the volume by mounting it at a path /cache.

Different containers in the pod can mount the volume at different paths.

Another type of volume that can be used is **hostPath**. This kind of volume has some security risks associated with it, as the filesystem from the parent node is used.

A **hostPath** volume mounts a file or directory from the node's filesystem into a Pod. This is not something that most Pods need, but it offers a good hack for some applications. One of the differentiators between **hostPath** and **emptyDir** is that the **hostPath** volume can be created before the pod, and you can specify whether you want a specific directory or file to exist before the Pod is running through the **hostPath,type** specification.

Following is a sample yaml for **hostPath** volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-redis
spec:
  containers:
  - image: redis
    name: redis-container
    volumeMounts:
    - mountPath: /cache
      name: cache-volume
```

```
  volumes:
  - name: cache-volume
    hostPath:
      path: /datacache
      type: Directory
```

Here, the optional field **`type`** set as **`Directory`** specifies that the **`datacache`** directory should exist or else the Pod creation fails.

External: Data on remote disks

You can mount data on remote network disks into your pods. Kubernetes takes care of mounting and unmounting the volume from remote disk when starting and terminating the pod. The content in such volumes is retained even when the pods are removed. This ensures that even if the pod is restarted on a different host machine, it has the required data, which is not the case when you use **`emptyDir`** or **`hostPath`** volumes.

Kubernetes supports standard protocols like **Network File System** (**NFS**) and iSCSI for mounting data on remote network storage. Even for many public and private cloud providers, Kubernetes has support to mount the volumes using their storage-based APIs.

If you have an NFS server or iSCSI server running with a share exported, you can create pods where the shared volume is mounted into your Pod.

Following is a sample yaml to mount volume using NFS:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-redis
spec:
  volumes:
  - name: test-nfs-volume
    nfs:
      server: test-nfs-server.example.com
      path: /test-nfs-volume
  containers:
  - image: redis
    name: redis-container
```

```
    volumeMounts:
    - mountPath: /test-nfs-data
      name: test-nfs-volume
```

Here, the **test-nfs-volume** is mounted at **test-nfs-data** for the Pod.

We will be discussing more storage options for Pods, such as **PersistentVolumes** and **StatefulSets**, in detail in *Chapter 6, Configuring Storage With Kubernetes*, which is dedicated to storage.

Health checks

Health checks are extremely important for pods, just like they are for humans. There are two types of health checks that we are going to look at: one that tells the high-level health for the pod and the other one that checks whether a pod is ready for the action, that is, to serve incoming traffic.

Kubernetes, kubelet in particular, uses probes to determine the health of the Pod containers. Probes are the diagnostics that are performed periodically. There are three types of probes that we are going to look at.

Startup probe

A startup probe tells kubelet whether or not the application inside the pod container has started. If this probe is setup, this is given the highest priority and liveness; readiness probes are not run until the startup probe succeeds.

Following is a sample **startupProbe** that you can specify in your pod specifications:

```
startupProbe:
   httpGet:
     path: /health
     port: 8081
   failureThreshold: 12
   periodSeconds: 10
```

Startup probes are useful specially for the applications that require high startup time.

In the preceding example, the container application is given 2 mins (120 seconds) of startup time because **kubelet** will check whether or not the application has started every 10 seconds for 12 times.

Liveness probe

A liveness probe talks about the health of the pod container. It asks whether the pod is sick, dead or alive. Following is a sample liveness probe that you can specify in your pod specifications:

```
livenessProbe:
   httpGet:
        path: /health
        port: 8080
  initialDelaySeconds: 5
  periodSeconds: 3
```

Kubelet sends an HTTP **GET** request to the container application on /health path every 3 seconds, which is specified under the **periodSeconds** field. The **initialDelaySeconds** is also very important as it indicates that **kubelet** should wait for 5 seconds before performing the first probe on the pod.

If the application returns a success code, that is, 2xx or 3xx for /health path, then **kubelet** considers the container to be alive and healthy. If the application returns a failure code, the container is killed and restarted, as the probe is considered as failed.

If you want your container to be killed and restarted if a liveness probe fails, then specify a **restartPolicy** as **Always** or **OnFailure**.

Readiness probe

A readiness probe's intent is to determine whether the incoming traffic can be sent to the pod container. Following is a sample readiness probe that you can specify in your pod specifications:

```
readinessProbe:
   httpGet:
      path: /ready
      port: 8080
   initialDelaySeconds: 10
   periodSeconds: 5
```

Here, **Kubelet** sends an HTTP **GET** request to the container application on /ready path every 5 seconds. The **initialDelaySeconds** field indicates how long the **kubelet** should wait before probing the pod for the first time.

Readiness probe and liveness probe can be the same most of the time. Moreover, you can decide if just the liveness probe may suffice based on your use case, but readiness probe can be useful in cases where application may get busier handling an unusually long time-taking request. In such scenarios, if you do not want to kill the pod or send it requests either, this is where readiness probe helps notify the **kubelet** to not send further requests to the busy pod.

In the preceding examples, we have seen only **httpGet** kind of checks where an HTTP **Get** request is sent to the specified port and path on Pod's IP address. The HTTP check is considered successful when the response for the request is in 2xx or 3xx series.

There are three other kinds of checks that you can use in probes:

- **exec**: In this check, the specified command is executed inside the container. The check is considered successful if the command execution status code is 0.
- **tcpSocket**: In this check, **kubelet** tries to open a socket to your container on the specified port. If the connection is established, then the check is considered successful as the port is open.
- **Grpc**: This check is in alpha mode and available only if the **GRPCContainerProbe** feature gate is enabled. gRPC is a framework developed by Google for making remote procedural calls. If the application responds with **SERVING** as a code upon calls, then the check is considered successful.

POD security

You can set security settings for a pod or a container in it through a specification called **SecurityContext**. Let us start with the same yaml that we used earlier with additional specifications for **SecurityContext**:

```
apiVersion: v1
kind: Pod
metadata:
     name: my-redis
spec:
  securityContext:
    runAsUser: 100
    runAsGroup: 300
    fsGroup: 200
  containers:
```

```
- image: redis
  name: redis-container
  securityContext:
    allowPrivilegedEscalation: false
```

Here, the **`securityContext`** specified for pod applies to ALL the containers in the pod. Also, you can specify the container specific **`securityContext`**, as shown. The **`runAsUser`** and **`runAsGroup`** fields specify the user ID and user group used for running processes within the container, respectively. The **`fsGroup`** specifies the group ID that owns the volume and files created in the volume, if any, by the container application.

Pod Security Standards

Let us look at what are Pod Security Standards and then we will see how we can apply them using Admission Controller.

As the name suggests, the Pod Security Standards help define the pod behavior. The standards are broadly categorized into three policies or profiles: Privileged, Baseline, and Restricted. The names of the profiles indicate how stringent or relaxed they are in imposing standards. Privileged is the least restricted policy, whereas Restricted is the most restricted policy and follows best practices for pod hardening. Baseline policy lies between the two and is easy to use for common container workloads.

One of the easiest and most preferred ways to enforce security standards is to apply them at the namespace as a label. The labels applied at namespace level help enforce the security standards to all the pods within that namespace. This is very useful considering that pods pertaining to a specific application are usually put under a same Kubernetes namespace. You may want to apply different security standards for different namespaces because different applications may have different security needs.

There are three types of labels that you can apply:

- **Enforce**: With this label, pods violating the policy would be rejected.
- **Audit**: With this label, policy violations will be allowed, and audit log will have an entry detailing the event with violations.
- **Warn**: With this label, policy violations are allowed, and a warning will be shown to the user.

Following is a sample label that you can apply to a namespace, either through yaml or with the **`kubectl label`** command, to enforce baseline security policy to all pods in the namespace:

`pod-security.kubernetes.io/enforce: baseline`.

Pod Security Admissions

Before we learn about Pod Security Admissions, it is important to know that Admission Controller—yes, **one more controller** in the scheme of Kubernetes things—is the engine that helps enforce Pod Security Standards on the Pods.

Admission Controller, in simple terms, is a program that intercepts the requests to the Kubernetes API server and does validations or updates on the objects, as required. Admission controllers **allow** all sorts of observational activities, such as reading and listing, which do not change the state of the system, and any kind of modifications, like create, update and delete, to the Kubernetes objects can be controlled via Admission Controller.

Now let us understand how to apply the Pod Security Standards using an Admission Controller through a very basic yaml file. This is the last yaml you are going to see in this chapter, so let us keep it short:

```
apiVersion: apiserver.config.k8s.io/v1
kind: AdmissionConfiguration
plugins:
- name: PodSecurity
  configuration:
    apiVersion: pod-security.admission.config.k8s.io/v1
    kind: PodSecurityConfiguration
    defaults:
      enforce: "privileged"
      audit: "privileged"
      warn: "privileged"
    exemptions:
      usernames: [ “user1”, “user2” ]
      namespaces: [ “ns1”, ”ns2” ]
```

The **`PodSecurity`** plugin mentioned here is the admission controller, which acts on the creation and modification of the pod. The controller validates pods as per the specified security context and the Pod Security Standards. The usernames and namespaces that you want to exempt from security policies need to be listed in the arrays under exemptions. There are many other values that you can use under defaults and exemptions, but all of them are not listed here for the sake of simplicity.

Conclusion

Pods are the atomic unit in the world of Kubernetes. A pod provides the running environment for containers, and containers cannot exist on their own in Kubernetes. You can use a **`yaml`** file with specifications to create and update pods. **`kubectl`** is very powerful tool to create, access, update and delete pods. You can see the logs for pods, run commands in pods and start an interactive session within the pod for debugging purposes.

Containers share the resources from the pod, and you can specify requests and limits for memory and CPU through pod specifications. Containers can crash, so you can store the data required for application using volumes. Volumes can point to a storage inside pod or outside of the pod. You can run health checks on pods to identify their liveness and readiness, and you can define the security for pods using Pod security contexts and Pod Security Standards.

Points to remember

- The **`restartPolicy`** specified for pod defines the behavior for container creations after they are terminated.
- You can use the **`kubectl`** port-forward command to forward requests from a port on your local machine to a port on a specific pod in a Kubernetes cluster.
- The **`kubectl exec`** command with flag is used to start an interactive session within the pod.
- Requests and limits specified in the pod specification apply to the containers, and attempts to access more resource than the limit fail.
- The **`emptyDir`** volume creates an empty directory upon pod creation and is the easiest way to persist data in a pod. The data in **`emptyDir`** is retained across container start, crash, and restart.
- You can connect Pods to a network storage using standard protocols like NFS and iSCSI.
- Liveness probe is used to check whether or not a container is alive, while readiness probe is used to decide whether or not a container is ready to serve the incoming requests.

Questions

1. The **`kubectl port-forward pod-name 8080:80`** command is not working as expected for a pod. How will you find the root cause for this?

2. A container in the application is failing after running for a certain period. The logs indicate that the application is throwing an **`OutOfMemory`** error. What would be your suggestion to fix the crashing application?
3. Your container application is terminating again and again. How will you see the logs of the terminated application?

Answers

1. Check the pod **`yaml`** to check whether the container application is using port 80 or some other port. If the port is correct, check whether the application is running successfully by looking at the logs.
2. Increase the memory limit for the container application or analyze why the application is requesting more memory than the limit.
3. Use the **`kubectl logs pod-name --previous`** command to see the logs for a terminated instance of a container.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 3
HTTP Load Balancing with Ingress

Introduction

Pods, the main building block of Kubernetes, come and go. This could be due to failures, for example, the node becoming unreachable, or it could be due to the application's apparent scaling up and down. Pods are recognized (unsurprisingly) in the K8S cluster by IP address; hence, it becomes crucial to understand how the Kubernetes cluster adds or deletes pods (with IP) to and from the scheme of things, respectively, and how a newly created pod becomes part of the workflows.

The critical part for any application is how it manages network traffic to (egress) and from other applications/entities, inside and outside the closed network, where the application is deployed. Kubernetes calls its HTTP-based load-balancing system *Ingress*. This chapter will discuss the Ingress in detail and cover additional topics around other networking constructs in K8S.

Structure

In this chapter, we will discuss the following topics:

- Networking 101
    - Configuring Kubeproxy
    - Configuring container network interfaces

- Ingress specifications and Ingress controller
- Effective Ingress usage
- Advanced Ingress
    - Running and managing multiple Ingress controller
    - Ingress and namespaces
    - Path rewriting
    - Serving TLS
- Alternate implementations – NGINX, Gloo, Ambassador, Traefik
- API gateways
    - Need for API gateways
- Securing network
    - Via network policies
    - Via third-party tool
- Best Practices for securing a network
    - Handling network upgrades

Objectives

In this chapter, you will learn how the traffic flows in and out of applications deployed in Kubernetes and various abstractions\Kubernetes constructs (objects) available, that facilitate the scale up and down of an application possible without compromising on the reliability and security of applications. You will also learn how to manage the network's security and how to apply network governance in your deployments. Finally, you will see a discussion about how to upgrade the network with minimal downtimes.

Networking 101

In this section, you will see the networking processes that run under the hood while creating a pod. In the previous chapters, you investigated how an image is created and pushed to the container registry. When there is a need to spin up a new pod, the image is pulled from the container registry, and a pod is created. This is done via the container runtime environment or CRE (for example, docker). The obvious question that comes to mind is, how does this newly created pod get an IP address, and how will this IP address be accessible by other entities in the Kubernetes cluster?

Since all pods need to have a unique IP address in the cluster, it is vital to ensure that each node in the cluster has its unique subnet assigned, from which the pods

are assigned IP addresses on that node. A node is assigned this subnet range when the same node is first registered to the cluster. Refer to *Figure 1.7, Kubernetes Master*, from *Chapter 1, Docker and Kubernetes 101*. The master node contains a controller component, which has various daemons running in a control loop, ensuring the resource allocations' state in the cluster and controlling the IP allocation. When the *nodeipam* is passed as an argument to the controller manager, the --controllers command line flag, it assigns each node a dedicated range of IP address from cluster CIDR. These assignments of IP ranges across nodes are disjoint sets; and hence, each pod on a node is assigned a unique IP address across the cluster. The **`podCIDR / .spec.podCIDR`** range could be listed using the following **`kubectl`** command:

```
$ kubectl get no <nodeName> -o json | jq '.spec.podCIDR'
```

Now since you know how an IP is assigned to a pod, let us see how a connection is created between a container running in the pod and other entities in the cluster. For this, Container Network Interface comes into the picture.

The **Container Network Interface** (**CNI**) is an interface between the network and the **Container Runtime Interface** (**CRI**), and it configures network routes. Consider *Figure 3.1*, which features the role of CNI:

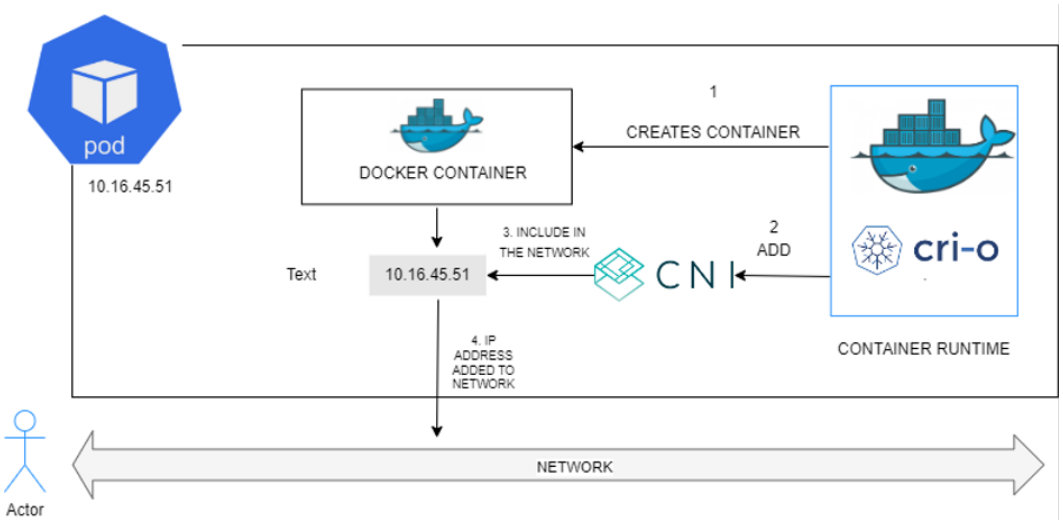


Figure 3.1: *Role of CNI*

Refer to the numerical labeling in *Figure 3.1* with the following corresponding numerical explanations:

1. The container Runtime interface creates a container where the application code will be up and running.
2. GitHub says CNI calls it as operations, and the strings are passed as a value of the **`CNI_COMMAND ENV`** var. CNI defines four operations: **`ADD`**, **`DEL`**, **`CHECK`**, and **`VERSION`**.

3. CNI triggers the command to include the IP address (**`10.16.45.51`** above) assigned to the container into the network.
4. The IP address is added to the Kubernetes cluster network and is now accessible to other actors in the system.

Kubernetes node and pod setup are not simple, and multiple things are happening under the hood. The container inclusion on the network includes all the aspects taken care of, and hence, the CNI's role to **`INCLUDE IN THE NETWORK`** (labeled as 3 in *Figure 3.1*), is not a single-step process. A few of the steps included are as follows:

1. Executing bridge plugin
2. Executing the ipvlan plugin
3. Executing static plugins, and so on

There are a few key Kubernetes network requirements, and any CNI (among multiple available, such as Weaveworks, Calico, Antrea, and Flannel) that caters to those requirements will be an excellent fit to be used as CNI. The Kubernetes network requirements are as follows:

1. Each pod gets its IP, and a container within the pod shares the namespace. If your pod has an IP address **`10.16.45.51`** (in *Figure 3.1*) and contains two containers, then the IP address will be shared by both containers, and the differentiating factor will be the port. So, **`10.16.45.51/80`** and **`10.16.45.51/90`** could be valid configurations.
2. All PODs communicate with other PODs without **Network Address Translation** (**NAT**).
3. All nodes can communicate with other nodes without NAT.
4. The IP of the POD is the same throughout the cluster.

As stated, **NAT** stands for **Network Address Translation**, and it is a way to map multiple local devices' private addresses to a public IP address before transferring information. For example, in most organizations, multiple local devices employ a single IP address. Another example of this could be your home router.

Let us now jump out of the Kubernetes pod and discuss networking, since pods have a unique IP address, an application, or an entity willing to communicate with the pods using the IP address in communication. However, these pods come and go, so communication established like this can work for some time but will eventually fail. To overcome this, Kubernetes brings in an abstraction layer known as service. Consider *Figure 3.2* to see a feature of Real-world Kubernetes Application:

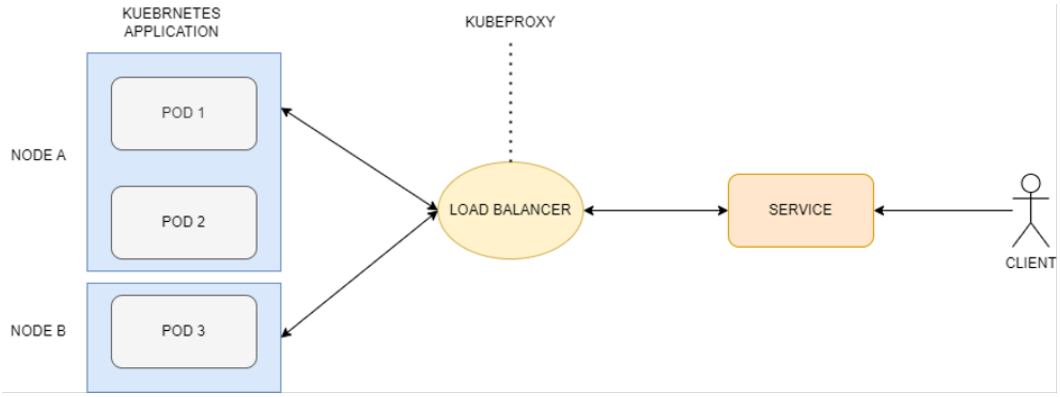


Figure 3.2: *Real world Kubernetes Application*

The preceding diagram represents a real-world Kubernetes application setup. The diagram shows that the application has three pods (**Pod 1**, **Pod 2**, and **Pod 3**) running on two nodes: Node A and Node B. These pods are abstracted from the clients using the Kubernetes service. Kubernetes service uses a load balancer behind the scenes to direct traffic to different pods. The advantage of this setup is that the ephemeral nature of pods will not disrupt the working, creation and deletion of pod its handling of network resources is internal to abstraction, that is, Kubernetes Service. As discussed in *Figure 1.8, Kubernetes Worker* from *Chapter 1, Docker and Kubernetes 101*, and its explanation, Kubeproxy manages and maintains the network rules on each worker node. These network rules facilitate communication across your pods from a client inside or outside the Kubernetes cluster.

Note that the actual picture of real-world use cases is much more complex, and you will eventually reach there by the end of the chapter. But with this discussion in place, let us investigate some key aspects to be considered while configuring CNI and Kubeproxy.

Configuring Kubeproxy

Kubeproxy keeps an eye on the control plane of Kubernetes and installs network rules for each service on the node, where Kubeproxy is running. If you have a cluster with five nodes, each node will run its Kubeproxy daemons; all these defined Kubeproxy daemons will listen to the control plane for pulling information to change and configure network rules. The information is pulled and updated whenever there is a change in the service. Kubeproxy realizes in readiness probe to determine whether a backend pod is healthy and could be used to route traffic.

Kubeproxy could be configured in iptables (default) and IPVS mode. Kubeproxy aims to identify one pod where an incoming request must be directed. There could

be multiple data structures and strategies to identify the real IP address. IP table way of configuring Kubeproxy generally suffices in most situations. However, when the number of deployments increases, identifying the destination IP address of a pod becomes slow, so IPVS comes to the rescue. It uses the hash table as a data structure and supports almost unlimited scale under the hood.

IP tables support a round-robin way of identifying the pod IP; however, IPVS supports multiple other algorithms. These algorithms are **round-robin** (**rr**), **least connection** (**lc**), **destination hashing** (**DH**), **source hashing** (**sh**), **shortest expected delay** (**sed**), and **never queue** (**nq**).

Kubeproxy is a daemon running as pods in the Kubernetes cluster. To see all the pods running as Kubeproxy in your cluster, use the following command:

```
$ kubectl get pods -n kube-system | grep kube-proxy
```

To update the mode from IP tables to IPVS, edit the config map using the following command and update the property mode to IPVS and apply the yaml. Details of config map will be covered in the subsequent chapters.

```
$ kubectl edit configmap kube-proxy -n kube-system
```

Change the `mode` from **“”** to `ipvs`:

```
mode: ipvs
```

Restart the Kubeproxy pods:

```
$ kubectl get po -n kube-system

$ kubectl delete po -n kube-system <pod-name>
```

The preceding command just deletes the existing pods, and kube-proxy pods come up automatically as the system restores the state for deleted pods.

Configuring container network interfaces

Kubernetes’ default CNI is kubenet, an oversimplified CNI that works with multiple on-premises and public cloud setups. Kubenet is a basic network provider and does not have important features like egress control, BGP (Border Gateway Protocol) and mesh networking. Limitation on number of nodes in a Kubernetes cluster makes it unfavorable for production deployments.

Choosing a CNI is one of the most asked questions in community, and there are multiple successful examples available where the following CNI were used:

- Calico
- Canal (Flannel + Calico)
- flannel
- kopeio-vxlan

- kube-router
- romana
- Weave Net

All these CNI providers use a daemon set installation model, where their product deploys a Kubernetes Daemon set. Just use kubectl to install the provider on the master once the K8s API server has started. Refer to each project's specific documentation.

For example, installing Calico as network provider is a three-step process:

```
$ kubectl create -f https://raw.githubusercontent.com/projectcalico/
calico/v3.24.5/manifests/tigera-operator.yaml

$ curl https://raw.githubusercontent.com/projectcalico/calico/v3.24.5/
manifests/custom-resources.yaml -O

$ kubectl create -f custom-resources.yaml
```

One must analyse the various tools available and finalize one which suits the requirements. Broad areas that need to be evaluated are Network Model, Route Distribution, Network policies, External datastore, Encryption, Ingres/Egress policies, and Commercial Support.

Ingress specifications and Ingress controller

In *Figure 3.2*, you observed the need to have a Kubernetes service in front of a collection of pods to serve the traffic coming in from outside the cluster. Let us revisit *Figure 3.2* with a few changes, as *Figure 3.3*, where real-world Kubernetes application is exposed to the Internet, with some additional elements to expose the service over web:

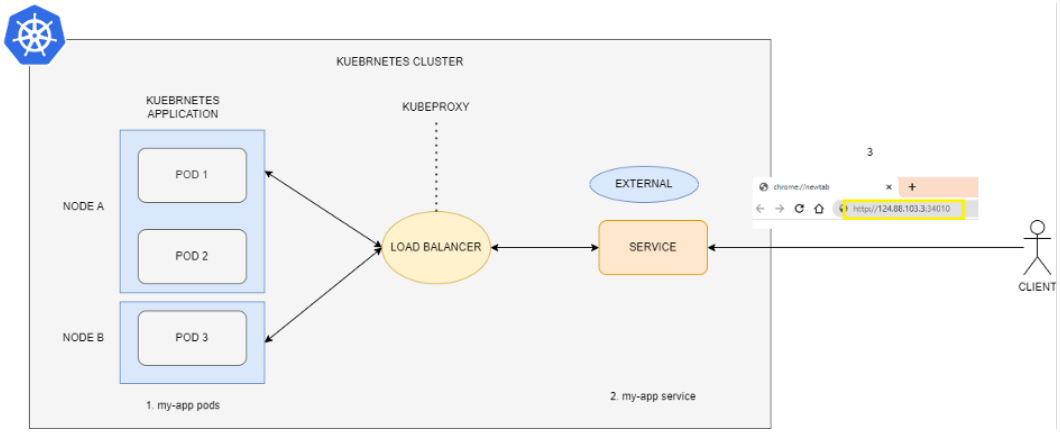


Figure 3.3: *Real world Kubernetes Application exposed to Internet*

In the preceding figure, **POD 1**, **POD 2**, and **POD 3** represent the pods of **`my-app`** (labelled as 1) application. This **`my-app`** application is exposed to external HTTP traffic using an external service **`my-app`** service (labelled as 2). An outside entity (a user persona or other external application) can interact with the **`my-app`** external service either by calling REST API or via a a browser (labelled as 3). If you observe the highlighted area in the browser, an external HTTP entity used IP and Port of the service, which might be alright in case of test cases and experiments, but it is far from how the final product should look. Ideally, the external entities should interact via HTTPs and should be using hostname in place of IP.

Consider *Figure 3.4*, which introduces the concept of Ingress:

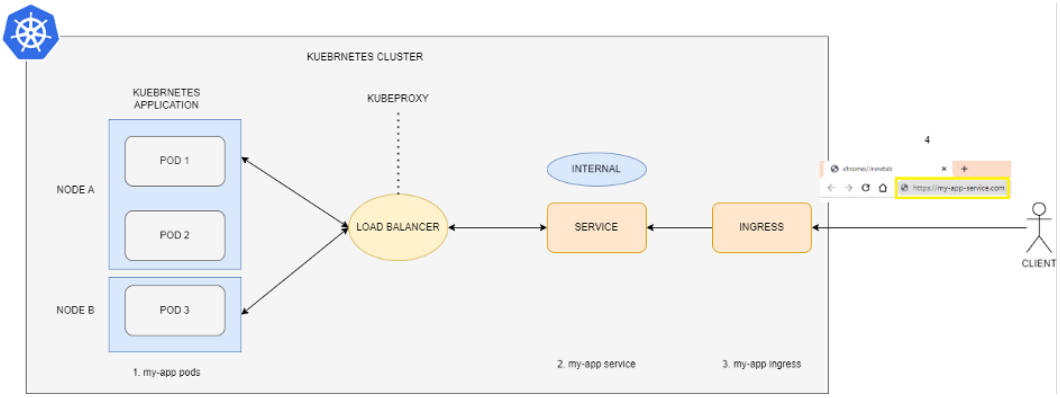


Figure 3.4: *Kubernetes Application exposed via Ingress*

In the preceding figure, Label 1 remains the same, that is, **`my-app`** pods. My-app service, which was an external service in *Figure 3.3*, is now an internal service (labelled as 2). The internal service has now been called by Ingress (labelled as 3). An external HTTP entity can now call the application using the host name over HTTPS by using Ingress specifications.

You had already seen, in the previous chapter, how to create pods from docker images. In this chapter, you will see how to expose the pod first as a service and then wrap it using ingress. For this exercise, refer to codebase. A prerequisite is to have the **`my-app`** image available locally. To build the image, go to the app folder and trigger command and follow the given steps:

1. Build docker image locally:

   ```
   docker build -t my-app:1.0.1
   ```

2. Deploy a Pod. Go to the root folder of codebase:

   ```
   kubectl apply -f pod.yaml
   ```

   All constructs in **`pod.yaml`** are already explained in Chapter 2, PODs.

3. Create a service using the **`external-service.yaml`**. This yaml creates a load balancer service:

   ```
   kubectl apply -f external-service.yaml
   ```

   The following Kubectl command will fetch all the services deployed on your cluster:

   ```
   kubectl get services |grep my-app
   ```

You will get the output shown in *Figure 3.5:*



Figure 3.5: *Output*

You can use the highlighted IP, that is, 35.193.65.79, and the port **`30010`** to access the **`my-app-service`** service.

The **`external-service.yaml`** file looks as shown in *Figure 3.6*:

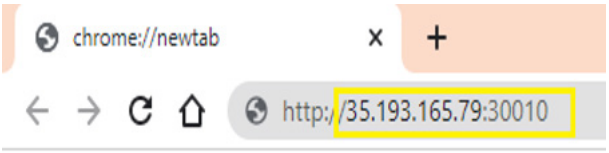


Figure 3.6: *How the file looks*

Refer to the following code:

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: my-app-service
5. spec:
6.   selector:
7.     app: my-app
8.   ports:
9.   - protocol: "TCP"
10.    port: 6000
11.    targetPort: 5000
12.    nodePort: 30010
13.  type: LoadBalancer
```

Line number 2 specifies the type of Kubernetes object as Service, and line number 13 specifies a service type of **`LoadBalancer`**. This means you are creating an external IP address to the service. Line number 12 is the port number on which the user can access the application.

Let us now dig deep down into how a service identifies which pod belongs to the service and how will a service point to one of the containers inside a pod (having multiple containers).

Consider *Figure 3.7*, which features service IP and port mapping; you will see the same image discussed previously, with a few more complexities added:

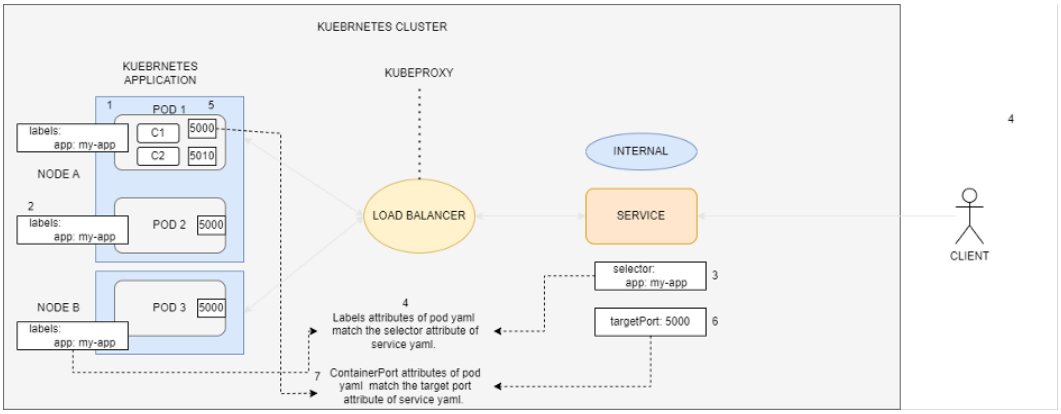


Figure 3.7: *Service IP and port mapping*

Refer to the numerical labeling in *Figure 3.7* with the following corresponding numerical explanations:

1. POD 1 has multiple containers – C1 & C2.
2. Each pod definition contains a label attribute. Refer to line 12 in **`pod.yaml`**.
3. Each service definition file contains selector attribute. Refer to line 6 in **`external-service.yaml`**.
4. Labels attributes of pod **`yaml.`** match the selector attribute of service yaml. A service will include all the pods in the cluster that match the selector attribute. Since pod 1,2 & 3, all have same label, that is, **`my-app`**, the **`my-app-service`** selector attribute will include all of them as part of the service.
5. In pod 1, there are two containers running at port **`5000`** and **`5010`**, mentioned as the **`containerPort`** attribute in **`pod.yaml`** (line number 20). How will the service know which exact container to direct traffic to?
6. Service definition yaml contains the **`targetPort`** attribute. Refer to line 11 in external-**`service.yaml`**.

7. Container Port attributes of pod **`yaml.`** match the target port attribute of service **`yaml.`**

   Load balancer service is just one of the services you can configure among multiple other options available, such as **`ClusterIP`** and **`NodePort`**. The intention of each of the option is the same: to allow external HTTP traffic to Kubernetes cluster.

8. Till now, you had seen how to create a service and counter the ephemeral nature of pods. Service objects operate at Layer 4 (as per the OSI model). This means that a service can only forward a TCP of UDP connection and does not look inside the connection. Generally, a Kubernetes cluster contains multiple services, so you must open multiple services to the outside world. For HTTP (layer 7) traffic, Kubernetes has Ingress. There are two parts in Ingress setup. Ingress configuration is done using a standard construct provided by Kubernetes. On the other hand, the implementation, that is, the software component that reads the ingress configurations and route HTTP traffic is not provided out of the box. Hence, you need to set one up for your cluster.

   Ingress provides a single hostname, which could be used by all external HTTP traffic, and based on attributes like hostname or URL, ingress controller directs the incoming traffic to the appropriate service. Ingress controllers also act as the entry point to the cluster. Moreover, one more thing to keep in mind while setting up ingress is that the services that are abstracted away behind the ingress are internal services, though this is not a mandatory step. The only difference between an internal and external service is that internal services have no node port attribute mentioned. Furthermore, the type of service must be default, that is, of the **`ClusterIP`** type. Refer to the **`internal-service.yaml`** file present in the codebase. Deploy the **`internal-service.yaml`** using the following command and then use it in defining ingress:

   ```
   $ kubectl apply -f internal-service.yaml
   ```

   There are multiple implementations of ingress controllers available, such as **`contour`** and **`nginx`**. You already got a glimpse of **`nginx`** in *Chapter 2, PODs*. It is time to investigate its role.

Installing Contour

```
$ kubectl apply -f https://j.hept.io/contour-deployment-rbac
```

The preceding command needs to be executed by cluster admins. This one line works for most configurations. It creates a namespace called **`heptiocontour`**. Inside that namespace, it creates a deployment (with two replicas) and an external-facing service of type **`LoadBalancer`**.

Installing Nginx Controller

```
$ kubectl apply -f  https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.5.1/deploy/static/provider/cloud/deploy.yaml
```

The preceding command creates a service of type load balancer and creates a serving pod with replication factor 1. We will continue with nginx controller for hands-on exercises in this book.

Once the **nginx** controller is installed, you can get the external IP with the following command:

```
$ kubectl get -n <namespace> service nginx-ingress  -o wide
```

The preceding command gives the following output:

```
NAME            TYPE        CLUSTER-IP      EXTERNAL-IP    PORT(S)
AGE     SELECTOR

nginx-ingress   ClusterIP   10.51.243.143   35.232.66.22   443/TCP,80/
TCP   1d   run=nginx-ingress
```

Consider the EXTERNAL-IP column. Different public cloud providers have different config for **external-ip**. For example, AWS configures host name, while GCP and Azure uses an IP address. If you are using **minikube**, you will probably not have anything listed for EXTERNAL-IP. To fix this, you need to open a separate terminal window and run **minikube** tunnel and carry out the following steps:

- **Configuring DNS:** For ingress to work well, you need to configure DNS entry for the external address of your **ClusterIP** service. You can map multiple hostnames to a single external endpoint, and the Ingress controller will play traffic cop and direct incoming requests to the appropriate upstream service based on the hostname. For this chapter, we will assume that you have a domain called **handon-k8s.com**. You need to configure two DNS entries: **tiger.handon-k8s.com** and **lion.handon-k8s.com**. If you have an IP address for your external **ClusterIP** service, you will want to create A records. If you have a hostname, you will want to configure **CNAME** records.

- **Configuring a local hosts file:** If you do not have a domain or if you are running Kubernetes locally using Minikube, you can configure the local host file to add in IP addresses. Put the following entry in the local hosts file:

  ```
  <ip-address> tiger.handon-k8s.com  lion.handon-k8s.com
  ```

  Ip-address is the same **external-ip** address that we got while describing the nginx controller service.

  You are now all set to define the first ingress in its simplest form. Refer to the **ingress.yaml** file in the codebase:

```
1. apiVersion: networking.k8s.io/v1
2. kind: Ingress
3. metadata:
4.   name: my-app-ingress
5. spec:
6.   backend:
7.     serviceName: my-app-service
8.     servicePort: 6000
```

The service configured here is **my-app-service** (line number 7), and the service port is **6000** (line 8), which is the port mentioned in **external-service.yaml** at line number 10.

Apply the preceding yaml configuration to create a simplest ingress with just one service configured as back end:

```
$ kubectl apply -f ingress.yaml
```

In the next section, you will investigate effective strategies and ways to configure a production level setup.

You can verify the setup using the **kubectl get** and **describe** commands:

```
$ kubectl get ingress
$ kubectl describe ingress my-app-ingress
```

The preceding describe command gives the following output:

```
Name:             my-app-ingress
Labels:           <none>
Namespace:        default
Address:
Default backend:  my-app-service:6000 (10.48.22.218:5000)
Rules:
  Host        Path  Backends
  ----        ----  --------
  *           *     my-app-service:6000 (10.48.22.218:5000)
Annotations:  <none>
Events:
  Type     Reason  Age                   From           Message
  ----     ------  ----                  ----           -------
  Normal   Sync    2m13s (x3 over 5m42s)  loadbalancer-controller
Scheduled for sync
```

Looking at the highlighted sections confirms that there is just one default back-end service configured.

Effective Ingress usage

In this section, you will dive deep into some practices around Ingress, which are generally used in production environments. As discussed, Ingress's primary aim is to redirect requests based on their properties.

Utilizing hostnames

One of the most common scenarios is to look at the HTTP host header and direct the traffic based on the header. Let us add another ingress object for redirecting traffic to my-app-service for any traffic directed to **`tiger.handon-k8s.com`**. Consider the following YAML file, which demonstrates this concept:

```
1. apiVersion: networking.k8s.io/v1
2. kind: Ingress
3. metadata:
4.   name: myapp-host-ingress
5. spec:
6.   rules:
7.   - host: tiger.handon-k8s.com
8.   http:
9.     paths:
10.    - backend:
11.        serviceName: my-app-service
12.        servicePort: 6000
```

Line number 7 is where you can see the configuration related to host name. The same file is available in the codebase by the name **`ingress-host.yaml`**.

When you describe the preceding ingress definition using **`kubectl`**, you will get the following output:

```
$ kubectl describe ingress myapp-host-ingress


Name:             myapp-host-ingress
Labels:           <none>
Namespace:        default
Address:
```

```
Rules:
  Host                  Path  Backends
  ----                  ----  --------
  tiger.handon-k8s.com
                           my-app-service:6000 (10.48.22.218:5000)
Annotations:            <none>
Events:
```

As you can see in the highlighted section, **tiger.handso.k8s.com** is the host name, and it directs traffic to **my-app-service** at port **6000**.

Utilizing paths

Another common and interesting situation is to direct traffic not just based on hostname but also based on the path. You can do this by simply specifying a path in the paths entry. For the same example above, suppose you had deployed another my-app service named **my-app-service-2**. Consider the following yaml configuration:

```
1. apiVersion: networking.k8s.io/v1
2. kind: Ingress
3. metadata:
4.   name: myapp-path-ingress
5. spec:
6.   rules:
7.   - host: tiger.handon-k8s.com
8.   http:
9.     paths:
10. - path: "/"
11.      backend:
12.        serviceName: my-app-service
13.        servicePort: 6000
14. - path: "/a/"
15.   backend:
16.        serviceName: my-app-service-2
17.        servicePort: 6000
```

Pay attention to line numbers 10 and 14, where different URL paths redirect the traffic to different services in the back end. When there are multiple paths on the same host listed in the Ingress system, the longest prefix matches. In the preceding

example, a **tiger.handon-k8s.com/** request will direct traffic to **my-app-service,** and **tiger.handon-k8s.com/a/** will direct the traffic to **my-app-service-2**.

The code can be referred in codebase in the file ingress: **url-path.yaml**.

Advanced Ingress

The extent and maturity of the feature will depend on the type of Ingress controller configured. This section will not compare feature maturity in various ingress controllers but will talk about features in general.

Running and managing multiple Ingress controllers

There is often a need to configure multiple Ingress controllers, for example, when you want to have Nginx and Contour running together. You can install both, and while defining the ingress object, remember to mention the **kubernetes.io/ingress.class** attribute.

If the **kubernetes.io/ingress.class** annotation is missing, the behavior is undefined. Multiple controllers will likely fight to satisfy the Ingress and write the status field of the Ingress objects.

Ingress and namespaces

Due to security concerns, Ingress defined in a particular namespace can only direct traffic to services deployed in its namespace. However, multiple Ingress objects in different namespaces can specify subpaths for the same host. These Ingress objects are then merged to come up with the final config for the Ingress controller.

This cross-namespace behavior means that Ingress must be coordinated globally across the cluster. If not appropriately coordinated, an Ingress object in one namespace could cause problems (and undefined behavior) in other namespaces.

Path rewriting

Not all, but many controller implementations allow path rewriting. Using path rewriting or not depends on the use case, but if you know that your ingress controller implementation supports path rewriting, for example, Nginx controller does it, you must mention **nginx.ingress.kubernetes.io/rewrite-target: /** in the annotation section of the Ingress rules. Consider the following configuration:

```
1. apiVersion: networking.k8s.io/v1
2. kind: Ingress
3. metadata:
```

```
4.   annotations:
5.     nginx.ingress.kubernetes.io/rewrite-target: /$2
6.   name: rewrite
7.   namespace: default
8. spec:
9.   ingressClassName: nginx
10.  rules:
11.  - host: tiger.handson-k8s.com
12.    http:
13.      paths:
14.      - path: /action(/|$)(.*)
15.        pathType: Prefix
16.        backend:
17.          service:
18.            name: http-svc
19.            port:
20.              number: 80
```

Just as **tiger.handson-k8s.com/action** is rewritten to **tiger.handson-k8s.com/**, **tiger.handson-k8s.com/action/new** is rewritten to **tiger.handson-k8s.com/new**.

Refer to line number 5, which has the path rewrite setup in place.

Serving TLS

Ingress supports both TLS and HTTPS for traffic coming from outside the cluster. The first step is to specify a secret with TLS certificate and keys. You can create secret by using the **kubectl apply** on the following yaml configurations (refer to the **tls-secret.yaml** file in codebase):

```
1. apiVersion: v1
2. kind: Secret
3. metadata:
4. creationTimestamp: null
5.   name: tls-secret-name
6.   type: kubernetes.io/tls
7. data:
8.   tls.crt: <base64 encoded certificate>
9.   tls.key: <base64 encoded private key>
```

Once the secrets are created, you can use the following (refer to the **`ingress-tls-secret.yaml`** file in codebase):

```
1. apiVersion: extensions/v1beta1
2. kind: Ingress
3. metadata:
4.   name: myapp-host-ingress
5. spec:
6.   tls:
7.     - hosts:
8.       - tiger.handon-k8s.com
9.       secretName: tls-secret-name
10.  rules:
11.  - host: tiger.handon-k8s.com
12.    http:
13.      paths:
14.      - backend:
15.          serviceName: my-app-service
16.          servicePort: 6000
```

Refer to line number 9, which specifies to use certification for any communication with host **`tiger.handon-k8s.com`**.

Managing TLS secrets can be difficult, specially when the team is not used to it. Additionally, certificates can be costly as well. There is a non-profit called **Let's Encrypt**, running a free of cost certificate authority that is API-driven. Since it is API-driven, it is possible to set up a Kubernetes cluster that automatically fetches and installs TLS certificates for you.

Alternate implementations

As per our discussions till now, there are multiple options available for the ingress controller. Each option has something where the other controllers lag. Analysis and study are needed before zeroing in on one of the available options. Following is a list of a few prevalent aspects which every team will need to analyze in the probable ingress controller.

- **Protocol support**: Do you need integration with gRPC or TCP/UDP?
- **Enterprise support**: Do you need a paid (commercial/enterprise) support for a production environments or mission critical workloads.

- **Advanced features:** Are you looking for a lightweight solution or are canary deployments or circuit breakers must-haves for your use case?
- **API gateway features:** Do you need some API Gateway functionalities (for example, rate-limiting) or a pure Kubernetes Ingress?

Based on support for various features, *Table 3.1* shows a capability matrix for some of the most common and most adopted ingress controllers.

Feature	NGINX Ingress	Traefik	Contour	Ambassador	Gloo
Protocols	http/https, http2, grpc, tcp/udp	http/https, http2 (h2c), grpc, tcp, tcp+tls	http/https, http2, grpc, tcp/udp, tcp+tls	http/https, http2, grpc, tcp/udp, tcp+tls	http/https, http2, grpc, tcp, tcp+tls
Based on	nginx/nginx plus	traefik	envoy	envoy	envoy
Traffic routing	host, path, header, method, query param (all with regex expect host)	host (regex), path (regex), headers (regex), query, path prefix, method	host, path	host, path, method, header (all with regex)	host, path, method, header, query param (all with regex)
Traffic distribution	canary, a/b (routing rules), blue-green (service in the upstream)	canary, blue-green, shadowing	canary, blue-green	canary, a/b, shadowing, http headers, acl, whitelist	canary, shadowing
Upstream probes	retry, timeouts, active health checks (based on http probe for pod)*	retry, timeouts, active, circuit breaker	timeouts, active	retry, timeouts, active checks, circuit breakers	retry, timeouts, circuit breakers
Load balancing	round-robin, least-conn, ip-hash, hash, random, least-time*, sticky sessions*	weighted-round-robin, dynamic-round-robin, sticky sessions	round-robin, sticky sessions, weighted-least-request, ring hash, maglev, random	round-robin, sticky sessions, weighted-least-request, ring hash, maglev, random	round-robin, sticky sessions, least request, random
Enterprise versions with Paid support	Yes	Yes	No	Yes	Yes

Feature	NGINX Ingress	Traefik	Contour	Ambassador	Gloo
Basic DDoS protection	max-conns, rate limit, rate-limits (with custom annotations)	max-conns, rate limit, ip whitelist	max-conns, max-request	rate limit, load shedding	rate limit*

Table 3.1: *Ingress Controllers*

It is generally advisable to perform a proof of concept to close out on a particular controller. Also, these controllers are evolving every day, so readers are requested to recheck the capability when they shortlist controllers.

API gateways

When the software architectures started moving away from Monolith to Microservices and enterprises adopted Microservices, a new set of issues and complexity started appearing. A few common ones are as follows:

- **Latency**: To perform any function, such as, viewing products on an eCommerce app, a client must make several calls to different microservices and the microservice themselves make internal calls to other microservices. These calls increase the number of round trips and result in longer wait times.
- **Security**: Since each microservice is accessed via a public endpoint, this opens up your application to cybersecurity issues. Each service should ideally have authentication and authorization, but implementing it takes too much time and effort.
- **Tight coupling**: Direct communication between microservices means that client apps are tightly coupled to internal microservices, and when the latter is updated or retired, it impacts client apps too.

These areas, coupled with the scale, worsened the situation. These issues can be solved by introducing an intermediate component (middleman) between a client (entities calling microservice) and a back-end API. The primary purpose of this middleman will be to perform load balancing, provide security measures, and promote loose coupling — a gateway. Let us discuss how it does that.

Need for API gateways

An API gateway helps you decorate your workflow deployment with several additional features, which enables you to implement cross-cutting features across all your APIs.

Routing requests

An API gateway lets you expose a single endpoint to external entities and does the routing of those requests to actual services.

Cross-cutting concerns

All interactions between a client and backed API happen via the gateway, it is an excellent place to apply below category of rules and gather telemetry data:

- **Authentication and authorization**: A gateway is your first line of defense against potential attackers that can perform essential security functions: antivirus scanning, token translation, decryption and encryption, validation, and many more.
- **Log tracing and aggregation**: A gateway keeps detailed audit logs used for debugging, reporting, and analytics.
- **Rate limiting**: A gateway enforces policies against resource overuse (either accidental or deliberate) and allows you to configure API invocation at runtime, so the service is consumed only at the required rate.
- **Load balancing**: To efficiently handle requests, a gateway balances the load between nodes of service to ensure the application's availability during versioning or other changes in the service.

Translating different protocols

Most external APIs respond in REST messages, but the internal implementation might be in a completely different format. For example, legacy systems still work with SOAP protocol, or you want to benefit from gRPC. The gateway can convert REST calls into different compatible protocols without having the engineering team modify its internal architecture.

Securing network

Kubernetes relies on a collection of internal (Kubeproxy) and third-party components (CNI plugins, Ingress controllers) to manage network configuration and traffic; securing Kubernetes requires admins to leverage a mix of native and third-party tools to ensure security.

Securing via network policies

Network policies are the most native out-of-the-box constructs available in Kubernetes to secure networks. They are configurations that define rules that govern how pods interact with each other. These policies are essential from the

admin point of view, and governance could be set in place using them, defining the uniform security policy for the whole cluster. Generally, these rules are tied to actual deployments via labels and namespaces. For example, suppose you want to set up a network policy that prevents the back-end egress between pods running in namespace **`custom-namespace`** for all the back-end pods. The following YAML configuration will do just that:

```
1.          apiVersion: networking.k8s.io/v1
2.          kind: NetworkPolicy
3.          metadata:
4.            name: deny-backend-egress
5.            namespace: custom-namespace
6.            spec:
7.              podSelector:
8.              matchLabels:
9.                tier: backend
10.       policyTypes:
11.       - Egress
12.           egress:
13.           - to:
14.             - podSelector:
15.             matchLabels:
16.             tier: backend
```

In the preceding YAML configurations, refer to line number 5, where the namespace is mentioned, and line 16, where the pod label is mentioned. When you apply the preceding policy, these rules will automatically be applied to all currently running and future pods matching the labels and namespace.

Network policies are an excellent way to enforce policies, but they uncover a few critical areas. For example, they only focus on pods. You cannot use them to define policies for your nodes or any other resource. Another uncovered area is that network policy has no way to identify the abuse of a policy. They do nothing to detect or alert you to potential security problems. Lastly, these policies do not provide a mechanism to encrypt data.

Securing via third-party tool

Network policies, no doubt, are the native way to apply security policies on the Kubernetes cluster. However, security features like encryption of data, identifying policy abuse, and alerting is configured by using various third-party components (not

available out of the box by Kubernetes). There are multiple security tools available in the industry, and because the specific security tools and features available from various third-party networking solutions for Kubernetes vary, there is no one-size-fits-all solution to addressing network security requirements through external tools.

Service mesh, such as Istio, provides features that network naïve Kubernetes network security policy misses and provides a lot of telemetry data and logs to identify the potential security loophole in the applications. This service meshes are gaining a lot of popularity these days, as their meshes are cloud agnostic, and hence once defined and configured correctly, the same policy will work across multi-cloud and on-prem deployments.

Best practices for securing a network

Some of the best practices for securing a network are as follows:

- **CNI plugins in use support network policies**: Among multiple different CNI options available, it is recommended to use an alternative that supports the network policies (both egress and ingress) of the K8S cluster. If the selected CNI does not, then restricting the traffic in the cluster will not be possible.
- **Ingress and egress network policies are applied to all workloads in the cluster**: K8S network policies allow developers and admins to enforce which traffic is allowed/disallowed in the Kubernetes cluster. The best practice is to create a default network policy to ensure that all pods and namespaces are secured. This avoids accidentally exposing an app or version that does not have a defined policy. Generally, this default network policy is defined as denying all ingress and egress traffic. Hence, it becomes the duty of each workload to open all the ingress-egress traffic, which is needed for itself.
- **Encrypt all communications inside the cluster**: Kubernetes does not encrypt the traffic between pods and nodes by default. However, from a security and compliance perspective, it is advised to put encryption in place for all the in-cluster communication. This is generally done either by configuring third-party tools like Weave and Calico or service mesh like Istio and linkerd. These tools and meshes add security and reliability features to applications by transparently inserting encryption at the platform level, and hence each application does not need to configure its mechanism. Two features of these tools used for secure communication are as follows:
    - M**utual TLS — mLTS**: Add encryption and certificates-based identity to cluster workloads
    - **Authorization policy**: Enforce traffic rules on the services level

  This book will cover service mesh - Istio and will discuss the aspect of not trusting any component in the cluster as safe (Zero trust).

- **The Kubernetes API, kubelet API, and etcd are not publicly exposed on the internet:** The control plane is the core of Kubernetes and allows users to view containers, schedule new Pods, read Secrets, and execute commands in the cluster. Because of these sensitive capabilities, the control plane should be highly protected. To accomplish this, following are some key activities recommended for your Kubernetes setup:
    - Set up robust authentication methods
    - Disable access to the internet and unnecessary or untrusted networks
    - Use RBAC policies to restrict access
    - Secure the **`etcd`** datastore with authentication and RBAC policies
    - Protect **`kubeconfig`** files from unauthorized modifications

The preceding discussion is not the complete set of security best practices for a Kubernetes-based setup. It is just a start, and you will be looking, in detail, at the security aspects one should keep in mind while using Kubernetes as a hosting platform for applications.

Conclusion

The way Kubernetes supports Ingress feature is quite unique. On the one hand, K8S specifies the way to define Ingress rules, and on the other, it allows you to configure Ingress implementation of your choice. The whole Ingress setup not only allows external entities to access your services in secure way but also abstracts the ephemeral scaling up and scaling down of pods. Kubernetes setup involves both out-of-the-box and third-party components, and hence, there are different strategies to manage these categories of components. As Kubernetes is maturing every day, so is the way traffic is handled in and out of cluster.

Points to remember

- Pods are ephemeral in nature; they come and go. Hence, using Pods address (the IP) is not right in communication with other entities.
- To tackle the mentioned nature of Pods, Kubernetes provides an abstraction called Service. External services can interact with a service, and pods' ephemeral nature is hidden behind a Service.
- However, exposing service also exposes IP address of the Service, so it is not the most recommended way to handle external entity communications. K8S has Ingress objects for tackling this.
- Ingress allows you to handle routing rules for the incoming HTTP requests.
- You can define network polices to govern the security aspects of the Kubernetes cluster.

- An API gateway helps you decorate your workflow deployment with several additional features, which enables you to implement cross-cutting features across all your APIs.

Multiple choice questions

1. **What is the technical difference between internal and external Kubernetes Services?**
   a. There is no difference; it is just terminology defining the purpose. If an external service is wrapped with ingress definition, it becomes back-end service.
   b. Back-end services cannot be accessed from external entities because `nodePort` is not specified.
   c. Services, by default, are external services.
   d. None of the above
2. **How can you apply governance on the security aspects of Kubernetes Network?**
   a. Via network policies
   b. Via third-party tools
   c. Both
   d. None

Answers

1. **b**
2. **c**

Questions

1. How does a pod get an IP address?
2. How can you ensure that the external entity is a valid candidate to connect to Kubernetes application/deployment?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 4

Kubernetes Workload Resources

Introduction

For any system that you build, user experience is very important, and the ease of use of the system is the key to it. Kubernetes simplifies the creation of several underlying objects by wrapping them with more user-friendly objects. It is like asking a contractor to build a modularized kitchen for you and the contractor building it with all the required equipment and features that you ask. Same is the case here; you do not need to ask Kubernetes to create pods for you. Instead, you ask for deployments, and Kubernetes does that by creating replica sets and Pods required underneath. Pods are what contain containers and your application image in turn, but you do not need to create pods directly. You can use Kubernetes Workload Resources like replica sets, deployments, jobs, DaemonSets, and StatefulSets as per your needs.

You would often want more than one instance of your application running, as this helps handle a greater number of incoming requests. Moreover, in the case of crashes, you have a backup instance running, which can take the load. You also do not want to scale up and down the number of pods manually, and Kubernetes provides another object to do that as well.

Most Kubernetes objects you see in this chapter have a single purpose. This is very much aligned with the single responsibility principle of the SOLID design principles used in software development.

One thing to note is that you might want to just 'deploy' your application or create a cron job to run it repeatedly, or even create multiple copies of your application with all of them having pods at their core, which are managed by these special objects.

Structure

In this chapter, we will discuss the following topics:

- ReplicaSets
    - Designing ReplicaSets
    - Creating ReplicaSets
    - Inspecting ReplicaSets
    - Scaling ReplicaSets
    - Deleting ReplicaSets
- Deployments
    - Creating Deployments
    - Managing Deployments
    - Updating Deployments
    - Deployment Strategies
    - Monitoring Deployment Status
    - Deleting Deployments
- Daemon Sets
    - Creating DaemonSets
    - Restricting DaemonSet to specific nodes
    - Updating DaemonSets
    - Deleting DaemonSets
- Kubernetes Jobs
    - Kubernetes Job Objects
    - Job Patterns
    - Pod and container failures
    - Cleaning up finished jobs automatically
    - CronJobs

Objectives

This chapter will provide details to the readers about the different workload resources that Kubernetes provides, on top of pod object. We will discuss ReplicaSets and Deployments first, which are mostly used for regular applications, and then we will explore DaemonSets and Jobs, which are used for special use cases. Knowing these workload resources will help readers identify what object they should choose for what purpose.

ReplicaSets

ReplicaSet maintains a set of replicas or pods. That is what its only aim is. This object simplifies the life cycle management of the pods. The containers are managed by Pods, and ReplicaSet manages the pods in it. *Figure 4.1* features a ReplicaSet containing two pods with the same container image:

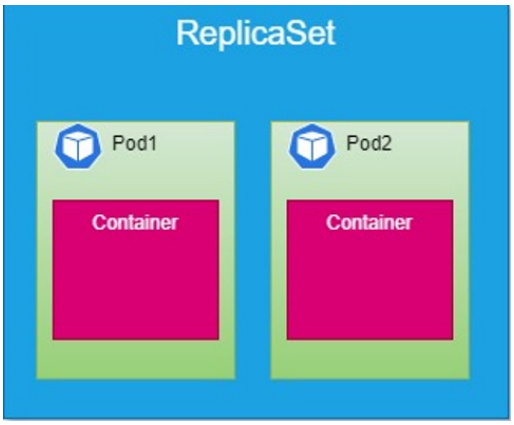


Figure 4.1: *ReplicaSet*

Deployment is a more user-friendly concept that manages ReplicaSets and provides declarative updates to Pods, along with several other features. Therefore, using deployments instead of ReplicaSets is preferred, unless you want finer control over orchestration. If you use deployments properly, you will hardly ever need to play with and manage the ReplicaSets.

Having said that, let us cover ReplicaSets first to know some basics before we look at what deployments are.

Designing ReplicaSets

ReplicaSets should contain the pods for stateless applications, which means they do not store the data or the state required for the functioning of the application. Why is

that so? It's because when a ReplicaSet is scaled down, any one of the pods can be killed, and your application should be resilient enough to cope up with the sudden loss. This is one of the differences between ReplicaSets and StatefulSets. StatefulSet is another Kubernetes object that we will discuss later in the book, but for now, you can think of ReplicaSets as **stateless** sets. That is what ReplicaSets are designed for.

Creating ReplicaSets

Creating a ReplicaSet with specifications in a yaml file is very easy. You specify how many pods or replicas you want to keep running, which contain the image of your application or any public image that you specify.

Let us start with a simple yaml and then we will look at a few important specifications:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-rs
  labels:
    app: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
     spec:
       containers:
       - name: nginx
         image: nginx
```

Here, the yaml is specifying the desired state as two *replicas of pods should be running, where every pod contains nginx image*. You save this yaml content in a file called **nginx-**

rs.yaml and run the **kubectl apply** or **kubectl create** command to create the ReplicaSet:

```
Kubectl apply -f nginx-rs.yaml
```

Let us take a note of the key attributes in this yaml:

- **spec.selector.matchLabels** specifies the label that the ReplicaSet looks for to acquire the pods.
- **spec.template** specifies the template for creating new pods if required.

The **matchLabels** is an important specification for ReplicaSets because as pods come and go, ReplicaSet filters the pods using these labels to identify which pods are controlled by it and which ones are not.

Inspecting ReplicaSets

The **kubectl describe** command can be used to get more information about specific pods, and you will get yaml response similar to that shown in the preceding example, with some additional information:

```
Kubectl describe rs nginx-rs
```

Note that the short form **rs** can be used instead of ReplicaSet in the **kubectl** command.

The snippet in *Figure 4.2* shows the details about the ReplicaSet, such as how many pods are desired and how many are currently running:

```
D:\>kubectl describe rs nginx-rs
Name:         nginx-rs
Namespace:    default
Selector:     app=myapp
Labels:       app=myapp
Annotations:  <none>
Replicas:     2 current / 2 desired
Pods Status:  2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=myapp
  Containers:
   nginx:
    Image:        nginx
```

Figure 4.2: *Describing a ReplicaSet*

Scaling ReplicaSets

You can use the **kubectl scale** command to scale the number of pods under a ReplicaSet. For the **replicaset nginx-rs** in discussion here, which has 2 pods as per the initial specification, we can use the following command if we want to increase the number of desired pods to 3:

```
Kubectl scale rs nginx-rs --replicas=3
```

If the scaling works, you shall see the output shown in *Figure 4.3*:

```
D:\>kubectl scale replicaset nginx-rs --replicas=3
replicaset.apps/nginx-rs scaled

D:\>kubectl describe rs nginx-rs
Name:         nginx-rs
Namespace:    default
Selector:     app=myapp
Labels:       app=myapp
Annotations:  <none>
Replicas:     3 current / 3 desired
Pods Status:  3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  app=myapp
```

Figure 4.3: *Scaling a ReplicaSet*

The ReplicaSet uses the pod template specified in its yaml specification to create new Pods. The same command can be used to scale down or reduce the number of pods within the ReplicaSet:

```
kubectl scale rs nginx-rs --replicas=2
```

The **`kubectl describe`** command shows the events related to the creation and deletion of pods. *Figure 4.4* features the events in scaling a ReplicaSet:

```
Events:
  Type    Reason            Age   From                   Message
  ----    ------            ----  ----                   -------
  Normal  SuccessfulCreate  32m   replicaset-controller  Created pod: nginx-rs-28bgz
  Normal  SuccessfulCreate  32m   replicaset-controller  Created pod: nginx-rs-wpm9w
  Normal  SuccessfulCreate  18m   replicaset-controller  Created pod: nginx-rs-nxdbt
  Normal  SuccessfulDelete  5s    replicaset-controller  Deleted pod: nginx-rs-nxdbt
  Normal  SuccessfulDelete  5s    replicaset-controller  Deleted pod: nginx-rs-28bgz
```

Figure 4.4*: Events in scaling a ReplicaSet*

The other and more declarative way to scale a ReplicaSet up or down could be to modify the **`nginx-rs yaml`** file with a new number of desired replicas and then use the **`kubectl apply`** command again.

If you are thinking that manually scaling up and down the application is not practical, you are totally right. When your application is serving real-time traffic, you do not want to monitor it continuously and scale the replicas by doing guess work. Instead, if the application is falling short on memory or CPU, and hence, is unable to serve requests, you want to **`AUTOSCALE`** the replicas to be able to serve the incoming requests. Kubernetes provides another object called **`HorizontalPodAutoscaler`**. Just keep this big name in mind for now, and we will discuss it in detail in *Chapter 10, Effective Scaling*, where we will discuss scaling the application.

Deleting ReplicaSets

The ReplicaSet can be deleted using the **`kubectl`** command:

```
kubectl delete replicaset nginx-rs
```

This command ensures that the pods created by the ReplicaSet are also deleted, as shown in *Figure 4.5:*

```
D:\>kubectl get pods
NAME             READY   STATUS    RESTARTS   AGE
nginx-rs-wpm9w   1/1     Running   0          42m

D:\>kubectl delete replicaset nginx-rs
replicaset.apps "nginx-rs" deleted

D:\>kubectl get pods
No resources found in default namespace.
```

Figure 4.5: *Deleting a ReplicaSet*

In case you just want to delete the ReplicaSet object and not the pods managed by the ReplicaSet, you can use cascade flag with value as orphan:

```
kubectl delete replicaset nginx-rs --cascade=orphan
```

Deployments

The deployment object in Kubernetes helps you manage releasing new versions of your application. The deployment object internally creates and manages the ReplicaSet. You do not need to manage the ReplicaSet managed by the deployment. Since deployment acts as a wrapper over ReplicaSet, there are several similarities between ReplicaSet and Deployment. *Figure 4.6* features a deployment containing a ReplicaSet:

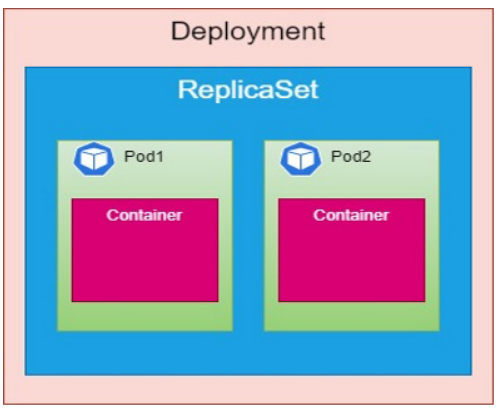


Figure 4.6: *Deployment containing a ReplicaSet*

Creating deployments

Let us begin with a declarative way to create a yaml with specifications for deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: myapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: nginx
        image: nginx:1.22.1
        ports:
        - containerPort: 80
```

The deployment spec, just like the ReplicaSet spec, has a pod template that is used to create new pods in the deployment if required.

You use the **`kubectl apply`** or **`create`** command to create a deployment with this yaml. The deployment will create two pods with nginx image as specified and a ReplicaSet.

You can verify it with the **`kubectl`** commands. *Figure 4.7* illustrates the creation of a deployment:

```
D:\>kubectl apply -f nginx-deployment.yml
deployment.apps/nginx-deployment created

D:\>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-665f8c45f5-46tpx   1/1     Running   0          14s
nginx-deployment-665f8c45f5-wb42n   1/1     Running   0          14s

D:\>kubectl get replicasets
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-665f8c45f5   2         2         2       20m
```

Figure 4.7: *Creating a deployment*

The ReplicaSet here is totally managed by the deployment object, so if you update the ReplicaSet directly and it is not in agreement with deployment object specifications, then the deployment will modify the ReplicaSet back to the **`Desired state`**.

Managing deployments

Use the **`kubectl describe`** command to get the details about a deployment:

```
kubectl describe deployment nginx-deployment
```

Refer to *Figure 4.8*:

```
OldReplicaSets:  <none>
NewReplicaSet:   nginx-deployment-665f8c45f5 (2/2 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  2d12h  deployment-controller  Scaled up replica set nginx-deployment-665f8c45f5 to 2
```

Figure 4.8: *Describing a deployment*

The **`OldReplicaSets`** and **`NewReplicaSet`** in the description tell which ReplicaSet is currently being managed by the deployment. In the preceding image, you can see that **`OldReplicaSet`** has no value. This means that only one ReplicaSet is active. If a deployment is in the middle of a rollout, you will see values for both the fields. We will discuss rollouts in further detail in a while.

Updating deployments

In the preceding yaml, the image specified for nginx was nginx:1.22.1 and number of replicas specified is 2. Let us say you want to update the image to nginx:1.23.3, which is a later version, and you want to upscale the number of replicas for the deployment from 2 to 3.

You should update the same yaml with these changes and use the **`kubectl apply`** command to update the deployment. This is how your updated yaml would look:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
      annotations:
        kubernetes.io/change-cause: "Use new nginx image"
    spec:
      containers:
      - name: nginx
        image: nginx:1.23.3
        ports:
        - containerPort: 80
```

This is a declarative way of updating deployments. Take note of the change-cause field under annotations. This is what is used while showing the rollout history. It is analogous to adding comments in pull requests or merge requests, while merging code in master branch using version control tools like git.

To check the status of the deployment as it happens, you can use the **`rollout status`** command for the deployment:

```
kubectl rollout status deployment nginx-deployment
```

The output of the command in *Figure 4.9* shows that replicas with new image are being created one after the another, and the old replicas are then terminated. This ensures that there is no downtime for your application:

```
D:\>kubectl rollout status deployment nginx-deployment
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 2 out of 3 new replicas have been updated...
Waiting for deployment "nginx-deployment" rollout to finish: 1 old replicas are pending termination...
```

Figure 4.9: *Deployment rollout in progress*

The output of the **`rollout history`** command, as shown in *Figure 4.10*, gives details the revision using the change-cause field mentioned previously, and once the rollout is complete, you can see the successful status.

```
D:\>kubectl rollout history deployment nginx-deployment
deployment.apps/nginx-deployment
REVISION  CHANGE-CAUSE
1         <none>
2         <none>
3         Use new nginx image


D:\>kubectl rollout status deployment nginx-deployment
deployment "nginx-deployment" successfully rolled out
```

Figure 4.10: *Successful rollout*

Describing the deployment is done via the **`kubectl describe`** command:

```
kubectl describe deployment nginx-deployment
```

The output shows the events of new replicas being scaled up and old replicas being scaled down, as shown in *Figure 4.11*:

```
OldReplicaSets:  nginx-deployment-559fd676f (3/3 replicas created)
NewReplicaSet:   nginx-deployment-586779b78 (1/1 replicas created)
Events:
  Type    Reason             Age   From                   Message
  ----    ------             ----  ----                   -------
  Normal  ScalingReplicaSet  71m   deployment-controller  Scaled up replica set nginx-deployment-559fd676f to 1
  Normal  ScalingReplicaSet  71m   deployment-controller  Scaled down replica set nginx-deployment-665f8c45f5 to 1 from 2
  Normal  ScalingReplicaSet  71m   deployment-controller  Scaled up replica set nginx-deployment-559fd676f to 2 from 1
  Normal  ScalingReplicaSet  71m   deployment-controller  Scaled down replica set nginx-deployment-665f8c45f5 to 0 from 1
  Normal  ScalingReplicaSet  4s    deployment-controller  Scaled up replica set nginx-deployment-559fd676f to 3 from 2
  Normal  ScalingReplicaSet  4s    deployment-controller  Scaled up replica set nginx-deployment-586779b78 to 1
```

Figure 4.11: *Events in a deployment*

You can see the values for **`OldReplicaSets`** and **`NewReplicaSet`** here, as the rollout is in progress. If you do not like the behavior of the new version of your deployment and want to pause it, you can use the **`kubectl rollout pause`** command:

```
kubectl rollout pause deployment nginx-deployment
```

To resume a paused rollout, use the **`kubectl rollout resume`** command:

```
kubectl rollout resume deployment nginx-deployment
```

If you want to undo a rollout, use the **`kubectl rollout undo`** command:

```
kubectl rollout undo deployment nginx-deployment
```

The better alternative to rollback a deployment is to update your yaml with the previous specifications and perform **`kubectl apply`** again. This ensures that the content of yaml indicates the details of the current deployment in the Kubernetes cluster.

Deployment strategies

One important specification in Kubernetes deployments is called **strategy**. There are two strategies possible. The first is called **`RollingUpdate`**, and it is the default one. This is what we saw in the previous examples, where new replicas are created before killing the old version. The second one is called **`Recreate`**, where old replicas are killed first and then the new ones are brought up. Obviously, this can cause some downtime for your application and shall be used with caution.

Readiness checks, which we discussed in the chapter on pods, help determine whether or not the pod brought up recently is ready to serve the traffic. If the readiness probes are not set for your pods, the deployment controller would not know if it should create new pods.

Monitoring deployment status

The **`kubectl describe`** command for a deployment tells the status of the deployment. You can look for the conditions array in the output of the **`kubectl describe`** command to check whether the deployment is progressing successfully or if it has failed. Refer to *Figure 4.12*:

```
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
```

***Figure 4.12**: Status of a Deployment*

Here the progressing condition has status as True because new ReplicaSet is made available. You can use this status and the reason for debugging purpose in case of errors with your deployment.

Deleting deployments

Deleting deployment with the **kubectl** command is easy.

```
kubectl delete deployment nginx-deployment
```

You can use **flag –cascade=orphan** with the **kubectl delete** command in case you want to delete just the deployment object and not the pods and ReplicaSets underneath it.

With that much information and huge respect in our hearts for deployments for simplifying our life, let us move on and discuss DaemonSets.

DaemonSets

ReplicaSets and Deployments are used for regular applications where multiple pods serve the incoming traffic. For ReplicaSets or deployments, all the pods could be on the same machine or node. But if you have a requirement where you want a specific functionality running on every machine in your cluster, then DaemonSet is what you want.

DaemonSets are not regular applications, but they are meant to add capabilities and features to the Kubernetes cluster itself. Examples could be a log collector service or a monitoring agent that needs to run on every machine. **kube-proxy** is a DaemonSet you will find in every Kubernetes cluster under namespace **kube-system**.

In an autoscaled Kubernetes cluster, where nodes keep coming and going without user intervention, DaemonSets are very helpful. The DaemonSet automatically adds the proper agents to each node, as it is added to the cluster by the autoscaler.

Creating DaemonSets

Playing captain obvious here, you can create a DaemonSet in a declarative way by using a set of specifications in a yaml file and the **kubectl apply** command. The DaemonSet is a declarative object like others and is managed by a controller.

The yaml for redis DaemonSet is as follows:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: redis-ds
  labels:
    app: redisds
```

```
spec:
  selector:
    matchLabels:
      app: redisds
  template:
    metadata:
      labels:
        app: redisds
    spec:
      containers:
      - name: redis
        image: redis:7.0
```

Most of the fields in this specification, like pod template, selector and so on, are what you have seen before. The difference between a DaemonSet and a ReplicaSet is that the DaemonSet does not ask for number of replicas; instead, it creates pods on all the **`eligible`** nodes in the cluster. Refer to *Figure 4.13*:

```
D:\>kubectl apply -f redis-ds.yaml
daemonset.apps/redis-ds created

D:\>kubectl get pods
NAME            READY   STATUS    RESTARTS   AGE
redis-ds-8nng7  1/1     Running   0          7s

D:\>kubectl get ds
NAME       DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
redis-ds   1         1         1       1            1           <none>          23s
```

Figure 4.13: *Creating a DaemonSet*

Note that you can use **`ds`** as a shortcut for DaemonSet in all **`kubectl`** commands.

Once a DaemonSet is in place, it will add new pods to every new node that gets added to your cluster; similarly, for the nodes that get deleted, the DaemonSet pod will be removed from them.

If a node is modified and a required label is removed from it, then the DaemonSet pod will also be removed from that node by the DaemonSet controller. *Figure 4.14* features multiple nodes in a cluster, where each node contains a pod as per DaemonSet specification:

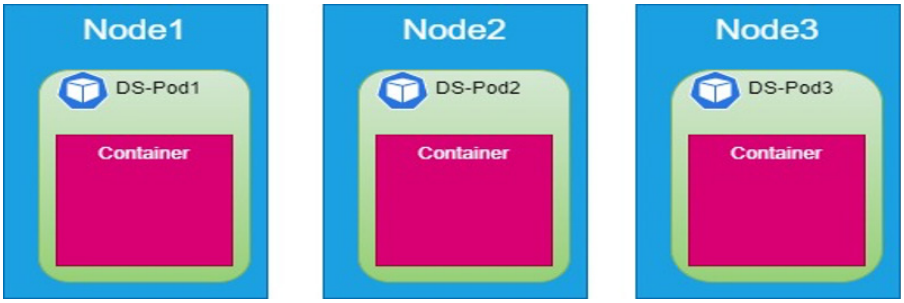


Figure 4.14: *DaemonSet Pods*

The DaemonSet pods are generally not deployed on control plane nodes in the cluster because control plane nodes usually have taints or repelling techniques defined for non-system service functions and workloads.

Restricting DaemonSets to specific nodes

Thanks to the declarative architecture of Kubernetes, there is a way to create a DaemonSet that creates pods only on specific nodes, and you can use node selector for that. Node selectors are key-value pairs used in pod specifications to ensure that the pods are created only on the nodes having matching labels. Consider a use case where the application in DaemonSet demands GPU or SSD, and let us say that it is available only on specific nodes in the cluster. You can use the node selector to ensure that DaemonSet pods are not created on ineligible nodes.

The following is a spec from DaemonSet yaml, where pod template specifies node selector:

```
spec:
  selector:
    matchLabels:
      app: redisds-gpu
      gpu: “true”
    template:
      metadata:
        labels:
          app: redisds-gpu
          gpu: “true”
        spec:
          nodeSelector:
```

```
      gpu: "true"
  containers:
  - name: redis
    image: redis:7.0
```

This ensures that pods from DaemonSet are created only on nodes with the **gpu: "true"** label, as shown in *Figure 4.15:*

```
D:\>kubectl apply -f redis-ds.yaml
daemonset.apps/redisds-gpu created

D:\>kubectl get ds
NAME          DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR   AGE
redis-ds      1         1         1       1            1           <none>          7h21m
redisds-gpu   0         0         0       0            0           gpu=true        18m
```

Figure 4.15: *DaemonSet with no pods*

As shown in *Figure 4.15*, if there are no nodes matching the selector, no pods will be created even though the DaemonSet object is created. Once a node that matches the specified node selector is available, a pod will be scheduled on it.

Updating DaemonSets

Updating DaemonSets to newer version of the image is very similar to updating deployments or ReplicaSets. The declarative and preferred way is to update the yaml specifications and use the **kubectl apply** command.

To enable the rolling update feature of a DaemonSet, you must set **spec.updateStrategy.type** to **RollingUpdate**. This is the **default** value for **updateStrategy**.

The other possible value for **updateStrategy.type** is **OnDelete**, where new pods in DaemonSet are created only after old pods are deleted.

Let us discuss the **RollingUpdate** strategy in further detail. Just like deployments, the **RollingUpdate** strategy gradually updates members of a DaemonSet until all the Pods in DaemonSet are running the new configuration. There are three parameters that control the rolling update of a DaemonSet:

- **spec.minReadySeconds**, which determines how long a Pod must be **ready** before the rolling update proceeds to upgrade subsequent pods
- **spec.updateStrategy.rollingUpdate.maxUnavailable**, which indicates how many pods may be unavailable during the update; the default value is 0
- **spec.updateStrategy.rollingUpdate.maxSurge**, which indicates the maximum number of pods that may have the new configuration during the update; the default value is 0

Both **maxSurge** and **maxUnavailable** cannot be 0 at the same time. The **kubectl rollout** commands can also be used to update the DaemonSets, just like we saw for deployments:

```
kubectl rollout status daemonset redis-ds
```

Refer to *Figure 4.16*:

```
D:\>kubectl rollout status ds redis-ds
Waiting for daemon set "redis-ds" rollout to finish: 0 of 1 updated pods are available...
Waiting for daemon set "redis-ds" rollout to finish: 0 of 1 updated pods are available...
daemon set "redis-ds" successfully rolled out
```

Figure 4.16: *DaemonSet rollout status*

While a rollout is in progress, you can pause and resume it with the **kubectl rollout pause** and **kubectl rollout resume** commands. Just like deployments, you can undo a rollout with the **rollout undo** command, and the history for rollouts can be checked with the **rollout history** command. Refer to *Figure 4.17*:

```
D:\>kubectl rollout undo ds redis-ds
daemonset.apps/redis-ds rolled back

D:\>kubectl rollout history ds redis-ds
daemonset.apps/redis-ds
REVISION  CHANGE-CAUSE
2         <none>
3         <none>
```

Figure 4.17: *Undoing a DaemonSet rollout*

Deleting DaemonSets

The way of deleting DaemonSet is no different than the way of deleting other workload objects in a Kubernetes cluster. Yes, you have got the hang of it: it is using the **kubectl delete** command. Refer to *Figure 4.18*:

```
D:\>kubectl delete ds redis-ds
daemonset.apps "redis-ds" deleted

D:\>kubectl get pods
No resources found in default namespace.
```

Figure 4.18: *Deleting a DaemonSet*

Deleting the DaemonSet deletes the pods managed by it. The **flag –cascade=orphan**, like earlier, can be used to delete just the DaemonSet object and not the pods underneath.

Kubernetes Jobs

A Job in Kubernetes is another way to manage pods to perform certain workloads. Just like deployments and DaemonSets, you specify the pod template, and the Job object will create one or more pods and will continue to retry execution of the pods until a specific number of pods successfully terminate. The Job tracks the successful completion of pods. When a specified number of successful completions is reached, the task, and the Job in turn, is complete.

One of the ways is to create one Job object to reliably run one pod to completion. The Job object will start a new pod if the first pod fails or is deleted (for example, due to a node hardware failure or a node reboot).

You can also use a Job to run multiple pods in parallel. To run a Job at specific intervals or at a fixed time of the day, you can create CronJobs.

Jobs

Let us look at a sample yaml to create a Job in Kubernetes that executes a Python script inside a container:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pythonjob
spec:
  template:
    spec:
      containers:
      - name: pythonjob
        image: pythonjob
        imagePullPolicy: IfNotPresent
        args:
        - ./main.py
        command:
        - python3
      restartPolicy: Never
  backoffLimit: 2
```

As we have seen in the yamls earlier, **`spec.template`** is the template for pods to be created for the job's execution; **`restartPolicy`** and **`backoffLimit`** are interesting terms, and we will discuss them in a while.

The **`main.py`** file should be present in the container image, and it is run using the **`python3`** command. The packages, if any, required to execute the Python script, should be installed beforehand in the container using dockerfile. When you create a job using yaml specification in the preceding file, you can check that a pod is created to execute the task. Refer to *Figure 4.19*:

```
D:\>kubectl apply -f pythonjob.yaml
job.batch/pythonjob created

D:\>kubectl get pods
NAME              READY   STATUS      RESTARTS   AGE
pythonjob-p7774   0/1     Completed   0          7s
```

Figure 4.19: *Creating a Job*

Note the pod name in this command, which we can use to get logs for the pod and see the outcome of the job execution. Refer to *Figure 4.20*:

```
D:\>kubectl logs pythonjob-p7774
CPU usage is 6.3
memory usage is svmem(total=26725761024, available=24464711680, percent=8.5, used=1866567680, free=21506768896,
```

Figure 4.20: *Logs of a Job*

The Python script executed as part of the job provides details of CPU usage and memory usage of the system, which is what you see in the logs for the Job Pod:

The **`kubectl describe`** command gives you the details of the job, like statuses of the pods created, image used and the events that took place. Refer to *Figure 4.21*, which shows the partial output of the command:

```
Duration:         10s
Pods Statuses:    0 Running / 1 Succeeded / 0 Failed
Pod Template:
  Labels:  controller-uid=09aef617-e0ee-48a8-bad5-0911247167c2
           job-name=pythonjob
  Containers:
   pythonjob:
    Image:      pythonjob:latest
    Port:       <none>
    Host Port:  <none>
    Command:
      python3
    Args:
      ./main.py
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Events:
  Type    Reason            Age   From            Message
  ----    ------            ----  ----            -------
  Normal  SuccessfulCreate  56s   job-controller  Created pod: pythonjob-p7774
  Normal  Completed         46s   job-controller  Job completed
```

Figure 4.21: *Describing a Job*

Job patterns

A Job object can be used in multiple ways or patterns to accomplish a task. A task can be divided into multiple work items. The most widely used pattern is to use a single Job to complete a task or all work items underneath it. The other variant is to use a single Job object per work item, which can create some complexities due to a multiple Jobs and hence, should be used with caution. The number of pods that can be running at a time is defined by the **`spec.parallelism`** field in the Job specification.

The other pattern is to create a Pod for every work item, or a single Pod can process multiple work items. This should be decided based on the complexity of the work items and/or the task itself.

Many approaches use a work item queue that requires a queue service. The details of the work items are put in the queue, and multiple jobs or pods, depending upon the approach chosen, will pick single or multiple messages from the queue and process.

Pod and container failures

A container in a Pod may fail for several reasons, and if this happens and the **`restartPolicy`** is set to **`OnFailure`**, then the Pod stays on but the container is re-run. If **`restartPolicy`** is set to **`Never`**, then the Pod is considered failed and a new pod is created. The application in the container should handle the scenarios where the container restarts, either in the same pod or a different one.

Moreover, in case of container failures, it is better to specify how many retries should be made by the controller before marking the Job as failed. The **`backoffLimit`** field is part of the pod specification and indicates the number of retries allowed. The default value for **`backOffLimit`** is **`6`**. The pods where execution fails for a Job are recreated by the Job controller with an exponential back-off delay like 10s, 20s, 40s, and the delay is capped at 6 minutes.

Cleaning up finished jobs automatically

If you delete a Job, it will clean up the pods it created. Suspending a Job is also possible, and then the Job will delete its active pods until it is resumed. Refer to *Figure 4.22*:

```
D:\>kubectl delete job pythonjob
job.batch "pythonjob" deleted

D:\>kubectl get pods
No resources found in default namespace.
```

Figure 4.22: *Deleting a Job*

Finished or completed Jobs are not automatically deleted, and keeping them around will put pressure on the API server, which is not ideal. This helps you check the pods and logs for the completed jobs and do some debugging if required.

But in most cases, you want to and should ensure automatic deletion of the pods after the Job is complete. Make use of **`spec.ttlSecondsAfterFinished`** in the job specifications to delete the pods after the defined number of seconds. The **`ttlSecondsAfterFinished`** feature is available in stable state since Kubernetes version 1.23.

CronJobs

A CronJob creates a Job to run repeatedly after a specified interval, or at a specified time every day, every week and so on. CronJob is like a wrapper object over Job, and it creates Job objects as per the schedule to run the defined task. Following is a sample yaml:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: pythoncronjob
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: pythonjob
            image: pythonjob
            imagePullPolicy: IfNotPresent
            args:
            - ./main.py
            command:
            - python3
          restartPolicy: OnFailure
```

The schedule contains 5 *s, which means this CronJob will create a Job every minute and run the application in the container. The 5 *s indicate the values for different time units, as follows:

- The first * indicates minute and possible values other than * are 0-59.
- The second * indicates hour and possible values other than * are 0-23.
- The third * indicates day of the month and possible values other than * are 1-31.
- The fourth * indicates month and possible values other than * are 1-12.
- The fifth * indicates day of the week and possible values other than * are 0-6.

Table 4.1 features some sample entries for schedule and their meaning:

0 * * * *	Every hour at the first minute of hour
0 0 * * *	Every day once at the first minute of the day
1 3 5 11 *	At 03:01 on 5th day of month November
0 0 10 * 5	Every Friday at midnight and 10th of every month at midnight

Table 4.1: *Entries for schedule and their meaning*

Figure 4.23 shows that CronJob creates a Job and pod associated with the Job at the specified interval (beginning of the next minute in this case). After a CronJob is deleted, the controller deletes the child objects of Jobs and pods:

```
D:\>kubectl apply -f python-cronjob.yaml
cronjob.batch/pythoncronjob created

D:\>kubectl get jobs
No resources found in default namespace.

D:\>kubectl get jobs
NAME                      COMPLETIONS   DURATION   AGE
pythoncronjob-27867754    0/1           4s         4s

D:\>kubectl get pods
NAME                            READY   STATUS      RESTARTS   AGE
pythoncronjob-27867754-4bgt8    0/1     Completed   0          9s

D:\>kubectl delete cronjob pythoncronjob
cronjob.batch "pythoncronjob" deleted

D:\>kubectl get jobs
No resources found in default namespace.

D:\>kubectl get pods
No resources found in default namespace.
```

Figure 4.23: *Creating and deleting a CronJob*

Conclusion

We witnessed the power of Kubernetes architecture in this chapter. We saw how Kubernetes provides the ability to create different objects for different purposes, where the low-level object underneath is the same, that is, a pod.

You can create ReplicaSets to run multiple copies of your application inside a pod, and that way, you do not have to manage individual pods. Deployment is another critical object in the world of Kubernetes, which manages ReplicaSet on its own. Deployment helps release newer versions of your application and a lot of **`kubectl`** commands are available for ease of use.

DaemonSets are not the regular applications, but they are meant to add additional capabilities and features to the nodes in Kubernetes cluster. DaemonSets provide an abstraction for running a set of Pods on every node in a Kubernetes cluster.

Jobs are the Kubernetes objects that can be used to run workloads, which is completed by one pod all the way. The other way could be to run the workload that can be completed by multiple pods. In which case, one big task can be divided into smaller chunks and pods come and go as the work allotted to them is complete.

StatefulSet is another important workload resource that we will look at in an upcoming chapter on Storage with Kubernetes.

Points to remember

- Choose deployments over ReplicaSets to manage multiple copies of your application. Deployments create and manage the ReplicaSets on their own.
- Use the **`change-cause`** annotation in deployment yaml to specify the reason for creating a new version of the deployment.
- Readiness checks set on pods help deployment controller with the **`RollingUpdate`** of your deployment. Readiness check tells the controller that the pod is ready to serve traffic and hence, the new pod can be brought up.
- The pod template specified in DaemonSet should have **`restartPolicy`** set to **`always`**. If unspecified, the default value is also **`always`**.
- You can use node selectors in pod template inside DaemonSet to ensure that pods are created only on specific nodes in the cluster.
- To delete the pods of the job automatically after finishing, use the **`ttlSecondsAfterFinished`** specification for the Job.
- You can use the **`CronJob`** object to run a Kubernetes job at certain intervals, and the times of the **`CronJob`** are based on the time zone of the **`kube-controller-manager`**.

Questions

1. You have a web application deployed on Google Cloud Platform's Cloud Run. The application is made up of multiple stateless microservices, and you want to port the application to Kubernetes. Which workload resource will you use?
2. You want to trigger a task to download files from a remote server whenever the user clicks a button in your web application. It is okay to perform this download in a back-end service. Which workload resource will you choose to accomplish this?
3. You want the DaemonSet pod to run on all nodes in the cluster. However, when you check the number of pods created for the DaemonSet, it is always one less than the number of nodes in the cluster. What could be the reason for this, and how will you fix it?

Answers

1. You can deploy containerized images of the application microservices in a Kubernetes cluster using deployments.
2. Since the download is going to happen on demand, create a job object with a specific curl command to download the file whenever user clicks a button.
3. This happens mostly due to control plane pod repelling the non-system feature pods. Ensure that you really want the DaemonSet pod to run on control plane node as well, and then add the required tolerations to the DaemonSet pod specifications.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 5

ConfigMap, Secrets, and Labels

Introduction

Kubernetes, since its inception, has matured into an orchestration framework that not only handles the complexities of the infrastructure it is running on but also has multiple features that make application development effective. A general expectation of modern-day workload is multi environment deployment of the same domain code. When you extend the multi deployment environment thought to container-based deployments, it means you build a container image once and deploy it in multiple environments. This deployment of a container image needs some variables as per the environment, and to tackle this, Kubernetes provides the constructors of ConfigMaps and Secrets.

Another aspect of efficiently managing application is to identify components that belong to a flow/user journey or application, for example, how many containers of application ABC are running in the system currently. Identifying the number of containers is just a query; what you essentially need is a way to group in the infrastructural footprint of the application. To handle this grouping, Kubernetes provides the constructs of Labels.

In this chapter, we will take a detailed look at how to effectively use ConfigMaps, Secrets, and Labels for applications deployed on Kubernetes cluster.

Structure

In this chapter, we will discuss the following topics:

- ConfigMaps
    - Creating ConfigMaps
    - Consuming ConfigMaps
- Secrets
    - Creating Secrets
    - Consuming Secrets
- Managing ConfigMaps and Secrets
    - Listing
    - Creating
    - Updating
- Upgrading Secrets and ConfigMaps
- Applying and modifying labels
- Labels selectors
- Defining annotations

Objectives

After studying this chapter, you should have a good understanding of how to pass runtime variables, both secure and human readable, to a container image with the help of ConfigMaps and Secrets. You will also learn how to assign labels to your deployments and query them effectively to real-world scenarios. Lastly, you will see the concept of annotations and how you can benefit from the construct of annotations.

ConfigMap

A software application generally has two parts: code and its configurations. Code is something that does not change when you intend to deploy the application in multiple environments. It is the associated configurations that change as per environment. Code is generally built once, and the binary deployable code is used across environments. For example, suppose you are writing a Java application that interacts with a MySQL database. This application, when deployed across multiple environments, will have different **`connection`** parameters for the MySQL database. Hence, you generally extract the connection string of MySQL into a separate configuration. Configuration management can be a topic, as different engineering

team manage configs differently. For instance, some of them manage it by just keeping configurations in file, and others manage configurations via putting them into databases, and so on.

Imagine that you want to containerize this Java application and want to pass config information to the cluster while you are instantiating a container. These configurations generally become environment variables inside the container and could be easily used by applications. ConfigMaps is one of the mechanisms Kubernetes uses to inject configuration data into applications pod. **`ConfigMap`** is combined with the pods just before its instantiation, and this means that your Pod definition files can be easily reused across environments by just changing the **`ConfigMap`**.

Consider *Figure 5.1* for a better understanding of the preceding discussion:

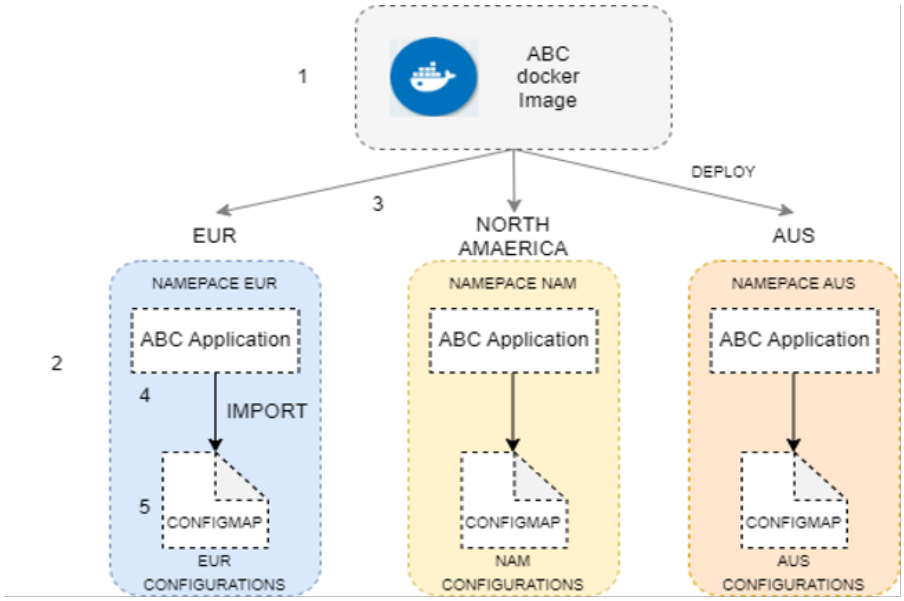


Figure 5.1: *Introduction to ConfigMap*

Refer to the numerical labelling in *Figure 5.1* with the corresponding numerical explanations as follows:

1. ABC is the name of the application for which a docker image is created and pushed to a container image registry.
2. Represent a POD in EUR region. There are three environments showcased in the image: EUR, North America, and AUS. Segregation and definition of environment vary from organization to organization. For example, three different regions might mean different namespaces for an organization, while for another organization, it might mean separate Kubernetes cluster.

3. The same docker image (generated in step 1) is pulled and instantiated in different environments (EUR, North America, and AUS). These deployments are done in namespaces. As discussed earlier (Chapters 1 and 3), namespace enables the isolation of a group of resources in a Kubernetes cluster.

```
1. kubectl create namespace <namespace-name>
2. kubectl create deployment snowflake --image=registry-name/
   image-name:<tag>  -n=<namespace-name>--replicas=2
3. kubectl get deployment -n=<namespace-name>
```

   Line 1 creates a namespace, which is used in line 2 by deployment name snowflake. You can use **-n** or **-namespace** in all **kubectl** commands to bind the effect to a namespace.

4. It represents a running pod using the container image.
5. The POD running (described in step 4) imports the **ConfigMap** configured for each environment. The same docker image (built once) imports different **ConfigMap** configured for each environment to create pods.

Let us now investigate the aspects of creating and using **ConfigMap**.

Creating ConfigMap

ConfigMap, just like other Kubernetes objects, can be created both in imperative and declarative manner. Assume that you have a key value file, i.e., **config.txt**, with the following content:

```
db.host=10.51.248.17

db.port=3306
```

Moreover, assume that you want to add a few more properties (**environment = frontend**) to your **ConfigMap** during runtime. You can create **ConfigMap** using the following command:

```
kubectl create configmap first-config-map --from-file=config.txt --from-
literal=environment=frontend --namespace=sample-namespace
```

In the preceding command, **first-config-map** is the name of the **ConfigMap**. You can trigger the following commands to see the details about the **ConfigMap**:

```
kubectl describe configmaps first-config-map -n sample-namespace
```

The output of the preceding command looks like this:

```
1. Name:         first-config-map
```

```
2. Namespace:    sample-namespace
3. Labels:       <none>
4. Annotations:  <none>
5. Data
6. ====
7. config.txt:
8. ----
9. db.host=10.51.248.17
10. db.port=3306
11.
12. environment:
13. ----
14. frontend
15. BinaryData
16. ====
17. Events:  <none>
```

As you can see between line numbers 5 and 14, the data you configured as part of the **ConfigMap** creation command is present. Moreover, if you look closely at the metadata fields (line 3 and 4), they contain the default values.

You can get the yaml version of the created **ConfigMap** with the following command:

```
kubectl get configmap first-config-map -n sample-namespace -o yaml
```

The outcome will be a yaml representation of the **ConfigMap**, as follows:

```
1. apiVersion: v1
2. data:
3.   config.txt: |
4.     db.host=10.51.248.17
5.     db.port=3306
6.   environment: frontend
7. kind: ConfigMap
8. metadata:
```

```
    creationTimestamp: "2023-01-30T23:24:09Z"
    name: first-config-map
    namespace: sample-namespace
    resourceVersion: "9083226"
    uid: b909a64d-8de5-4642-9d07-633a2daa863d
```

As you can see, **ConfigMap** is just a collection of key value pairs. You can create the preceding yaml first and use the **kubectl apply** command to create **ConfigMap** as well. Refer to the codebase section associated with the chapter. The following command helps you create **ConfigMap** in a declarative way:

```
kubectl create -f first-config-map-via-yaml.yml
```

There are multiple ways to use the preceding commands; for example, rather than giving just one file with **--from-file=<file-name>**, you can use it multiple times to create one **ConfigMap** from multiple files.

Consuming ConfigMaps

There are three ways in which you can use a **ConfigMap** inside a pod:

- Consume **ConfigMap** in environment variables
- Set command-line arguments with **ConfigMap**
- Consuming **ConfigMap** via volume plugin

Let us look into when and how to use each of the preceding strategies.

For the sake of demonstration of the strategies, assume that you have two **ConfigMap** (**database-config** and **tier-config**) already created.

The details of database-config are as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: database-config
  namespace: default
data:
  host: 10.51.248.17
  port: 3306
```

Details of tier-config are:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: tier-config
  namespace: default
data:
  tier: frontend
```

You can get these created by triggering commands on the files in the attached codebase, as follows:

```
kubectl create -f db-configmap.yml
kubectl create -f tier-configmap.yml
```

Let us now use **ConfigMap** as per the three mentioned strategies.

Consume ConfigMap in the environment variables

ConfigMap can be used for populating the environment variables both one-by-one and in their entirety. For example, refer to the below pod definition, which uses the two **ConfigMaps -** `database-config` and `tier-config`, created above. Refer to the **configmap-as-env-variable.yml** file in the attached codebase, and trigger the following command to create a pod:

```
kubectl create -f configmap-as-env-variable.yml
```

Following is a slice of code from the **configmap-as-env-variable.yml** file, which is our current topic of discussion:

```
containers:
  - name: test-container
    image: gcr.io/google_containers/busybox
    command: [ "/bin/sh", "-c", "env" ]
    env:
      - name: DB_HOST
        valueFrom:
          configMapKeyRef:
            name: database-config
```

```
10.             key: host
11.       - name: DB_PORT
12.         valueFrom:
13.           configMapKeyRef:
14.             name: database-config
15.             key: port
16.       envFrom:
17.         - configMapRef:
18.             name: tier-config
```

The preceding pod definition creates a container using the **busybox** image available on internet (line 3), and it prints all the environment variables of the container (line 4).

Consider lines 5 to 15, which showcase a way to use individual variables inside a **ConfigMap**. Additionally, lines 16 to 18 showcase how to use the entire **ConfigMap** in one go.

If you look at the logs of the pod, you will see the following output:

```
2023-01-31 06:21:21.926 IST DB_PORT=3306
2023-01-31 06:21:21.926 IST HOSTNAME=demo-configmap-as-env-variable
2023-01-31 06:21:21.926 IST tier=frontend
2023-01-31 06:21:21.926 IST DB_HOST=10.51.248.17
```

Set command-line arguments with ConfigMap

ConfigMap can be used to configure the value of the arguments in a container. This can be done using the substitution syntax **$(VAR_NAME)**. Refer to the **configmap-as-cmd-variable.yml** file in the attached codebase and trigger the following command to create a pod:

```
kubectl create -f configmap-as-cmd-variable.yml
```

The following is a slice of code from the **configmap-as-cmd-variable.yml** file, which is our current topic of discussion:

```
1. containers:
2.   - name: test-container
3.     image: gcr.io/google_containers/busybox
4.     command: [ "/bin/sh", "-c", "echo $(db_host) $(db_port)" ]
```

```
5.     env:
6.       - name: DB_HOST
7.         valueFrom:
8.           configMapKeyRef:
9.             name: database-config
10.            key: host
11.      - name: DB_PORT
12.        valueFrom:
13.          configMapKeyRef:
14.            name: database-config
15.            key: port
```

Consider line 4, which simply prints the **db_host** and **db_port** properties from configured database-config ConfigMap (line 5 to 15).

Consuming ConfigMap via volume plugin

ConfigMap configurations can be made available inside containers via the volume plugin as well. Kubernetes volume are the smallest deployable units, which connect the persistent storage with ephemeral pods. Data is saved at a persistent storage, and pods get the reference to the data when created. The life cyle of data is not tied to that of a pod, that is, even if the pod is deleted, the data will remain as is.

There are a couple of ways to achieve this. The most elementary path is to configure volumes with files, where key is the file name and the content of the file is the value of key. Refer to the **configmap-via-volume.yml** file in the attached codebase and trigger the following command to create a pod:

```
1. containers:
2.     - name: test-container
3.       image: gcr.io/google_containers/busybox
4.         command: [ "/bin/sh", "-c", "cat /etc/config/database-config.
   host" ]
5.       volumeMounts:
6.       - name: config-volume
7.         mountPath: /etc/config
8.    volumes:
```

```
9.      - name: config-volume
10. configMap:
11.   name: database-config
```

The contents of volume definition (line 8 to 11) are mounted inside the pod at location **/etc/config**, as mentioned in line 7. The output of the preceding exercise will be the host value (that is, 10.51.248.17) mentioned in **ConfigMap database-config**.

We can also control the paths within the volume where **ConfigMap** keys are projected. Refer to the **configmap-via-volume-path.yml** file in the attached codebase and trigger the following command to create a pod:

```
1.  containers:
2.      - name: test-container
3.        image: gcr.io/google_containers/busybox
4.        command: [ "/bin/sh", "-c", "cat /etc/config/path/to/host" ]
5.        volumeMounts:
6.        - name: config-volume
7.          mountPath: /etc/config
8.     volumes:
9.      - name: config-volume
10.       configMap:
11.         name: database-config
12.         items:
13.         - key: host
14.           path: path/to/host
```

Pay attention to line 14, where the database-config ConfigMap's host key value is mounted to a specific location (that is, **path/to/host**); the same has been used in line number 4.

Among all the defined strategies, one might think as to why there are multiple strategies to accomplish the same thing. The answer to that question is that it depends on your use case. For example, suppose you have an application where you want to not let the config change take effect right away. You want to apply the change when the application restarts. In this case, strategy 1 and 2, that is, **Consume ConfigMap** in environment variables and set command line arguments with **ConfigMap**, will work. These strategies make the variable available as the environment variable at the creation time of Pod, and any change to the config map values will not take effect

unless we restart the pod. On the other hand, if you use the **ConfigMap** via volume plugin, any changes made to **ConfigMap** will be available right away inside the pod and hence, no Pod restart is needed.

Secrets

In the previous section, you saw how **ConfigMap** could be used to pass configuration values to pods during their creation. Certain types of data are extra sensitive. This extra sensitive attribute is managed via the Kubernetes object **Secret**. Secrets are passed to pods at the time of creation, the same way we do with **ConfigMap**. It is just that the value of the passed variables is encrypted.

Creating Secrets

Secrets can be created in two ways:

- Via the test files
- Via yaml configuration

The following command elaborates the commands to create secrets via files:

```
1. echo -n 'admin' > ./username.txt
2. echo -n '1f2d1e2e67df' > ./password.txt
3. kubectl create secret generic db-credentials \
4.    --from-file=./username.txt \
5.    --from-file=./password.txt
```

Lines 1 and 2 create two files: one with username and other with password; lines 3 to 5 use these files to create secrets.

The second way to create secrets is via the following YAML file:

```
1. apiVersion: v1
2. kind: Secret
3. metadata:
4.    name: app-secrets
5. type: Opaque
6. data:
7.    username: YWRtaW4=
8.    password: MWYyZDFlMmU2N2Rm
```

In the preceding YAML file, lines 7 and 8 are base64 encoded values for username and password. Store the preceding yaml into a **secrets.yaml** file and execute the following command:

```
kubectl apply -f secrets.yaml
```

You can see the details of the created secrets with the following command:

```
kubectl describe secrets app-secrets
```

The output of the preceding command is as follows:

```
1. Name:         app-secrets
2. Namespace:    default
3. Labels:       <none>
4. Annotations:  <none>
5.
6. Type:  Opaque
7.
8. Data
9. ====
10. password:  12 bytes
11. username:  5 bytes
```

In this output, line 1 represents the name, and line 2 tells the namespace. If you want this secret to be created in a certain namespace, you can attach **-n <namespace-name>** to the described command. Lines 3 and 4, that is, **Labels** and **Annotations**, were not passed while creating secrets. The concepts of label and annotations will be discussed later on in this chapter. Line 6 represents the type of secret that you had created, and lines 8 to 11 is the data section of the secrets.

Coming back to line 6, the type is available in the following built-in types. The main aim of defining the type is to put a validation on the type of information in the secret. These validations are not related to checking the values of secrets but are merely structural checks. For example, the type you used in the preceding code is **Opaque** (default). This file could also be used to define basic authentication type secrets, similar to the preceding example, which defines all the fields needed to define the secret. Refer to the codebase attached with the chapter and take a look at **basic-authentication-secrets.yaml**. You can use the **kubectl apply** command to create the secret.

Refer to this link for a complete list of the types of secrets:

https://kubernetes.io/docs/concepts/configuration/secret/#secret-types

Consuming Secrets

Just as there are two strategies of using **ConfigMap**, there are two ways to consume secrets:

- Mounted as volume
- Exposed as environment variables

Secret volume sources are expected to be configured before they are being used, because when you use such sources in creating Pod objects, pod object creation step verifies existence of volume sources. If you specify a secret in pod creation yaml and the secret is not accessible or not created, the Kubectl will periodically try to fetch the secret. In case it fails to fetch a secret, it reports an event to the pod, which includes the details of the problem fetching the secret.

Consuming Secrets mounted as volume

In this section, you will be revisiting the example you saw in case of **ConfigMap**. Refer to the **secret-via-volume.yml** file attached with the codebase of the chapter. Snippet of file **secret-via-volume.yml**, relevant to current discussion is as follows:

```
1. containers:
2.   - name: test-container
3.     image: gcr.io/google_containers/busybox
4.     command: [ "/bin/sh", "-c", "cat /etc/config/username" ]
5.     volumeMounts:
6.     - name: config-volume
7.       mountPath: "/etc/config"
8.   volumes:
9.     - name: config-volume
10.      secret:
11.        secretName: app-secrets
```

Line numbers 5 to 7 create a volume mount inside the pod. Line numbers 8 to 11 assign secrets inside the mounted location. As you can see in line 4, secrets are present as files in the mounted **location/etc/config**.

When the secrets are mounted as volumes, Kubernetes keeps a track of the modifications and makes the changed value eventually consistent inside the pod. Kubectl keeps a cache of secrets keys and values that are used as volumes inside

the pod. The **configMapAndSecretChangeDetectionStrategy** field in the Kubectl configuration controls the strategy Kubectl hold to make the modified value available. By default, the strategy used is Watch. The total time taken for the updated value to be present includes the time Kubectl talks to detect change, plus the time taken by Kubectl to update the cache.

Consuming Secrets as environment variables

This strategy of consuming Secrets is very similar to the strategy of using **ConfigMap** as environment variables. Refer to the yaml file named **secrets-as-env-variable.yml**, a snippet of which is as follows:

```
containers:
- name: test-container
  image: gcr.io/google_containers/busybox
  command: [ "/bin/sh", "-c", "env" ]
  env:
  - name: Username
    valueFrom:
      secretKeyRef:
        name: app-secrets
        key: username
  - name: Password
    valueFrom:
      secretKeyRef:
        name: app-secrets
        key: password
```

The constructs are very similar to the **ConfigMap** way of doings things, with some minor changes in the attribute keys. As in the case of **ConfigMap**, a pod restart is needed to reflect modified values in Secrets.

Private docker registries

Kubernetes allows to pull images from private container registries. However, for pulling those images, credentials need to be passed while pulling the image. There is also a possibility that you are freeing to multiple such private repositories. This results in managing credentials for each private registry on every possible node in the cluster.

Image pull secrets is a construct that leverages secret API to automate the distribution of private registry credentials. Image pull secrets are kept as normal secrets and are consumable in the following manner.

Use the **`create secret docker-registry`** to create this special kind of secret:

```
$ kubectl create secret docker-registry app-image-pull-secret \
--docker-username=<username> \
--docker-password=<password> \
--docker-email=<email-address>
```

Consider the **`secrets-as-env-variable-image-pull.yaml`** file attached with the codebase. It is the same one that you had seen in the *Consuming Secrets as environment variables* section, with just one additional attribute, as follows:

```
imagePullSecrets:
  - name: app-image-pull-secret
```

If you intend to pull the image from the same repository (rather than multiple, as discussed previously), you can get the secret added to the default service account associated with each pod. This removes the need to mention these secrets in every pod you create.

Managing ConfigMaps and Secrets

ConfigMap and Secrets are managed via Kubernetes APIs. In this section, you will learn about the commands to list, create and update Secrets and ConfigMap.

Listing

You can use **`kubectl get`** commands to list both Secrets and ConfigMap. Consider the two commands and their outcome when executed on cluster.

The command to list down all the secrets belonging to the specified namespace is as follows:

```
kubectl get secrets -n <namespace>
```

The output of this command is as follows:

```
NAME                  TYPE                                  DATA   AGE
app-secrets           kubernetes.io/basic-auth              2      43h
db-config             Opaque                                3      112d
```

The command to list down all the **ConfigMap** in the namespace is as follows:

```
kubectl get configmaps -n <namespace>
NAME                                    DATA    AGE
csv-parsing-configuration               1       24d
golden-record-management                 1       28d
```

You can describe the single **Secret** or **ConfigMap** object:

```
kubectl describe secret app-secrets -n <namespace>
kubectl describe configmap csv-parsing-configuration -n <namespace>
```

Creating

Generally, Secrets and ConfigMap are created using the **kubectl create secret** generic or **kubectl create configmap**. The ways to pass the content to the two commands vary. Following is the list of ways that could be used:

- **--from-file=<filename>**: Load from the file with the secret data key the same as the filename
- **--from-file=<key>=<filename>**: Load from the file with the secret data key explicitly specified
- **--from-file=<directory>**: Load all the files in the specified directory where the filename is acceptable
    - key name
- **--from-literal=<key>=<value>**: Use the specified key / value pair directly

You can also combine any of the preceding two options in a single command.

Updating

There are primarily two ways to update Secrets and ConfigMaps for your application. As discussed earlier, if you want the application to automatically reflect the modified values, then 'using Secrets and ConfigMap via volume plugin' is the preferred approach. You will not need to restart the application. Let us investigate both approaches in more detail as follows:

- **Updating from file:** If you have a manifest for your **ConfigMap** or Secret, and **ConfigMap** and Secret maintained as Yaml files, you can modify the files and modify the Secret and **ConfigMap** with **kubectl replace -f <filename>**. You can also use **kubectl apply -f <filename>** if you previously created the resource with **kubectl apply**. This strategy can work in case of ConfigMap,

but it is not the recommended way to manage secrets. Secrets are sensitive information, and you managing the yaml files separately could be a concern.

- **Editing Secret\ConfigMap:** This is another way to tweak ConfigMap, which uses the **`kubectl edit`** to pull a version of your ConfigMap in the editor, which you can update. While updating secrets, ensure that you are putting the base64 encoded value. This edits command does not make the conversion automatically.

  ```
  kubectl edit configmap my-config -n <namespace>
  ```

ConfigMaps and Secrets are great ways to provide runtime configuration to the container image built once and launched across multiple environments. Generally, in a project, we build code once and create an image. We do various static and dynamic, security scan to identify vulnerabilities in the image based on security standard defined across Enterprises and push the image to container registry. This entire process of reliably building the image is a process and takes time. It is recommended to build the image as part of the build process of deployment pipeline and use the same image while deploying to all the environments.

Applying and modifying labels

Modern-day applications keep growing, both infrastructurally and programmatically. If you talk specifically about infrastructure, an application generally starts small and eventually matures into a more prominent application due to multiple reasons. We need to scale up due to the data load increasing every day. More components are getting added. For example, you can attach a cache to lower the response time of your APIs. Kubernetes provides two foundational constructs of Labels and annotations to organize, mark and cross-index all your resources to represent a group.

Labels are key-value pairs that can be attached to any Kubernetes object, such as Pods and replica sets. Users can pull information about all the Kubernetes objects by querying the labels. On the other hand, annotations are key/value pairs that hold non-queryable information that tools and libraries could leverage.

Labels have very simple syntax. They are key/value pairs, where strings represent both the key and value. Let us try to use one of the pod definitions you have already seen and attach labels to it.

Consider the **`configmap-as-cmd-var-labels.yaml`** file, which is a copy of the **`configmap-as-cmd-var.yaml file`**, with the following additional lines:

```
1. metadata:
2.   name: demo-configmap-as-cmd-variable
3.   labels:
```

```
4.     envionment: handson-exercises
5.     app: sample-app
```

Consider lines 3 to 5. When you run the following command, it will create pods that are labelled with two key values: **environment=handson-exercise** and **app=sample-app**:

```
kubectl create -f configmap-as-cmd-var-labels.yml
```

You can list the labels on all the pods using the following command:

```
kubectl get pods -n <namespace> --show-labels
```

You can get all the labels on a particular pod using the following command:

```
kubectl get pod demo-configmap-as-cmd-variable --show-labels
```

The **demo-configmap-as-cmd-variable** pod is created as part of the following exercise. The show label command on this pod (**demo-configmap-as-cmd-variable**) should return the two labels we defined in the pod definition yaml:

```
NAME                READY  STATUS      RESTARTS        AGE       LABELS
demo-configmap-as…   0/1            Completed     0
16m          app=sample-app,environment=handon-exercises
```

You can also assign single\multiple labels to an already running pods, as follows:

```
kubectl label pod demo-configmap-as-cmd-variable
{key1=value1,key2=value2,key3=value3}
```

Not just pods, you can also modify any deployment with the following command:

```
kubectl label deployments <deployment-name> "canary=true"
```

You can delete a label as well by applying a dash suffix. Take a look at the following command:

```
kubectl label deployments <deployment-name> "canary-"
```

You can list all the labels associated with pods with the following command:

```
kubectl get pods --show-labels
```

The output of the preceding command looks as follows:

```
NAME                              ...  LABELS
demo-configmap-as-cmd-variable    ...  env=prod,ver=1,...
demo-configmap-as-env-variable    ...  env=prod,ver=1,...
demo-configmap-via-volume-basic   ...  env=test,ver=2,...
demo-secrets-as-env-variable      ...  env=prod,ver=2,...
```

Labels selectors

You can query the running objects in Kubernetes by defining the conditions of labels. This makes them extremely useful in organizing things running on a cluster. You can query a label via label selectors, which is just a string evaluated with a condition. For example, you can attach a label as **`ver=1`** and apply a label selector as follows:

```
kubectl get deployments -l 'ver=1'
```

There are two kinds of label selectors: equality-based and set-based selectors.

Equality-based selector

An equality-based selector is just a **IS/IS NOT** test. Consider the following example:

```
 env = prod
```

The preceding check will return all pods that have label with key **`env`** and value as **`prod`**.

You can trigger the following commands to see results:

```
kubectl get deployments --selector="env = prod"
```

The output of the preceding command will be as follows:

```
NAME                              ...  LABELS
demo-configmap-as-cmd-variable    ...  env=prod,ver=1,...
demo-configmap-as-env-variable    ...  env=prod,ver=1,...
demo-secrets-as-env-variable      ...  env=prod,ver=2,...
```

You can also use a negation operator in the parameters of above command, as shown below:

```
kubectl get deployments --selector="env! = prod"
```

The output of the preceding command will be as follows:

```
NAME                              ...  LABELS
demo-configmap-via-volume-basic   ...  env=test,ver=2,...
```

You can also combine two selectors with a comma, as shown here:

```
kubectl get deployments --selector="env = prod, ver != 2"
```

The preceding command will return all the pods whose **`env = prod AND ver !=2`**. The output of this command will be as follows:

```
NAME                                ...  LABELS
demo-configmap-as-cmd-variable   ...  env=prod,ver=1,...
demo-configmap-as-env-variable   ...  env=prod,ver=1,...
```

Set-based selectors

A set-based selector is just a **IN/NOT IN** test. Consider this example:

```
env IN (prod,test)
```

The preceding check will return all pods that have labels with key **`env`** and value as **`prod`** or **`test`**. You can trigger the following commands to see results:

```
kubectl get deployments --selector=" env IN (prod,test)"
```

The output of the preceding command will be as follows:

```
NAME                                ...  LABELS
demo-configmap-as-cmd-variable   ...  env=prod,ver=1,...
demo-configmap-as-env-variable   ...  env=prod,ver=1,...
demo-configmap-via-volume-basic  ...  env=test,ver=2,...
demo-secrets-as-env-variable     ...  env=prod,ver=2,...
```

Set-based selectors also allow combining two set-based selector with a comma.

Role of labels in Kubernetes architecture

Besides enabling users to organize and manage the underlying infrastructure, labels play a crucial role in linking various Kubernetes objects. All Kubernetes objects are independent, and there is no hierarchy among multiple objects. But in many cases, there is a need to relate two objects together. These relationships are defined using the constructs of labels and label selectors.

For example, if a service load balancer finds the pods, it should redirect traffic using the label selector. When people want to restrict network traffic in their cluster, they use network policy in conjunction with specific labels to identify pods that should or should not be allowed to communicate with each other. **`ReplicaSet`**, which creates and manages replicas of a pod, identifies the pod they manage via label selector. When a pod is built on a specific node pool, a pod identifies the node pool using the node selector to identify a set of nodes where a pod can be instantiated.

A label is a construct that cohesively manages the relationship between two independent objects in Kubernetes. An application can start small with an elementary set of labels, but these labels are expected to grow complex as and when the maturity of the application increases.

Defining annotations

Annotations are constructs that let you attach metadata with a Kubernetes object to assist tools and libraries. They could be used as tools to pass additional information to the external system.

Labels are used to recognize a group of Kubernetes objects. Annotations, on the other hand, are additional/extra information about an object, such as how to use it, associated policies, and point of origination. One might debate that if we put complete information on labels, will it serve the purpose of passing adequate metadata info to user? The answer is yes, it will, but it is up to the individual developers to choose annotation or label. Generally, in case of information being doubtful as the label, we make it an annotation.

Some common sample situations where annotations make sense are as follows:

- Track why an object exists in Kubernetes; what is the reason for the updates to the object?
- Communicate the schedule when the object runs triggers an action.
- Attach build, release, or image information that is not ideal for labels (can include a Git hash, timestamp, PR number, and so on).
- Provide extra data to enhance the visual quality or usability of a UI.
- Annotations are used in multiple places throughout Kubernetes, for use cases being rolling deployments. During such rolling deployments, annotations play a crucial role in keeping track of rollout status and provide the information required to roll back deployment to a previous state in case of issues.

Annotations are defined in the common metadata section in every Kubernetes Object. Refer to the **`secrets-as-env-variable-image-pull-annotation.yml`** file. Following is the snippet of the file that defines annotations:

```
1.          metadata:
2.            name: demo-secrets-as-env-variable
3.            annotations:
4.              xyz.com/icon-urls: "https://xyz.com/icon.png"
```

Line numbers 3 and 4 showcase how an annotation is attached to the pod.

Conclusion

ConfigMaps and secrets are ways to dynamically provide configurations to your container image. They allow the definition of the pod to be created once, and the exact definition has been fed with different variables per the deployment environment. Labels enable efficient groups of independent Kubernetes objects. Labels and label selectors are the tools for two Kubernetes objects to define relationships. While labels make objects queryable, annotations are the metadata associated with an object, making managing that object easy. They can be used to pass information related to the nature of Kubernetes objects.

Points to remember

- ConfigMaps and secrets provide a mechanism to pass dynamic configuration to your application. Separating configuration from application code enables and facilitates the 'build once and deploy many' principle of software engineering.
- Labels are used to group objects in a Kubernetes cluster. Labels are also used in selector queries to define a relationship between two independent objects of Kubernetes.
- Annotations provide metadata stored in key-value pairs, making the usage of the object efficient, both internal systems and externals.
- Labels and annotations are crucial to understanding how critical components in a Kubernetes cluster work collaboratively to ensure the desired state of the object in a cluster. Labels and annotations enable flexibility and provide the initial point for building automation tools and deployment workflows.

Questions

1. What is the key difference between a ConfigMap and Secrets?
2. What are two (among many) situations where Kubernetes objects use label selectors to define relationships?
3. What are the ways in which a secret or a ConfigMap provides configuration to a pod\deployment?
    a. via environment variable
    b. via volume plugin
    c. None of the above
    d. Both 1 & 2

4. Which will be your preferred approach to pass ConfigMap if you do not want your application to be restarted and still want the changed ConfigMap values to get reflected in the pod\deployment?

    a. Via environment variable

    b. Via volume plugin

Answers

3. **d**
4. **b**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 6

Configuring Storage with Kubernetes

Introduction

Stateless systems are simple, and everybody loves simple. However, not all systems can stay simple forever. Sooner or later, complexity makes its way into the system, and you must start storing the state of the application somewhere. Now 'somewhere' is a vague term, and there are multiple possibilities behind it. The data storage can be as simple as cache in the application for a session, or it could be the back-end database from where the data is queried to be shown in the application.

If you are not storing the application state in a database, you might want to use the storage available with a container or pod. However, as we know, a container or pod can go down, and the filesystem in it could be unavailable suddenly. This means if you want to store the application state or share the data between pods, you must use something external, and this is what volumes provide. The volumes' content can be made available to the pods or containers right after they restart.

It is also important to not give up on flexibility and scalability when building, deploying, and managing a stateful application. The persistent volumes and storage classes help in keeping your application logic to consume the storage and the infrastructure logic to provision the storage independent. Scaling stateful applications is more challenging with Kubernetes as compared to Stateless applications, and this is where StatefulSets help.

Structure

In this chapter, we will discuss the following topics:

- Storage provisioning in Kubernetes
    - Volumes
    - Persistent Volumes and Persistent Volume claim
    - Storage class
    - Using StorageClass for dynamic provisioning
- StatefulSets
    - Properties of SatefulSets
    - Volume claim templates
    - Headless service
- Installing MongoDB on Kubernetes using StatefulSets
- Disaster recovery
- Container storage interface

Objectives

We discussed some types of volumes in *Chapter 2, PODs*, when we discussed storage in relation with pods. In this chapter, we are going to go a step further and discuss persistent volumes, storage classes and StatefulSets. Just like different workload resources have pods at their center, different storage options in Kubernetes have volume at their center. We will discuss the ways to configure the storage statically and dynamically.

Storage provisioning in Kubernetes

The easiest way to ask for storage in Kubernetes is by creating a volume. A volume has multiple types, and you can create different types of volumes for different purposes. You can provision the volume statically or dynamically. In both cases, you will have to create **`PersistentVolume`** and **`PersistentVolumeClaim`** objects. Storage class provides a dynamic way to provision the storage for your application.

Volumes

Here's a bit of a refresher about volumes before we jump to Persistent Volumes: a volume is an object that references a storage. You give unique names to the volumes

you create, and they are associated to pods. The specifications for volume decide how the life cycle of a volume is associated with the life cycle of the Pod. A **`volumeMount`** is used to reference a volume, and it does this by name of the volume, and then it defines how to mount it and define what is called as a **`mountPath`**.

There are several volume types out there supported by Kubernetes, and we discussed **`emptyDir`**, **`hostPath`** and **`NFS`** with some examples in *Chapter 2, PODs*.

Persistent Volumes and Persistent Volume claims

A persistent volume is a cluster-wide storage unit with a life cycle that is completely independent from the pod. It is generally set up by the administrator, and then it talks to cloud storage, **network file system** (**NFS**), local storage, or any other kind of storage.

While the **`emptyDir`** volumes are attached to a Pod and **`hostPath`** volumes are attached to a Node, the persistent volumes are made available to the pod even if it gets rescheduled on a different node.

The persistent volume relies on a storage provider like NFS or a cloud storage or some other options.

You create **`PersistentVolumeClaim`** to *claim* or use this storage. These things are related as you can see; an administrator creates the storage using persistent volume, and then developers use the claim in the pod specification or deployment specification to use the storage.

Let us break the workflow down into steps:

1. A persistent volume is a cluster-wide resource that relies on some type of network attached storage. This is normally going to be provisioned by a cluster administrator, but obviously, trying out things locally on a single node cluster is also possible. So, creating some type of network storage resource, NFS, cloud, or other kind of storage should be the first thing to be done.

2. Once storage is available, we define a persistent volume, and that is going to be registered with the Kubernetes API.

3. After creating a **`PersistentVolume`** object, we need to create a **`PersistentVolumeClaim`** (**PVC**) to be able to use this persistent volume. Then, Kubernetes binds that PVC to the **persistent volume** (**PV**), which means the storage is available for use.

4. And finally, we are going to use this **`PersistentVolumeClaim`** to say that we have a pod, and we need to get to this storage for the pod, and so we are going to claim it. So, we can modify the pod template or the deployment

template, or maybe another type of resource, and we will bind to that **`PersistentVolumeClaim`**.

Now it is available to a pod, even if that pod gets rescheduled to a different node, because this type of storage is not specific to any of the worker nodes. So, with this type of approach, no matter where your pod gets scheduled, the pod will find its storage and happiness too. Refer to *Figure 6.1* to see how PV and PVC help bridge the container and storage:

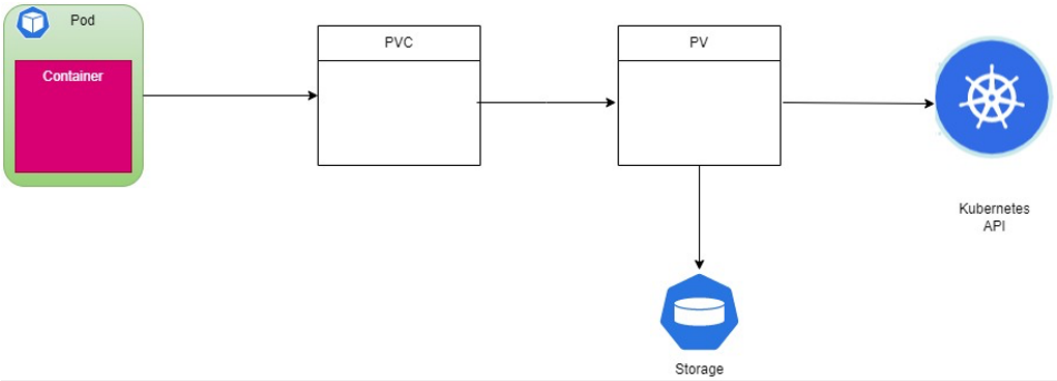


Figure 6.1: *PV and PVC*

You could say that the picture looks good now. With the PV and PVC in place, we would be able to talk to the PersistentVolume, which would allow your pod to read or write to that storage. So that is how it is done.

You may question: are these not too many steps just to use the storage in our application? You would be right, but the flexibility this approach provides is something that you will need once your application grows and matures. If you use the easiest way to define a volume directly inside the pod, then your application is also tied to the storage, because if the storage changes, you must change all the pod's volume definitions and apply them again. So, this does not make your project portable from one storage to the other. This is where the abstraction provided by **`PersistentVolume`** and **`PersistentVolumeClaim`** is useful. They are the separate objects that are deployed as part of the infrastructure, and the application developers will not have to worry about storage provisioning that way.

Let us look at the **`yaml`** specifications for PV and PVC, and we will discuss some important fields in them:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-volume
```

```
spec:
  accessModes:
  - ReadWriteOnce
  capacity:
    storage: 30Gi
  nfs:
    path: /tmpdata
    server: 172.72.1.2
  persistentVolumeReclaimPolicy: Retain
  volumeMode: Filesystem
```

This specification creates a PV pointing to an NFS server. The administrator would have to ensure that this server exists first, and then the PV object should be created with the specifications shown in the preceding code.

The terms **`metadata.name`** and **`capacity`** are quite self-explanatory, but **`accessModes`** demands a discussion. Let us look at some of the possible values for **`accessModes`** and their meanings:

- **`ReadWriteOnce`**: The volume can be mounted or used for read-write operations by a single node. Multiple pods running on the same node can access the volume.
- **`ReadWriteOncePod`**: The volume can be mounted or used for read-write operations by a single pod. This mode ensures that only one pod across the whole cluster can read that PV or write to it. This has limited support, and Kubernetes documentation would indicate the current volume types supported. This mode is still in beta mode with Kubernetes version 1.27 at the time of writing of this book.
- **`ReadOnlyMany`**: The volume can be mounted as read-only by many nodes.
- **`ReadWriteMany`**: The volume can be mounted as read-write by many nodes.

Similarly, **`volumeMode`** can have two possible values:

- **`Filesystem:`** This is the default mode used when the **`volumeMode`** parameter is omitted. With this mode, a **`filesystem`** is mounted into pods into a directory.
- **`Block:`** With this mode, a volume is presented into a pod as a block device, where filesystem is absent. This provides the fastest possible way to access a volume for pod, as there is no filesystem layer between the pod and the

volume. However, the application running in the pod must know how to handle a raw block device.

We talked particularly about NFS **PersistentVolume** here. However, Kubernetes supports several persistent volume drive types. There are persistent volume drivers for all major public and private cloud providers.

Let us now look at a **yaml** specification for the **PersistentVolumeClaim**:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongodb-persistent-storage-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  volumeMode: Filesystem
  volumeName: mongo-volume
```

The claim has the **resources.requests** that is used to specify the storage required, and the **accessMode** decides the nature of the usage. Do note that creating a claim does not guarantee the allocation of the storage. Rather, this object helps decouple the storage from the pod. As the PVC object does not have many details about the storage, it is portable. Thus, the user creates a claim stating the request, and a controller in Kubernetes will look for matching **PersistentVolumes**. If they match, then the PV is bound to a PVC.

The pod definitions will have claims as volumes. Following is an example pod yaml:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-mongo
spec:
  volumes:
  - name: mongo-volume
```

```
    persistentVolumeClaim:
      claimName: mongodb-persistent-storage-claim
  containers:
  - image: mongo:6.0.1
    name: mongod-container
    volumeMounts:
    - mountPath: /data/db
      name: mongo-volume
```

You can see here that **`PersistentVolumeClaim`** is mentioned under volumes specification. The **`volumeMounts.mountPath`** specifies to the pod where the volume would be mounted in the pod.

The claim must exist in the same namespace as that of the pod. The cluster then finds the **`PersistentVolume`** matching the claim, and then the volume is mounted into the pod at the specified **`mountPath`**.

When you are done with the volume, you can delete the PVC objects from the API that allows reclamation of the resource. The reclaim policy for a **`PersistentVolume`** tells the cluster what to do with the volume after the claim is released.

Currently, volumes can be retained, recycled, or deleted as per the specified reclaim policy:

- **Retain**: If the reclaim policy is specified as `retain`, that means even if the **`PersistentVolumeClaim`** is deleted, the corresponding **`PersistentVolume`** is not deleted. The administrator can manually claim the data by deleting the PV first and then the content in the storage to free up the space and use it for different purpose. Of course, if the data is required, a new PV can be created to use for a different purpose.
- **Recycle**: This option is deprecated and supported by limited volume types only, such as NFS and **`hostPath`**. The reclaim policy of recycle scrubs the data in the volume, and then the volume is made available for reclaim.
- **Delete**: If the reclaim policy is `delete`, then the data is deleted upon the deletion of PV object. This is generally the default policy for dynamically provisioned **`PersistentVolumes`**. This policy is also supported by limited volume types.

A volume can be in one of the following phases, which depend on the reclaim policy and the associated PVC. Refer to *Figure 6.2* to see roughly how a PV goes through different phases:

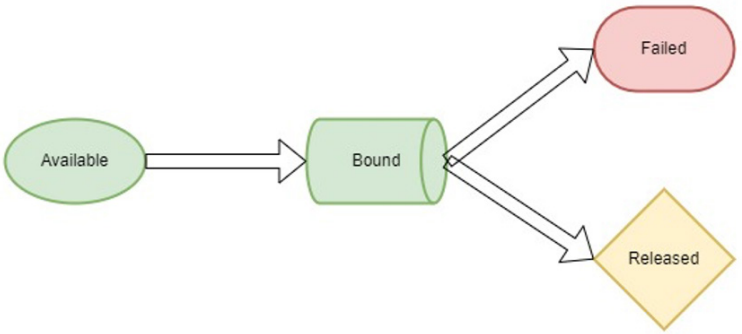


***Figure 6.2**: Phases of a PersistentVolume*

Phases of a PersistentVolume are as follows:

- **Available:** The volume is free and not yet bound to a claim.
- **Bound:** The volume is bound to a PVC.
- **Released:** The claim for the volume has been deleted, but the volume resource is not yet reclaimed by the cluster.
- **Failed:** The volume has failed its automatic reclamation.

Storage class

While you can manually create persistent volumes and use them with a persistent volume claim, there is another way called **`StorageClass`**, which is a dynamic way to provision the storage.

Now that we are going to discuss a little advanced concept, let us understand some downsides of using **`PersistentVolume`** to provision the storage:

- Using PV and PVC is a kind of multi-step process where storage admin must first create the storage, and then the Kubernetes admin has to create PV objects in Kubernetes.
- There is also a possibility of overallocation of disk space for **`PersistentVolumes`**, because a claim can be bound to the volume, which is prepared beforehand, and it may have a storage size bigger than needed, leading to a wastage of disk space.
- From the **`PersistentVolume`** specifications, it is clear that PersistentVolumes are not portable across different volume types. The PV that uses NFS cannot

be used as is, for using Google Cloud Persistent Disk because it needs change in the specification. Thus, you will have to create different PVs for different storage types.

The **`StorageClass`** object in Kubernetes API addresses these concerns and provides a way to dynamically provision the storage of required *class* and the exact required space. This should bring enough excitement to you to know more about the **`StorageClass`** object.

A **`StorageClass`** is nothing but an object with template, and it can be combined with a provisioner, and you can then dynamically set up the storage on the fly. On the other hand, what we discussed earlier would have been static where we have to go in and set up the storage, and create the PV and PVC, all by ourselves. Now, while you could certainly use a **`StorageClass`** locally, this is normally where an administrator would set up a **`StorageClass`** template, and then you could bind to it. Let us look at a sample yaml:

```
apiVersion: storage.k8s.io/v1

kind: StorageClass

metadata:

  name: gce-storage-class

parameters:

  type: pd-ssd

provisioner: kubernetes.io/gce-pd

reclaimPolicy: Delete

volumeBindingMode: Immediate
```

Each **`StorageClass`** contains the fields provisioner, **`reclaimPolicy`**, and parameters that are used when dynamically provisioning the **`PersistentVolume`**:

- The provisioner specified in the preceding example starts with Kubernetes.io, which indicates that it is an internal provisioner that is shipped with Kubernetes. There are also external provisioners that you can use for storage types like NFS or iSCSI and so on.
- The **`reclaimPolicy`** specified in the **`Storage`** class is what is inherited by the **`PersistentVolumes`** created using the storage class. We will discuss **`reclaimPolicy`** in detail when we talk about disaster recovery.
- The parameters specified for storage class describe the volumes belonging to the storage class. Different parameters may be accepted depending on the provisioner, and the maximum number of parameters that can be specified is

512. Another limitation of the **parameters** object is its total length, including its keys and values, cannot exceed 256 KiB.

volumeBindingMode is another interesting field that indicates when the volume binding and dynamic provisioning would occur.

- The default value is **Immediate**, which means that as soon as the **PersistentVolumeClaim** is created, the **PersistentVolume** will be created and bound. This is not exactly ideal, and in cases where pods have more scheduling requirements, they may not get the required **PersistentVolume** and would be left unscheduled.
- The other possible value for **volumeBindingMode** is **waitForFirstConsumer** where the volume is not created until a pod using the matching PVC is created. This will delay the binding and dynamic provisioning of **PersistentVolumes** until required.

Using StorageClass for dynamic provisioning

Storage class does not remove the need for PV and PVC objects. They are still needed. However, users can request dynamically provisioned storage by using the name of a storage class in their **PersistentVolumeClaim**.

Following is a sample yaml where a PVC is referring to **StorageClass** created in the preceding example:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ssd-persistent-storage-claim
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: gce-storage-class
```

So you create a storage class first and then the PVC, both using the **kubectl apply** command.

This claim results in an SSD-like persistent disk being automatically provisioned using **`gce-pd`** as provisioner, as per the storage class. The volume is destroyed when the claim is deleted.

A cluster administer can enable default behavior for dynamic provisioning by taking the following two steps:

1. Marking one **`StorageClass`** object default by adding the **`storageclass.kubernetes.io/is-default-class`** annotation to it
2. Enabling the **`DefaultStorageClass`** admission controller on the API server

This controller adds the **`storageClassName`** field pointing to the default storage class when the user has created a PVC object without specifying **`storageClassName`**. Of course, a default **`StorageClass`** should exist in the cluster.

Note that there should be at most one default storage class on a cluster. In a situation when multiple **`StorageClass`** objects are marked as default, a **`PersistentVolumeClaim`** without **`storageClassName`** explicitly specified cannot be created because the **`AdmissionController`** rejects the request.

StatefulSets

In *Chapter 4, Kubernetes Workload Resources*, we discussed all Kubernetes workload resources except StatefulSets. StatefulSets deserve special treatment because they are special; they are, in a way, ReplicaSets doing more than ReplicaSets. They manage the replicas for pods just like ReplicaSets, but they also guarantee the ordering and uniqueness of the Pods.

They are workload resources with Pods underneath them, but the pods have unique identifiers with them, unlike ReplicaSets or Deployments, which is what makes the pods under StatefulSets non-interchangeable. StatefulSets are widely used for applications that require ordered deployment and scaling of pods, and the applications that require stable storage.

Let us look at the properties of StatefulSets that make them popular.

Properties of StatefulSets

Some of the properties of StatefulSets are as follows:

- Each replica created by StatefulSet gets a persistent hostname with a unique index, for example, mongo-0, mongo-1, and so on.
- Each replica is created in the order from the lowest to the highest index, and the creation of new replica a pod is blocked until the replica pod at the

previous index is healthy and available. This rule is also followed during the scaling up of the StatefulSet.

- During the deletion of the StatefulSet, managed replica pods are deleted in order from the highest to the lowest. This rule is also followed during the scaling up of the StatefulSet.

These properties make the StatefulSets very useful for applications, which store the *state*. The fixed host names and guaranteed ordering means that the replicas can rely on the presence of certain replicas, which would be created before them.

It would be clearer why StatefulSets are useful for dealing with storage once we discuss the downsides of using ReplicaSets or deployments for such use cases. It is not technically impossible to use Deployments to create pods that handle the database requests. However, that approach poses certain limitations as your application grows.

Most databases are scaled up by creating replicas of the storage, and a primary instance is used to handle write operations, while secondary instances are used to handle read requests. Refer to *Figure 6.3* to understand this scenario:

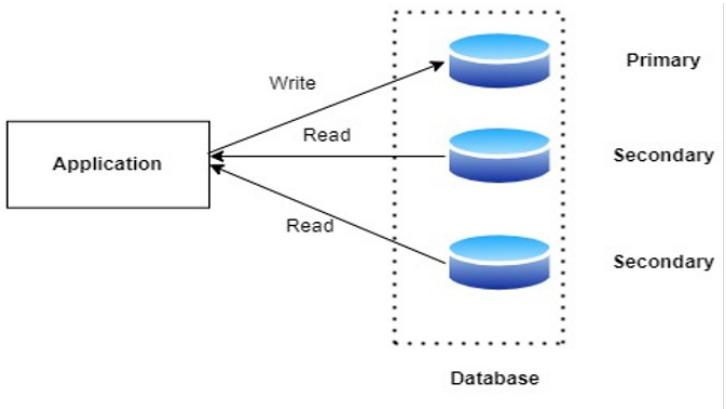


Figure 6.3: *Scaling a Database*

Moreover, if you use the deployment on top of the database, you will have a single PVC and PV talking to the database through multiple replicas. More and more replicas or pods will communicate to the same storage, and you will start seeing issues with reading and writing data to the database. In such cases, scaling up just the database application with a higher number of pods will not help. You will have a scenario, as shown in *Figure 6.4*:

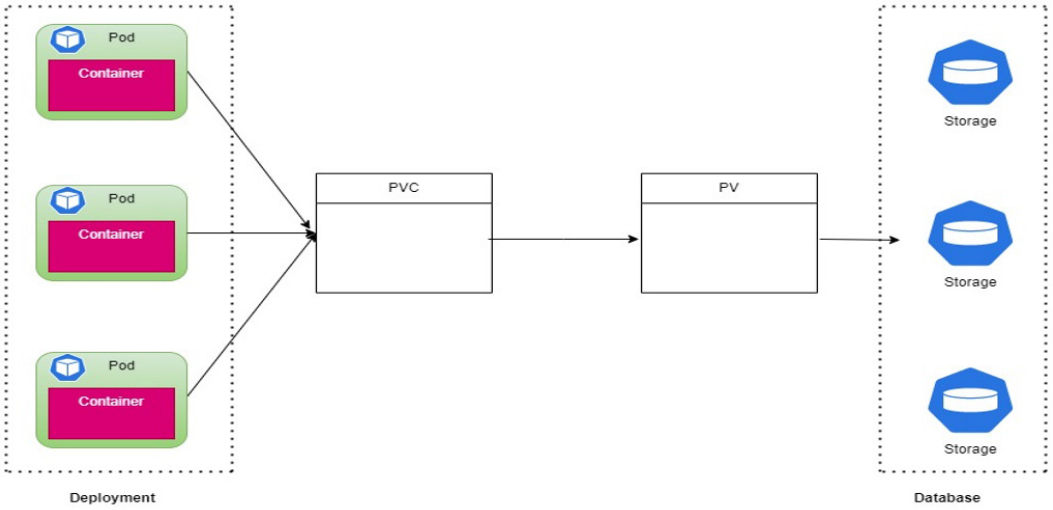


Figure 6.4: *Deployment on a database*

The StatefulSet object makes a big difference here and creates a separate PVC for each replica. PVC and PV connect the database with the application, as shown in *Figure 6.5*:

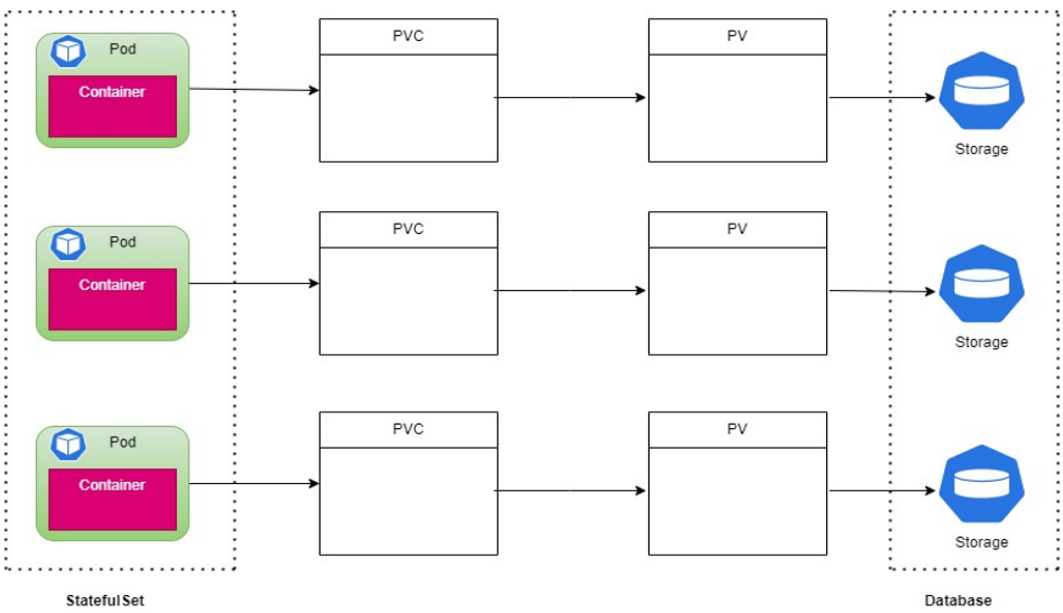


Figure 6.5: *StatefulSet connecting to a database*

Following is a sample yaml for a StatefulSet to create three replicas with a mongodb image:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mongodb
spec:
  serviceName: "mongodb-service"
  replicas: 3
  selector:
    matchLabels:
      app: mongodb
  template:
    metadata:
      labels:
        app: mongodb
    spec:
      containers:
      - name: mongo
        image: mongo
        command:
          - mongod
          - "--replSet"
          - rs0
        ports:
        - containerPort: 27017
          protocol: TCP
        volumeMounts:
        - mountPath: /data/db
          name: mongodb-persistent-volume-claim
  volumeClaimTemplates:
  - metadata:
      name: mongodb-persistent-volume-claim
```

```
  spec:
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 20Gi
```

Well, there are a lot of interesting fields here to discuss, as you can see. The **`serviceName`** here is that of the headless service, which we will talk about in a while, and **`volumeClaimTemplates`** is also an important specification that we will discuss.

Volume claim templates

In the case of singleton pods' templates, you can use **`PersistentVolumeClaims`**. In the case of StatefulSets, you can set **`.spec.volumeClaimTemplates`** in the StatefulSet specification itself to specify the required storage using **`PersistentVolumes`**. The **`PersistentVolumeClaims`** are then created by Kubernetes when you create the StatefulSet object. The **`VolumeClaimTemplate`** is analogous to the pod template used in workload objects like ReplicaSets and Deployments. One is used to provision pods, and the other one is used to create volume claims.

Headless service

In *Chapter 3, HTTP Load Balancing With Ingress,* we discussed Kubernetes Services, which are the Kubernetes objects used to route requests to the ephemeral pods. StatefulSet, being a special workload resource, has one more special demand. You need to create something called headless service to access the application managed by a StatefulSet. This is because you do not want load balancing and a single service IP. Instead, the need here is to get the IP addresses of all the pods inside the StatefulSet.

The pods in the StatefulSet communicate with each other because the pods rely on each other due to the ordered creation. Moreover, the data needs to be replicated among primary and secondary instances. The headless service does not give you load balancing; instead, when combined with StatefulSets, it gives us individual DNSs to access our pods, and in turn, a way to connect to all the MongoDB pods individually.

A headless service is a Kubernetes Service object with **`clusterIP`** set to None. Let us look at a sample yaml:

```
apiVersion: v1
kind: Service
metadata:
  labels:
```

```
    name: mongodb
  name: mongodb-service
spec:
  clusterIP: None
  ports:
  - port: 27017
    protocol: TCP
    targetPort: 27017
  type: ClusterIP
```

Installing MongoDB on Kubernetes using StatefulSets

The yamls we have seen so far are purposefully revolving around mongoDB. Let us now bring them all together. Let us first create a **`StorageClass`** to dynamically provision the storage using, let us say Azure SSD:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: azure-ssd-sc
provisioner: kubernetes.io/azure-disk
parameters:
  skuName: Premium_LRS
  location: eastus
  storageAccount: azure_storage_account_name
```

Here, parameters are specific to the provisioner, which is azure-disk. As you can see, they are different from the ones we saw in the earlier example of **`StorageClass`** for GCE persistent disk.

After **`StorageClass`**, we shall create a headless service called **`mongodb-service`** to get stable DNS names for the pods in StatefulSet. Then, we shall create a StatefulSet with three replicas of Mongo image.

You can use the yamls as is for creating headless service and StatefulSet that we saw as examples earlier. Headless service and StatefulSet are generally stored in the same yaml and created at the same time because the objects are dependent on each other.

That is mostly it. After StatefulSet is successfully created, you can see that the pods are created in an order. Refer to *Figure 6.6*:

```
D:\>kubectl get pods
NAME        READY   STATUS              RESTARTS   AGE
mongodb-0   0/1     ContainerCreating   0          34s

D:\>kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
mongodb-0   1/1     Running   0          58s
mongodb-1   1/1     Running   0          22s
mongodb-2   1/1     Running   0          15s
```

Figure 6.6: *Pods in a StatefulSet*

You can use the **`kubectl describe`** command to describe the StatefulSet and see the events in it.

Refer to *Figure 6.7* to see the events while creating the mongodb StatefulSet:

```
Events:
  Type    Reason           Age   From                  Message
  ----    ------           ----  ----                  -------
  Normal  SuccessfulCreate  27s   statefulset-controller  create Claim mongodb-persistent-volume-claim-mongodb-0 Pod mongodb-0 in StatefulSet mongodb success
  Normal  SuccessfulCreate  27s   statefulset-controller  create Pod mongodb-0 in StatefulSet mongodb successful
  Normal  SuccessfulCreate  22s   statefulset-controller  create Claim mongodb-persistent-volume-claim-mongodb-1 Pod mongodb-1 in StatefulSet mongodb success
  Normal  SuccessfulCreate  21s   statefulset-controller  create Pod mongodb-1 in StatefulSet mongodb successful
  Normal  SuccessfulCreate  16s   statefulset-controller  create Claim mongodb-persistent-volume-claim-mongodb-2 Pod mongodb-2 in StatefulSet mongodb success
  Normal  SuccessfulCreate  16s   statefulset-controller  create Pod mongodb-2 in StatefulSet mongodb successful
```

Figure 6.7: *Events in a StatefulSet*

You can observe that the **`PersistentVolumeClaims`** are created before creating the pods. Use **`kubectl get`** commands to check the existence of PVs and corresponding PVs.

One important thing in productionizing a MongoDB cluster is to add liveness checks to the Mongo containers. We learned in the chapter on pods that you can add liveness probe as a health check to determine whether a container is operating as expected. The Mongo tool itself provides a way to check the liveness of the container. We can add liveness probe, as follows, to the pod template within the StatefulSet object:

```
livenessProbe:
  exec:
    command:
      - /usr/bin/mongo
      - --eval
      - db.serverStatus()
  initialDelaySeconds: 10
  timeoutSeconds: 10
```

Just like other workload resources, the **`kubectl scale`** command can be used to scale the StatefulSet up and down.

Even after you delete the StatefulSet, PVCs and PVs are not deleted to ensure data safety. You can use shortform **`sts`** for StatefulSet in **`kubectl`** commands, as shown in *Figure 6.8*:

```
D:\>kubectl delete sts mongodb
statefulset.apps "mongodb" deleted

D:\>kubectl get pvc
NAME                                      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mongodb-persistent-volume-claim-mongodb-0   Bound    pvc-c1b4ef98-6900-41f5-8afd-4c77ef0bf951   50Mi       RWO            hostpath       10m
mongodb-persistent-volume-claim-mongodb-1   Bound    pvc-5d5fb1e4-640f-47be-a55a-84305f4ae66a   50Mi       RWO            hostpath       10m
mongodb-persistent-volume-claim-mongodb-2   Bound    pvc-059d1066-7aed-4feb-82b3-9f06ac460e1e   50Mi       RWO            hostpath       9m54s

D:\>kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
REASON   AGE
pvc-059d1066-7aed-4feb-82b3-9f06ac460e1e   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-2
         9m58s
pvc-5d5fb1e4-640f-47be-a55a-84305f4ae66a   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-1
         10m
pvc-c1b4ef98-6900-41f5-8afd-4c77ef0bf951   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-0
         10m
```

Figure 6.8: *Deleting a StatefulSet*

To clean up the volumes and free the storage, you can just delete PVCs, and the corresponding PV will be automatically deleted. Why is that so? If you said "because **`reclaimPolicy`** for PV is **`Delete`**", give yourself a pat on the back as you remembered this detail just fine.

Refer to *Figure 6.9*; you can observe that the number of PVs that is the output of the **`kubectl get pv`** command is automatically reduced upon the deletion of a PVC:

```
D:\>kubectl get pvc
NAME                                      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   AGE
mongodb-persistent-volume-claim-mongodb-0   Bound    pvc-c1b4ef98-6900-41f5-8afd-4c77ef0bf951   50Mi       RWO            hostpath       10m
mongodb-persistent-volume-claim-mongodb-1   Bound    pvc-5d5fb1e4-640f-47be-a55a-84305f4ae66a   50Mi       RWO            hostpath       10m
mongodb-persistent-volume-claim-mongodb-2   Bound    pvc-059d1066-7aed-4feb-82b3-9f06ac460e1e   50Mi       RWO            hostpath       9m54s

D:\>kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
REASON   AGE
pvc-059d1066-7aed-4feb-82b3-9f06ac460e1e   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-2
         9m58s
pvc-5d5fb1e4-640f-47be-a55a-84305f4ae66a   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-1
         10m
pvc-c1b4ef98-6900-41f5-8afd-4c77ef0bf951   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-0
         10m

D:\>kubectl delete pvc mongodb-persistent-volume-claim-mongodb-2
persistentvolumeclaim "mongodb-persistent-volume-claim-mongodb-2" deleted

D:\>kubectl get pv
NAME                                       CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   CLAIM
REASON   AGE
pvc-5d5fb1e4-640f-47be-a55a-84305f4ae66a   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-1
         27m
pvc-c1b4ef98-6900-41f5-8afd-4c77ef0bf951   50Mi       RWO            Delete           Bound    default/mongodb-persistent-volume-claim-mongodb-0
         27m
```

Figure 6.9: *Deleting a PVC*

Disaster recovery

A **`PersistentVolume`** statically created has a **`'Retain' reclaimPolicy`** by default. This is the safest option. If you use **`Delete`** as **`reclaimPolicy`**, then the **`PersistentVolume`** is deleted upon the deletion of the corresponding

PersistentVolumeClaim. Not everyone will like this automatic behavior, especially if the volume contains valuable data. Thus, it is safer to use the **Retain** policy. With the **Retain** policy, upon deletion of PVC, the corresponding PV will not be deleted. Instead, the PV is moved to the **Released** phase, from where you can manually recover the data.

PersistentVolumes that are dynamically created by a **StorageClass** will inherit the reclaim policy specified in the class. If no **reclaimPolicy** is specified when a **StorageClass** object is created, the default value used is **Delete**.

PersistentVolumes that are manually created and then managed via a **StorageClass** will have the reclaim policy that was used at the time of creation.

You can use the **kubectl patch** command, as follows, to change reclaim policy for your **PersistentVolume**:

```
kubectl patch pv pv-name -p
'{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

The **kubectl patch** command is very useful for modifying the API objects in place.

Container storage interface

It is very important to know about **Container Storage Interface** (**CSI**) when discussing storage in the world of Kubernetes. CSI is developed as a standard to enable storage vendors to develop storage plugins that are compatible with container orchestration systems like Kubernetes. Let us understand more about CSI to see how this adds to extensible nature of Kubernetes.

While discussing **StorageClass**, we saw that provisioner is one of the mandatory specifications that tells which volume plugin to be used for provisioning **PersistentVolumes**.

In Kubernetes, there are two categories of volume plugins:

- The first type of plugins is called **InTree** volume plugins, which are part of Kubernetes, meaning that their code is part of the Kubernetes code and hence, the provisioner name starts with **kubernetes.io**. These plugins face a tight coupling problem where the development is coupled with the Kubernetes release process. This makes it challenging for new vendors to add support for new storage systems. Moreover, as the plugin code is part of Kubernetes codebase, maintenance of Kubernetes code becomes more and more challenging.
- The second category is **OutOfTree** volume plugins. The code for these plugins is, as the name suggests, outside Kubernetes. Here, third-party storage providers control the release of the new volume plugins that can be

used in Kubernetes without the need for any changes in Kubernetes code. The standard used is called CSI, and it makes the Kubernetes volume layer extensible. **`OutOfTree`** plugins follow the standard for container storage, which is CSI. Another important advantage for storage vendors with CSI plugins is that the plugin can be used for all container orchestration systems, such as Docker Swarm, and not just for Kubernetes.

The CSI plugin is deployed as two components: a CSI controller component that contains controller implementation of the CSI driver and a node component that provides functions to mount the volume on a pod. The controller component is deployed on a node, and it has functions to control the storage, create and delete a volume, and so on. Node component runs as DaemonSet, meaning it is deployed on every node in the Kubernetes cluster.

A CSI plugin can be used for dynamic provisioning and static provisioning of the storage in Kubernetes. For dynamic provisioning, a provisioner field specific to the volume plugin corresponding to the chosen CSI driver should be used. Parameters required for the provisioner need to be set just like any other **`StorageClass`**.

To use CSI with static provisioning, the Kubernetes admin should provision some volumes with **`PersistentVolume`** objects containing the CSI specific fields. The CSI plugins documentation should be referred to for the details.

Conclusion

Storage can be provisioned manually or dynamically in Kubernetes using volumes. You create Kubernetes API objects like **PersistentVolume** (**PV**), **PersistentVolumeClaim** (**PVC**) and **StorageClass** to make the storage available to Pods, Deployments, ReplicaSets and so on. To deploy a stateful application, StatefulSets are used, which guarantees ordered creation and deletion of pods. Storage vendors can develop plugins to support their storage systems in Kubernetes, and they follow a standard called Container Storage Interface for developing such plugins. CSI plugins are not part of Kubernetes code, and their release cycle is independent.

Points to remember

- When there are multiple worker nodes or if the cluster is autoscaling, where nodes come and go, you want to use **`NodeAffinity`** to ensure that the **`hostPath`** volume is set up on the correct node.
- A volume can only be mounted using one access mode at a time. For example, a **`GCEPersistentDisk`** can be mounted as **`ReadWriteOnce`** by a single node or **`ReadOnlyMany`** by many nodes, but not at the same time.

- Pods created through a StatefulSet are created and removed in a fixed order. A headless service is required to provide DNS names for the pods managed by a StatefulSet.
- When a StatefulSet is deleted or scaled down, associated volume is not deleted. This helps ensure data safety.
- Use **`livenessProbe`** and **`readinessProbe`** on StatefulSet pods to check whether a pod is alive and whether it is ready to serve the traffic.
- CSI is the standard for developing storage plugins to support external storage systems in Kubernetes.

Questions

1. The data under a **`PersistentVolume`** is deleted when the PV object is deleted from Kubernetes API. How do you ensure that the data is not deleted upon deletion of PV object?
2. You want to deploy and use Cassandra as a database to store the details of the transactions performed daily through a web application deployed in a Kubernetes cluster. How would you set up Cassandra on the same cluster?

Answers

1. Check the reclaim policy for the **`PersistentVolume`** object and set it to **`Retain`**. This generally happens for dynamically provisioned volumes, where the default reclaim policy could be **`Delete`**, which should be modified to **`Retain`**.
2. Use a StatefulSet to create a Cassandra ring containing multiple pods of Cassandra. Create a headless service for DNS lookups between Cassandra pods and other services in your cluster.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 7
Introduction to Service Discovery

Introduction

Kubernetes is a dynamic system with numerous moving parts. For example, pods are ephemeral in nature; they come and go. This could be either due to some network/infrastructure issue, by design (*due to scaling parameters*) or by the nature of the pods, for example, spot nodes (on cloud) can give short life pods by infra design itself. This nature of Kubernetes, generally, helps in easy and efficient hosting, and does provide lot of flexibility to applications however it does pose the challenge of discovering and utilizing the newly created dynamic components. (like new pods added due to scaling). It becomes essential to have a well-constructed strategy to quickly identify the newly created and destroyed components to reduce delay for the smooth running of modern workloads within SLAs. For instance, the newly created pod should be quickly added to the endpoint, which accepts processing requests.

Structure

In this chapter, we will discuss the following topics:

- What is service discovery?
- Discovery patterns
    - Client-side discovery pattern

    - Server-side discovery pattern
- Service registry
- Registration patterns
    - Self-registration
    - Third-party registration
- Service discovery in Kubernetes
    - Service discovery in Kubernetes using etcd
    - Service discovery in Kubernetes using Kube proxy and DNS
        - Service objects
- Advance details
    - Endpoints
    - Manual service discovery
    - Cluster environment variables
    - Kubeproxy and cluster IPs

Objectives

In this chapter, you will learn what service discovery is in general look at and the design strategies used in the market to address the aspects of service discovery You will see the Kubernetes way of accomplishing service discovery and the various patterns that are common in the Kubernetes community, for example, service discovery using etcd, service discovery using Kubeproxy and DNS, and service discovery via third-party registries. In the last section, you will see understand essential concepts related to endpoints, manual service discovery, and cluster environment variables.

What is service discovery?

Service discovery is a mechanism by which applications and (micro)services find each other's locations on the network. Implementation includes both a server that is a centralized place which maintains a view of all discoverable services, and a client that connects to the server to discover a service (get and update the addresses). The concept of service discovery is old, with its own evolution story with the evolution of other computer architectures. Earlier, when different computers needed to locate each other, it was done through a single test file, **`HOST.TXT`**. The addresses were added manually to this file. This worked well because new hosts were added infrequently. With the current situation being drastically different due to the internet, hosts are

added at an increasing rate, so an automated and scalable system is needed. This led to the invention and widespread adoption of DNS.

Today, the concept of service discovery is continuously evolving, thanks to microservices. With microservices, addresses are added, deleted, and constantly changed as new hosts are added or existing ones are decommissioned. This transient nature of microservices has forced enterprises to adopt different and probably a more efficient ways for service discovery.

Take a look at *Figure 7.1* to understand the complexities further:

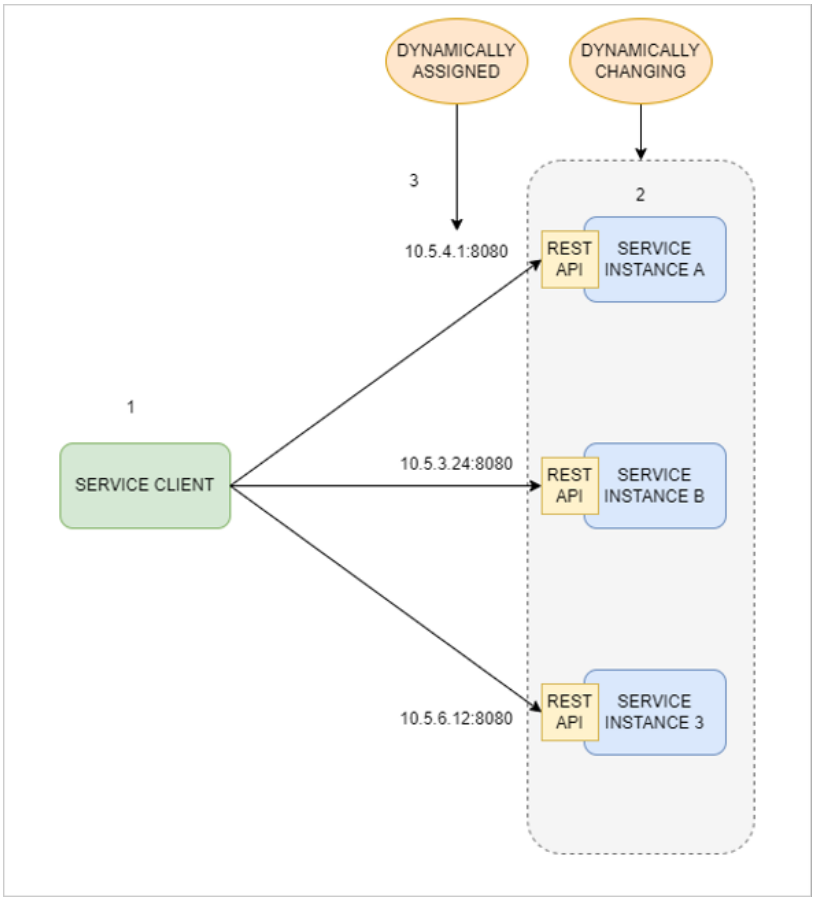


Figure 7.1: *Service Discovery for modern applications*

Refer to the numerical labeling in *Figure 7.1* with the corresponding numerical explanations, as follows:

1. The service client is the component that triggers REST API over HTTP(s). Your client code should have the required service discovery mechanism to get the exact IP and port for a service instance.

2. The REST API clients attached to each service instance are dynamically changing.
3. Since the service instance is constantly changing, so are the IP:port specifications. Service clients willing to connect to these APIs should have a way to tackle the dynamic locations.

To tackle the preceding issue, there are two strategies/design patterns used in the industry:

- Client-side discovery pattern
- Server-side discovery pattern

Let us look at each of the mentioned patterns one by one.

Client-side discovery pattern

In the case of a client-side discovery pattern, a client (component calling REST API) is responsible for determining the network locations (IP and port) of available service instances. In case you have multiple instances of service being available, the client should know of all the instances and be able to load balance across the instances. These details of instances are kept in a database known as **service registry**, and the client queries the database and applies a load balancing logic to finalize the network location for the service instance. Take a look at *Figure 7.2*:

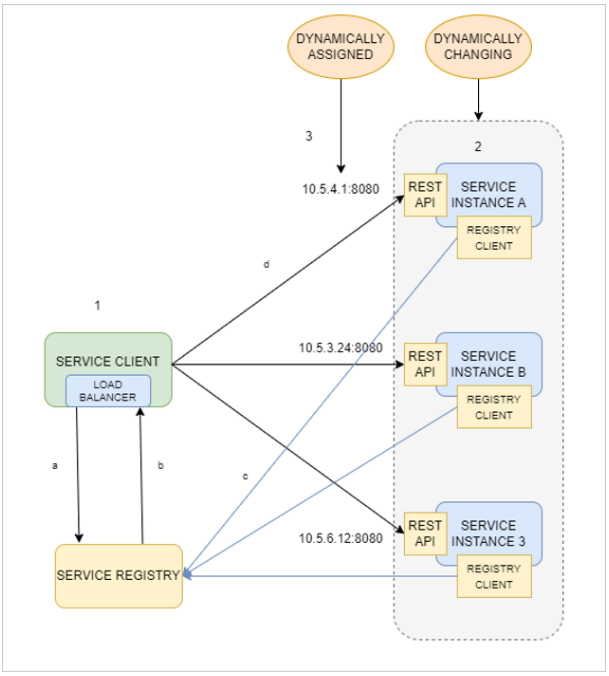


Figure 7.2: *Client-Side Discovery Pattern*

Refer to the numerical labeling in *Figure 7.2* with the corresponding numerical explanations, as follows:

1, 2, & 3: The explanation is identical to what you had already seen in the description of *Figure 7.1*:

a. Service client queries the service registry to get the network location of all service instances.

b. Network location of all service instances is returned. In this case, locations, that is, 10.5.4.1:8080, 10.5.3.24:8080, and 10.5.6.12:8080, will be returned.

c. Whenever a new service instance comes up, the registry client component of the service instance registers itself with the service registry. This way, the service registry, at any point in time, has complete information on all the instances of services that are running. An entry is deleted from the registry when the service instance terminates. A termination of instance is generally identified using the heartbeat mechanism.

d. Once the service client finalizes an instance of service, the API call will be made by applying the load balancing algorithm, and eventually, the final call will be made.

Netflix OSS is an example of a client-side discovery pattern, where Netflix Eureka is the service registry. Netflix ribbon acts as an IPC client that works with Eureka for balancing requests across all available service instances.

Pros of this type of service discovery are straightforward, easy to implement, and a low number of moving parts, making the system less error prone. Moreover, the dedicated load balancer enables exact identification of the service instance, where the call is going to be made. On the other hand, the service registry and service client are coupled. The service client must discover the service registry, and that discovery might need different implementations based on the type of service client.

Server-side discovery pattern

Another approach for service discovery is server-side discovery pattern. Take a look at *Figure 7.3*:

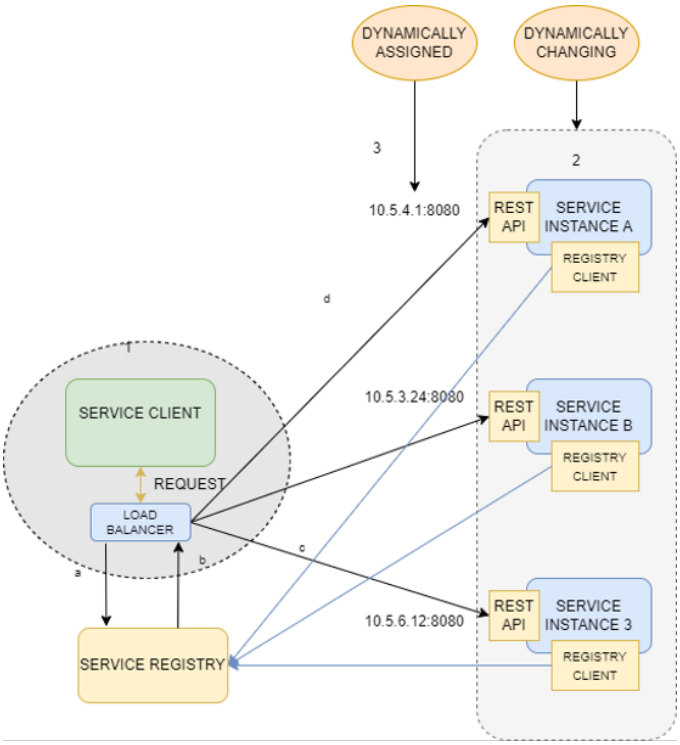


Figure 7.3: *Server-side discovery pattern*

Figure 7.3 is very similar to *Figure 7.2*, with little difference marked with a circle. Instead of the service client identifying the exact network location of the service, by querying the service registry and applying load balancing logic, the service client submits a request to the load balancer. The load balancer must query the service registry and then send the request to the actual service instance.

A system like Kubernetes uses a server-side discovery pattern. Kubernetes run a proxy on each host in the cluster, and this proxy plays the role of a server-side load balancer. To request a service, a client routes the request through the proxy using the host's IP address and the assigned port of the service.

One of the key advantages of using this discovery pattern is that clients must request a load balancer, so client implementation becomes simple. This eliminates the dependency to implement service discovery logic for each programming language and framework your service clients use. On the flip side, a disadvantage is the need to have a load balancer provided by the environment.

Service registry

You have seen the service registry's role in making services discoverable. In this section, you will deep dive more into service registry and how service instances register themselves to a service registry.

At the core, the service registry is a database that contains network locations of all the instances of services. Hence, the service registry must be highly available and scalable, and it must support continuous updates. The updates made to the service registry should also maintain high data consistency. A microservice generally registers its network location when it is initialized. The service registry keeps updating its registration continuously via the heartbeat mechanism, and when the microservice instance terminates, the registration is removed from the registry.

Netflix Eureka is a service registry option with a REST API for registering and fetching Service instance network locations. A service instance registers itself via a **`POST`** request, and every 30 seconds, it refreshes its registration using the **`PUT`** request offered by Eureka. A registration is removed when the service instance reaches a timeout during the heartbeat operation. Generally, multiple service registry instances are created to support high availability and reduce latency.

Here are a few other examples of service registries:

- **`Etcd`**: **`etcd`** is a distributed, consistent, highly available, keyvalue store that is used for service discovery and shared configuration (already seen in the previous chapters). You will investigate the service discovery side of this option later in the chapter. Two notable projects that use etcd are Cloud Foundry and Kubernetes.
- **Consul**: It is another famous tool for configuring and discovering services. It provides an API that allows clients to register and discover services very similar to Eureka, as described previously. Consul provides a facility to perform health checks and determine service availability.
- **Apache Zookeeper**: Apache Zookeeper is a widely used, highperformance coordination service for distributed applications. It was initially developed under the Hadoop ecosystem subproject but also works well for other use cases.

Though there is no depth to which one can dive while exploring the implementation side of the service registry, for the sake of this book, having a basic understanding helps the audience appreciate the upcoming sections.

Registration patterns

There are two patterns when it comes to service instances trying to register its location with service registry, and they are described as follows.

Self-registration pattern

As the name suggests, when utilizing this pattern, a service instance is responsible for registering and de-registering itself with the registry. Service registry generally assigns a time out to registration, and the service instance is responsible for sending periodic heartbeats to save its registration from expiring. A prime example of this pattern is Netflix Eureka.

One benefit of using this pattern is the simplicity and the services needed to trigger APIs. Conversely, each service must have the implementation to call the APIs. The alternative approach, which decouples services from the service registry, is the third-party registration pattern.

Third-party registration

When using this registration pattern, service instances do not register themselves. Instead, another component known as the service registrar keeps track of changes to the set of running instances by polling or subscribing to events. When a service registrar notices a newly created service instance, it registers it with the service registry. Registrar also deregisters the terminated services. An example of this pattern is the open-source project **`Registrator`**. It registers and deregisters service instances automatically, which are then deployed as Docker containers. It supports registries like **`etcd`** and **`consul`**.

The major advantage of this pattern is the loose coupling of services and service registry. Services do not need to implement logic to register/deregister themselves. The main disadvantage is the management of another highly scalable and available system, which has its management issues to be tackled.

Till now, you have investigated the basics of service discovery as a concept and the various design patterns followed to handle changes to services instances. In the next section, we will focus on how Kubernetes handles service discovery aspects.

Service discovery in Kubernetes

To better understand service discovery in Kubernetes, let us first draw an analogy between Kubernetes and more traditional architecture. Pods are interpreted/perceived in multiple ways by multiple people, but when it comes to networking, the perception of pod is pretty common across the board.

The Kubernetes documentation says that *Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration*.

Just like you drew an analogy between pods and instances, there is another analogy between traditional and Kubernetes services. In traditional architecture, instances are grouped to serve as one service; and just like that, in Kubernetes, multiple pods serve a service behind the scenes. Official documentation describes service as *an abstraction that defines a group of pods and a policy to access them (sometimes this pattern is called a micro-service).*

Making the analogy more powerful, a set of pods making up a service should be considered ephemeral, as neither the total pods nor the IP address of pods is final at any given time for a service. Hence, in Kubernetes, the issue of providing a reliable discovery mechanism holds.

There are primarily two ways in which the service discovery aspect is taken care of in Kubernetes. You will be going through both approaches, at different points in Kubernetes architecture. You looked at these concepts in *Chapter 3, HTTP Load Balancing Using Ingress*.

Service discovery using etcd

Pods are ephemeral. They come and go. Hence, the number of pods and their IP address keep changing. A service is a group of similar pods with the same software running inside it. The whole idea here is that a request is accepted by Kubernetes service to process, and how is this request directed to a pod.

One way that Kubernetes provides service discovery is via querying the **`etcd`** database using the endpoints API of Kubernetes. A client (trigger a request) can

query the pods running in the cluster for network location (IP and port) and place the call directly to the pods. Consider *Figure 7.4*:

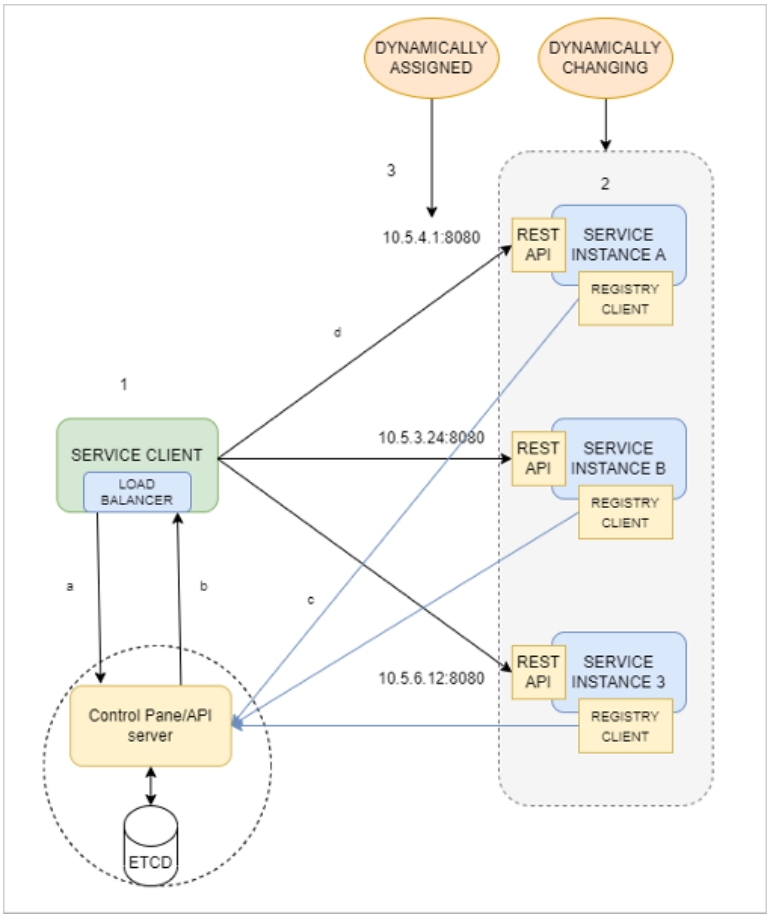


Figure 7.4: *Service Discovery via etcd*

Figure 7.4 is a copy of *Figure 7.2*, with changes in the dotted oval section. The API server plays the role of service registry, which provides the network location for service instances via querying the **`etcd`** datastore. The rest of the interaction remains the same. The **`etcd`** database has the updated information of all endpoints and is kept up to date via Kubernetes itself.

Calling the registry and contacting the K8s API server is the client's responsibility. Once the client has the information of all service instances, the client applies load balancing logic to create one network location and finally, calls the service on the selected network location. The logic to get the instances of service must be implemented in the client.

Kubernetes provides client libraries in various languages. For example, for Java, you can visit **https://github.com/kubernetes-client/java/releases** for more details.

You can install the preceding client using the following commands:

```
# Kubernetes client  java library
git clone --recursive https://github.com/kubernetes-client/java

# Installing project artifacts, POM etc:
cd java
mvn clean install
```

Once you have the preceding code built, you can mention this in the pom of your service client. Following is the code snippet that fetches all the pods for a service and gives you a hook to apply a load balancer algorithm to come up with one pod (network location) to make the final call. For running this code, appropriate security permissions and cluster role bindings should be present. A better way to handle this would be to package this code as pod and run it as a Kubernetes workload.

```
import io.kubernetes.client.ApiClient;
.
.
import java.io.IOException;

public class KubeConfigFileClientExample {
  public static void main(String[] args) throws IOException,
ApiException {

    // file path to your KubeConfig
    String kubeConfigPath = "~/.kube/config";

    // loading the out-of-cluster config, a kubeconfig from file-system
    ApiClient client =
        ClientBuilder.kubeconfig(KubeConfig.loadKubeConfig(new
FileReader(kubeConfigPath))).build();

    // set the global default api-client to the in-cluster one from above
```

```
17.     Configuration.setDefaultApiClient(client);
18.
19.     // the CoreV1Api loads default api-client from global
    configuration.
20.     CoreV1Api api = new CoreV1Api();
21.
22.     // invokes the CoreV1Api client
23.     V1PodList list = api.listPodForAllNamespaces(null, null, null,
    null, null, null, null, null, null);
24.     System.out.println("Listing all pods: ");
25.     for (V1Pod item : list.getItems()) {
26.       System.out.println(item.getMetadata().getName());
27.      // Apply load balancing logic on each item
28.     }
29.   }
30. }
```

Let us see what the preceding code does line-by-line:

- **Lines 1 to 4:** These are the import statements that import all the packages needed for the libraries used in the code.
- **Line 10:** It reads the **`kubeconfig`** file to create a connection to the Kubernetes cluster.
- **Line 23:** This code line fetches all the pods running across namespaces. The **`listPodForAllNamespaces`** method could be passed with appropriate arguments to get a more streamlined response for pods. In the code, everything is passed as null, as you want all the pods to be returned.
- **Lines 26 and 27:** Line 26 is printing all the pods. This whole list could be passed to a custom load balancing logic, and the outcome of this load balancing logic will give one network location, that is, IP and port for one pod.

As you can see, the biggest advantage is that you do not need to set up the registry. Kubernetes does half of the things for you, that is, registering the new service instances. However, the disadvantage is that the client must implement the logic and load balancing to identify one pod to finally send the REST API call to.

Service discovery in Kubernetes via Kubeproxy and DNS

In Kubernetes, you can define a service (a Kubernetes object) to abstract the underlying group of pods running code for a service. These service objects are valid DNS names and identify the pods of a service via applying the label selector construct. You have already seen a discussion in *Chapter 3, HTTP Load Balancing and Ingress*. The DNS record of a service is of type **`<service-name>.<service-namespace>`**, and since this name is constant for the service definition, it could be very quickly used by other services or clients. This single well-known address for a service removes the need for a service discovery logic on the client side.

Kubernetes, apart from having a DNS, introduces one more IP address for each service. This IP address is called the **`clusterIP`** (this is not the Cluster IP Service type). Just like you used the DNS name, a cluster IP could direct calls to a set of pods.

DNS is not a hard requirement for Kubernetes applications. A client can always get the cluster IP of a service by inspecting their environment variables. When a pod is initiated for a service, Kubernetes injects a few variables as environment variables. The structure of the environment variable is **`<service-name>_SERVICE_HOST and <service-name>_SERVICE_PORT`**.

Look at *Figure 7.5* to understand the relationship between cluster IP, DNS, and Service:

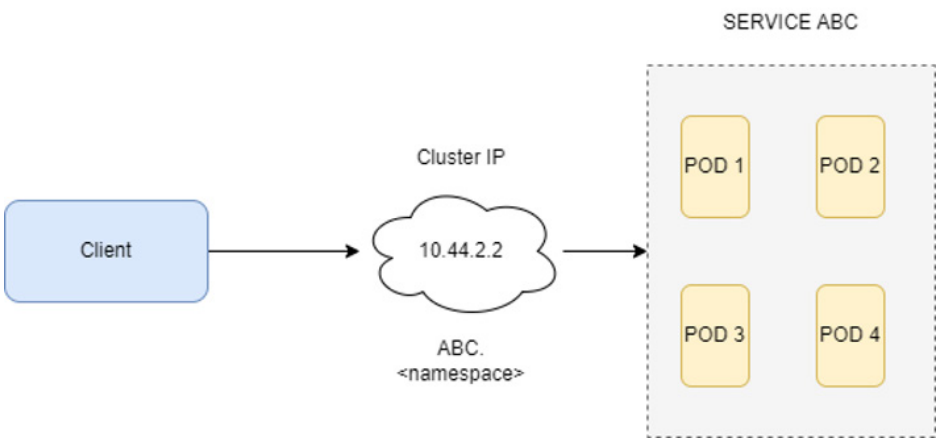


Figure 7.5: *Cluster IP*

In the preceding figure, a service ABC is deployed in Kubernetes having four pods. This service is associated with a DNS name of the form **`ABC.<namespace>`**. Along with the namespace, Kubernetes assigns a Cluster IP as the virtual address to the pods. Kubeproxy creates and manages this virtual IP for services other than the

external DNS name. As you already know, Kubeproxy is a component that runs on every node in the Kubernetes cluster. Kubeproxy operates at the network layer and transparently substitutes the cluster IP with an IP address of the pod using Linux constructs like iptables or **IP Virtual Server** (**IPVS**). Hence, Kubeproxy is an essential component of service discovery and load balancing in the cluster. Consider *Figure 7.6* to understand the discussion further:

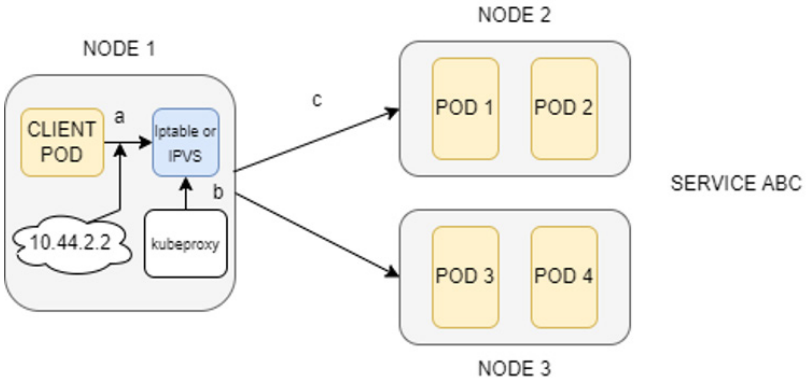


Figure 7.6: *Service Discovery via Kubeproxy*

Refer to the numerical labeling in *Figure 7.6* with the following corresponding alphabetical explanations:

a. Client calls the cluster IP.

b. **`kubeproxy`** works on the network layer using Linux capabilities like iptables or IPVS and automatically substitutes the destination **`clusterIP`** with an IP address of some service's Pod.

c. The client makes a call to the actual network location of the service instance.

In this server-side service discovery approach, a client accesses a single endpoint, that is, a stable static IP address or DNS, and there is no logic on the application side to identify the pod address from static IP/DNS. Simultaneously, as Kubeproxy is run on each node in the cluster, service discovery and load balancing occur on each cluster node. A node that is closer to the client is picked up, and Kubeproxy running on the node serves the purpose. The above described form of service discovery relies on the Linux network stack it is also called network-side service discovery. For other operating systems, more exploration is need to achieve this kind of service discovery; there are not enough examples available in industry who claim of using network-side service discovery strategy

Service objects

Service objects in Kubernetes provide yet another way to discover services, where a **`kubectl run`** command is used to create Kubernetes deployment, and this deployment is exposed via the **`kubectl expose`** command to generate a service DNS of type **`<service-name>.<namespace>`**. Consider the following command, which creates two service objects, i.e., **`tiger-eur-prod`** and **`lion-nam-prod`**:

```
1. kubectl run tiger-eur-prod \
2. --image=gcr.io/handson-k8s-demo/handson-k8s-tiger:v1 \
3. --replicas=3 \
4. --port=8080 \
5. --labels="ver=1,app=tiger,env=eur-prod"
6. kubectl expose deployment tiger-eur-prod
7. kubectl run lion-nam-prod \
8. --image=gcr.io/handson-k8s-demo/handson-k8s-lion:v2 \
9. --replicas=2 \
10. --port=8080 \
11. --labels="ver=2,app=lion,env=nam-prod"
12. kubectl expose deployment lion-nam-prod
```

Lines 1 to 5 create a Kubernetes deployment of **`name tiger-eur-prod`**. Line 6 exposes the deployment **`tiger-eur-prod`** as a service. Lines 7 to 12 create a deployment **`lion-nam-prod`** and expose it as a service.

Suppose you run the following command:

```
kubectl get services -o wide
```

This will list down your service object created earlier. The output will look as follows:

```
NAME              CLUSTER-IP              PORT(S)          SELECTOR
tiger-eur-prod    10.115.245.13    8080/TCP  ver=1,app=tiger,env=eur-prod
lion-nam-prod     10.115.242.3     8080/TCP   ver=2,app=lion,env=nam-prod
kubernetes         10.115.240.1     443/TCP          <none>
```

As the output shows, though you created two services, i.e., **`tiger-eur-prod`** and **`lion-nam-prod`**, a third service named Kubernetes got created automatically. This third service lets you talk to Kubernetes API from within the app.

If you observe the **SELECTOR** column, you will see that **tiger-eur-prod** gives a name to a selector and mentions which ports to talk to for the service. The **kubectl expose** command pulls label selector and port from the deployment definition. The service created earlier has a cluster IP and a virtual IP attached. This is the same IP you saw in depth in the previous section of the chapter. As stated earlier, this IP will load balance across all the pods that match the sector criteria.

You can interact with this service by port forwarding to any one of the **tiger-eur-prod** pods with the following commands on the terminal:

```
TIGEREUR_POD=$(kubectl get pods -l app=tiger-eur-prod \
-o jsonpath='{.items[0].metadata.name}')
kubectl port-forward $TIGEREUR_POD 7070:8080
```

You can access the service with this URL: **http://localhost:7070**.

DNS

Kubernetes installs a DNS service at the time of creation of the cluster. This DNS service is exposed to all the pods running in the cluster. Cluster IP, the stable virtual address created, is assigned a DNS name. A DNS name assigned is of the following form:

```
tiger-eur-prod.default.svc.cluster.local
```

Let us break this complete DNS name to understand what it means:

- **tiger-eur-prod:** The name of the service
- **Default:** The namespace that this service is in
- **Svc:** Recognizing that this is a service
- **cluster.local:** The default base domain name for the cluster, which could obviously be configured by the administrator during cluster creation

While referring to a service within the same namespace, you can just use the service name. However, when you intend to use a service in a different name space, you must mention the namespace as well, that is, **tiger-eur-prod.default**. You can use a fully qualified DNS name as well.

Readiness checks

When a Kubernetes application starts up, there is generally a delay between the time of starting and the time when the application (a pod) can serve the traffic. This delay could be anywhere between a few seconds to a few minutes. One facility your service object provides is to track which of your pods are ready to serve the traffic via the readiness check.

You can use the following command to edit a deployment:

```
kubectl edit deployment/tier-eur-prod
```

Add the following YAML snippet in the file (**`tier-eur-prod`**):

```
1.          spec:
2.            containers:
3.            ...
4.            name: tiger-eur-prod
5.            readinessProbe:
6.              httpGet:
7.                path: /ready
8.                port: 8080
9.              periodSeconds: 2
10.             initialDelaySeconds: 1
11.             failureThreshold: 4
12.             successThreshold: 2
```

Lines 5 to 12 introduce a new block with a tag readiness probe. This will enable the pods with a readiness check when they are created. The pods will be checked via HTTP **`GET`** to/ready on port **`8080`**. This check is done every 2 seconds, starting after 1 second of the pod creation request. If four successive checks fail, the pod will be considered unready, and no traffic will be served to the pod. However, if two requests are succeeded, pod will be regarded as ready.

The readiness check is a great way to signal that the pod does not want to entertain any further traffic. This could be in two cases: when you intend to perform a graceful shutdown and when something terrible is happening internally in the pod.

Advance details

In this section, we will discuss advanced details about topics we have already see in action. These are cluster environment variables and Kubeproxy and Cluster IPs. We will also cover two more topics that let you connect to a service in a different way, that is, view endpoints and how a manual service discovery can happen.

Endpoints

You had seen so many details around the Kubernetes cluster and deployed multiple Kubernetes objects but have yet to hear about the term endpoints. In Kubernetes

services, you saw that you could use labels to match service to pods. The service can send traffic to the newly created pod with a specific label. A service accomplishes this mapping based on a selector by adding this mapping to an endpoint. The endpoints track and maintain the IP address of the object where the traffic must be sent. When a selector label matches a pod label, the pod IP address is automatically added to endpoints.

You can use the following command to get the details about endpoints:

```
kubectl get endpoints
```

The output of the preceding command will be as follows:

```
NAME                      ENDPOINTS                                   AGE
tiger-eur-prod       10.48.7.145:8081,10.48.7.145:8080                 1d
lion-nam-prod          10.48.14.100:8080                               1d
```

As you can see, the **tiger-eur-prod** service has two pods with network locations **10.48.7.145:8081** and **10.48.7.145:8080**. A key point to note is that each pod IP ultimately has an endpoint created and aligned to a service.

Let us say you do not give the selector while creating a service, and then you can manually create these endpoints and attach them to the service. One might wonder why anybody would do that, but there are practical situations where it is needed. One such situation could be to have a resource to access an external service that does not reside on the Kubernetes cluster. You can configure your endpoints, and they create a service utilizing the endpoint. A good example could be an external database service for your container application.

For example, let us assume that you have a web application running on network location **192.168.7.50:80**. Let us first create an endpoint:

```
1. apiVersion: v1
2. kind: Endpoints
3. metadata:
4.   name: external-app
5. subsets:
6.   - addresses:
7.     - ip: 192.168.7.50
8.     ports:
9.     - port: 80
```

Once you create the preceding endpoint, you can use it in the service yaml:

```
1. apiVersion: v1
2. kind: Service
3. metadata:
4.   name: external-app
5. spec:
6.  ports:
7.     - protocol: TCP
8.       port: 80
9.       targetPort: 80
```

Refer to the two files in code shared with the chapter. Use the following commands to deploy the preceding yaml:

kubectl apply -f ext-endpoint.yaml

kubectl apply -f ext-endpoint-service.yaml

This concept, that is, to define a Kubernetes Service on top of an external service, is needed when you are deploying an application that refers to an external service. This external service will one day be migrated to Kubernetes cluster.

Manual service discovery

Kubernetes services are built on top of **SELECTORS**. You can use Kubernetes API to discover the pods and send the request to one of the pods.

kubectl get pods -o wide --show-labels -selector= app=tiger-eur-prod ,env=prod

```
NAME ...                                    IP ...
tiger-eur-prod -3408831585-bpzdz ...    10.112.1.54 ...
tiger-eur-prod -3408831585-kncwt ...    10.112.2.84 ...
tiger-eur-prod -3408831585-l9fsq ...      10.112.2.85 ...
```

Now, you can loop over this list of pods and trigger request to one of the pods. This is like the example we saw in *service discovery* using **etcd**. There too, we did the same thing; the only difference was that instead of a Java client, shell is used to demonstrate show manual service discovery.

Cluster IP environment variables

Generally, most users use Service DNS to get the cluster IP. However, there are some old ways still in use. One such old mechanism is injecting a set of environment variables into a pod at the time of pod creation and initialization. Consider that you have a back-end service running, having cluster IP as **`10.44.2.3`** running on port **`8080`**. If you want this cluster IP to be used by another front-end service, you can mention details of the back-end service while creating pods of the front-end service:

```
1. apiVersion: extensions/v1beta1
2. kind: Deployment
3. ...
4.     spec:
5.       ...
6.       containers:
7.       - name: frontend
8.         ...
9.         env:
10.        ...
11.       - name: API_URL
12.          value: 10.44.2.3:8080
13.       ...
```

The preceding is a very simple front-end service pod definition where you can pass the cluster IP of the back-end service as environment variables. This approach, while being correct technically, has some issues. The obvious ones are that the back-end service must be created before the front-end service. This might sound simple, but it can become exponentially complex as the application grows in size. Additionally, it might seem strange to many users to use environment variables. Hence, DNS is probably a much better solution.

Kubeproxy and cluster IPs

In this last section of the chapter, it is time to visit the role of kubeproxy and cluster IP play together to support service discovery. You had already seen this at high level when we discussed about service discovery using kubeproxy and DNS. Refer to *Figure 7.7:*

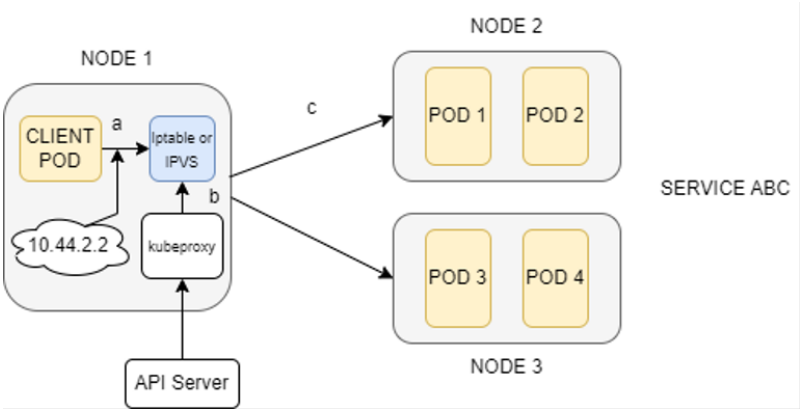


Figure 7.7: *Role of kubeproxy and cluster IP*

Figure 7.7 is an enhanced version of *Figure 7.6*, with one extra component added, that is, API server. For an explanation of alphabet labelling, refer to the previous sections.

In this diagram, the **`kubeproxy`** component watches for new services in the cluster through the API server. Kubeproxy then creates IP tables' rules in the kernel of the host machine to rewrite the destinations where the packets will be sent, that is, one of the pods of service ABC.

If the endpoints of the services change, **`iptables`** are rewritten. Even the cluster IP is assigned an IP by the API server, only when the service is created. However, while creating a service, you can specify a cluster IP by yourself. These cluster IPs cannot be modified without deleting and recreating the service.

Conclusion

Kubernetes is one of the systems that have their growth journey as per the need of modern-day applications and are very dynamic. Kubernetes' dynamic nature challenges the historical ways of service discovery over the network. Kubernetes supports both client-side and server-side service discovery. However, service object is generally the most recommended way and is well supported with easy creation mechanisms. For example, **`SELECTOR`** constructs to identify endpoints automatically.

Kubernetes introduces some new concepts, especially when it comes to dynamic service discovery, and there is a maturity and learning curve involved. However, once you accomplish it, your application can dynamically find services and react to infra changes and other complexities of scale. When you start designing an application, think about the API logically and let Kubernetes handle the aspects of service, such as placement and discovery of components running on different network locations.

Points to remember

- Kubernetes is a modern and dynamic system that challenges traditional service discovery.
- There are two patterns of service discovery: client-side and server-side service discovery.
- Kubernetes supports both the patterns of service discovery but recommends using the service-side discovery mechanism of service objects.
- Kubernetes uses the **`SELECTOR`** construct to identify pods that constitute one service. The number of pods can scale up and down, based on current needs of number of pods to serve traffic without breaching SLAs.
- Service object provides a construct of readiness checks using which you can define when your pods should start getting traffic. This is very important as the same help in the graceful application shutdown.
- You can make manual service discovery in Kubernetes, but considering the dynamic nature, it is not the recommended way.
- When there is a need to connect to a service outside the Kubernetes cluster, which will be ported to Kubernetes, always declare an endpoint and map it with a service in your cluster.

Questions

1. Kubernetes has a mechanism to perform both client-side and server-side service discovery.
    a. True
    b. False
2. Can you do manual service discovery in Kubernetes? If yes, how?
3. In Kubernetes, can you create a service object without mentioning the **`SELECTOR`**? If yes, how will Kubernetes identify the backend pods?
4. How is a cluster IP address resolved to a pod location?
5. What is Kubernetes' recommended way to create a service? And why?

Answers

1. True
2. Yes, manual discovery is possible. Refer to the *Manual service discovery* section of the chapter.
3. Yes, you can create a service without mentioning the selectors. For more details, refer to the last section of the book, *Advanced details*, and refer to the section on endpoints.
4. This is done with help of Kubeproxy, which create IP tables writes to ensure the discovery of actual network locations when a cluster IP is triggered.
5. Create Kubernetes service objects with proper label selectors mentioned. This is the best approach because using this, all the dynamic nature of pod placements will be taken care of automatically, behind the scenes, without any manual intervention.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 8
Zero Trust Using Kubernetes

Introduction

Kubernetes is a platform that manages and automates the deployment and scaling of applications that run in containers, hosted either in the cloud or on-premises. Leveraging the benefits of such a virtualized infrastructure to deploy microservices comes with hidden security complexities. There are three vital security concerns when it comes to an orchestration system like Kubernetes: malicious threat actors, supply chain risks, and insider threats.

Supply chain risks creep in primarily at the time of development and packaging of applications and are hard to mitigate. For mitigation, you must go back and correct the issue in the life cycle of application development. Malicious threat components can exploit the vulnerabilities and insecure configurations in components of Kubernetes, such as API server, worker nodes and control planes. Insider threats are generally user entities with special access to Kubernetes infrastructure and the intention to abuse the privilege they have.

In this chapter, you will gain insights into the challenges of securing a Kubernetes cluster. It will include the strategies that are commonly used by developers and system administrators to mitigate the potential risks in container applications deployed on Kubernetes.

Structure

In this chapter, we will discuss the following topics:

- Kubernetes security challenges
- Role-based access control
    - Identity
    - Role and role bindings
    - Managing RBAC
    - Aggregating cluster roles
    - User groups for bindings
- Introduction to zero trust architecture
    - Recommendations for Kubernetes pod security
    - Recommendations for Kubernetes network security
    - Recommendations for authentication and authorization
    - Recommendations for auditing and threat detection
- Zero trust in Kubernetes
    - Identity-based service to service accesses and communication
    - Include secret and certificate management and hardened Kubernetes encryption
    - Enable observability with audits and logging

Objectives

In this chapter, you will learn how to manage and control the three common high levels risks: supply chain risks, malicious threat actors, and insider threats. This chapter will talk about common strategies to mitigate supply chain risks by talking about risks and mitigation at the code, build, deploy, and run stages of applications. It includes scanning containers statically and dynamically to address current and future risks, and making sure the application run on pods that adhere to the ideology of least privilege so that when a threat left unnoticed has an impact, the intensity is minimal. We will discuss strategies to reduce the malicious threat actors by talking about firewalls to limit access and encryption to ensure confidentiality. You will also develop a detailed understanding of role-based access to control the third category of threat: insider threats.

Kubernetes security challenges

Container applications have got a big boost in adoption with microservices architecture gaining popularity. While containerized applications do provide a lot of flexibility and benefits when compared to monolithic applications, they also bring in some major complexities when it comes to creating and managing applications. The processing capabilities not only make it a good attraction for stealing data, but the scalable nature makes cyberattackers seek computational power (often for running cryptocurrency mining). In addition to these threats, the obvious traditional threats like **Denial of Service** (**DoS**) remain equally applicable. The following threats represent some of the most common ways of compromise for a Kubernetes cluster:

- **Supply chain threat:** The list of factors contributing to the supply chain category of risk is many. Any element or software artifact used to create and deploy an application can become a reason for introducing this threat. For example, a third-party library used in the creation of the application has a vulnerability to make your system fragile and open to attacks. This kind of compromise can affect Kubernetes at multiple levels:
    - **Container level:** Applications running inside the containers often use third-party libraries, and a vulnerable library can make a complete setup open for attacks. A malicious application can act as a launchpad for attackers to affect the underlying infrastructure or any other application deployed in vicinity of non-secure applications.
    - **Container runtime:** In Kubernetes, each node has a container runtime that pulls the image from the registry and deploys it in the cluster. Its responsibility includes isolating the system resources for each container. A vulnerability in runtime can result in insufficient segregation between two containers and that could prove fatal.
    - **Kubernetes cluster infrastructure:** The packages, libraries, and third-party software installed as part of the control plane can also make the system vulnerable. The excessive runtime permissions for the app can multiply the effect of the supply chain issue introduced. For example, getting admin access by the code and amplifying the issue introduced by stage 1/2/3 mentioned earlier.
- **Malicious actor threat**: An actor of this category generally attacks the system by exploiting vulnerabilities and stolen credentials from social engineering to achieve access. In Kubernetes, there are multiple components and APIs which an attacker of this category can exploit.
    - **Control plane:** The control plane has many APIs that create/manage the whole cluster. If an attacker gets control of the control plane, an

attacker potentially has your entire Kubernetes cluster to play with. This is one of the common ways to attack the system, so it becomes important for components (deployments) inside Kubernetes to have their security mechanisms in place. For example, even if the attacker has access to the control plane, it should not be able to trigger a micro-service API. This becomes the responsibility of microservice and its related deployment components.

  - **Worker node:** Worker nodes have two key services running, i.e., kubectl and kube-proxy, that are of interest to a cyberattacker. The worker nodes are outside of the control plane and hence, are more open to attacks.

  - **Applications:** A vulnerable application inside the cluster can make the application visible outside the cluster. An attacker can take control of not only the vulnerable application, but can also escalate privileges in a cluster and eventually, make the whole cluster vulnerable using this.

- **Insider threat:** This category covers actors who exploit vulnerabilities and privileges given to them while they are supposedly working on project/organization. These individuals have special knowledge and access permissions that were not revoked completely after a certain time, and these lapses in revoking access when aggregated over time, makes system vulnerable.

  - **Administrators:** K8s admins have the power to run any arbitrary command on a cluster that includes executing commands inside the containers as well. Admins also have physical access to the machines and hypervisors, which could result in the entire infrastructure being compromised. Hence, these Admin roles are generally controlled by appropriate RBAC roles.

  - **Users:** A user of the containerized application may know the credentials to access a containerized service in a Kubernetes cluster. This level of capability can result in the exploitation of not only the applications deployed but might also affect other cluster components.

Now that we understand the details about various types of threats that can occur at a high level in the Kubernetes setup, it is time to define the term *zero trust* in more detail. However, to understand aspects of zero trust, it is vital to understand a key building block: **Role-Based Access Control** (**RBAC**) construct of Kubernetes; it is key in ensuring authentication and authorization to cluster.

Role-based access control (RBAC)

RBAC was introduced into Kubernetes with version 1.5 and became generally available in version 1.8. RBAC restricts both access to and actions on Kubernetes API to ensure that only appropriate users have access to APIs in the cluster. With help of RBAC, you can ensure that only authorized users can access resources where the application is deployed, and you can also prevent unexpected accidents where a person mistakenly destroys your application.

Every request sent to Kubernetes, like any other system, is first authenticated. Kubernetes does not have a built-in identity store; rather, it relies on a pluggable third-party system like Azure Active Directory. Once the users are authenticated, the authorization phase starts. Authorization is a combination of the identity of the user, the resource, and the action the user wants to perform. If the user is authorized to act, action is performed; else, the API returns an HTTP 403 response.

To understand RBAC, it is critical to understand the following concepts/terms related to RBAC.

Identity

Any request that comes to Kubernetes has an identity. This identity could be system generated or of an actual user. Kubernetes can create distinctions between system identity and user identity. A request without identity in the system belongs to the unauthenticated group. System identities representing service accounts are created and managed by Kubernetes and are linked with components running inside the cluster. On the other hand, user identities associated with user accounts are actual users, such as a user running integration tests outside the cluster and interacting with K8S platform. Kubernetes has a common interface for all authentication providers, with every provider providing a username and a set of groups to which the user belongs. Some common authentication providers are as follows:

- HTTP basic authentication (largely deprecated)
- x509 client certificates
- Static token files on the host
- Cloud authentication providers like Azure Active Directory and AWS Identity
- Identity Access Management (IAM)
- Authentication web hooks

Managed installation of Kubernetes (such as GKE on GCP and AKS on Azure), automatically configures authentication for you. In the case of self-deployed Kubernetes, the flag needs to be configured on the Kubernetes API server appropriately.

Role and role bindings

Roles are a set of capabilities; for example, a dev role could be defined to create and delete pods. These roles created can then be bound to multiple users defined as identities, known as **role bindings**. Following is an example of a role definition that has the capability to create and modify pods and services:

```
1. kind: Role
2. apiVersion: rbac.authorization.k8s.io/v1
3. metadata:
4.   namespace: default
5.   name: crud-pod-services-role
6. rules:
7. - apiGroups: [""]
8. resources: ["pods", "services"]
9. verbs: ["create", "delete", "get", "list", "patch", "update",
   "watch"]
```

In the preceding role definition, line 4 represents the namespace for the role. This metion of namespace results in this role being available in the boundaries of one namespace. If we want to use it in different namespace, similar role has to be created again with mention of another namespace.. Line 5 represents the name of the role. Line 7 signifies that all the APIs belonging to all API groups are covered in the role definition. Line 8 represents the resources on which this role is applicable, and line 9 represents the set of actions this role can perform.

As step two, you can bind this role to different users and user groups. Consider the following role binding definition to bind the preceding role named **`crud-pod-services-role`** to user **`jack`** and user group **`dev`**:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: RoleBinding
3. metadata:
4.   namespace: default
5.   name: pods-and-services-binding
6. subjects:
7.   - apiGroup: rbac.authorization.k8s.io
8.     kind: User
```

```
9.      name: jack
10.   - apiGroup: rbac.authorization.k8s.io
11.     kind: Group
12.     name: dev
13. roleRef:
14.   apiGroup: rbac.authorization.k8s.io
15.   kind: Role
16.   name: crud-pod-services-role
```

In the preceding role binding definition, role binding with name pods-and-services-binding (line 5) is created in namespace default (line 4). Subjects (line 6 to 12) provide the details of user **`jack`** (line 7 to 9) and user group **`dev`** (line 10 t0 12). The role definition uses the **`crud-pod-services-role`** role (line 13 to 16) created earlier.

In some cases, you might want to define a role that spans across your entire cluster. To do this, you can use **`ClusterRole`** and **`ClusterRoleBinding`**. The constructs are very similar to the role and role binding constructs used previously. Here's a **`ClusterRole`** sample:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: ClusterRole
3. metadata:
4.     name: crud-secrets-clusterole
5. rules:
6.   - apiGroups: [""]
7.     resources: ["secrets"]
8.     verbs: ["get", "watch", "list"]
```

Look at line number 2, which defines the kind as **`ClusterRole`**. The metadata section (lines 3 and 4) does not contain any reference to namespace. Lines 5 to 8 represent operations of get, watch and list secrets. Following is the **`ClusterRoleBinding`** definition using the previous **`ClusterRole`**:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: ClusterRoleBinding
3. metadata:
4.   name: crud-secrets-clusterolebinding
5. subjects:
```

```
6.    - kind: Group
7.      name: cluster-admin
8.      apiGroup: rbac.authorization.k8s.io
9.  roleRef:
10.   kind: ClusterRole
11.   name: crud-secrets-clusterole
12.   apiGroup: rbac.authorization.k8s.io
```

This cluster role binding allows anyone in the **`cluster-admin`** (line 7) group to read secrets in any namespace (defined via **`ClusterRole`** above and referenced in line 12).

You would have noticed that the verbs tag in each of the **`RoleBinding`** definitions, which defined the actions on the resources (such as pods and services in **`crud-pod-services-role`** and secrets in **`crud-secrets-clusterole`**), correspond to HTTP methods of APIs. The commonly used verbs in Kubernetes RBAC are listed in *Table 8.1*:

Verb	HTTP method	Description
Create	`POST`	Create a new resource
Delete	`DELETE`	Delete an existing resource
Get	`GET`	Get a resource
List	`GET`	List collection of resources
Patch	`PATCH`	Patch (partially modify) an existing resource
update	`PUT`	Modify the complete resource
Watch	`GET`	Watch for updates to resource
proxy	`GET`	Connect to resource via WebSocket Proxy

Table 8.1: *Common RBAC verbs*

Kubernetes has several built-in cluster roles. You can view them using the following command:

```
kubectl get clusterroles
```

While most roles that are outcomes of the preceding commands are used for system utilities, there are two key roles defined for actual users:

- **cluster-admin**: This role provides complete access to your cluster.
- **Admin**: This role provides complete access to all namespaces.

You can view the built-in **RoleBinding** as well, with the following command that is very similar to the preceding command for **ClusterRoles**:

```
kubectl get clusterrolebindings
```

When a Kubernetes API starts up, it automatically configures multiple default **ClusterRoles**. If you have made any changes to the cluster roles, those changes will wiped out when you restart the API server. To ensure that your changes remain as is after a restart and during time of reconciliation loops, you must set **rbac.authorization.kubernetes.io/autoupdate** to false in the built-in **ClusterRole** resource. Role **system:unauthenticated** is a built in **ClusterRole** that lets unauthenticated users discover the APIs of Kubernetes cluster.

Managing RBAC

Creating and managing RBAC configurations is complex and can introduce key security issues if not done correctly. To check what actions a person can take, you can use the **can-i** command.

The **can-i** command is of the following form:

```
kubectl auth can-i <action> <resource>
```

Kubernetes cluster admins use the impersonation feature to test what a user can do. This helps them configure and test right user permissions on the cluster. For example, the following command can be used to enquire if we can create a pod:

```
kubectl auth can-i create pods
```

In the real-world use cases, RBAC is created and managed by creating and managing either JSON or YAML file. These files are maintained in version-controlled repositories such as git. The need for auditing, accountability and rollback are prime factors toward maintaining artifacts in version control.

The **kubectl** command-line tools come with a **reconcile** command in addition to the **apply** command that reconciles a text-based set of roles and role bindings with the current state of the cluster.

Consider the role binding **pods-and-services-binding** that we created in the earlier sections of the chapter. It has a role reference mentioned as **crud-pod-services-role**. Now, let us say you want to update the same role binding with a different role reference, that is, **crud-pod-services-role-update**. For this, you will probably pull the file from git and make the necessary changes and use the **kubectl apply -f <filename>** command.

The step will fail with the following log:

```
The RoleBinding "crud-pod-services-role-update" is invalid: roleRef:
Invalid value: rbac.RoleRef{APIGroup:"rbac.authorization.k8s.io",
Kind:"Role", Name:" pods-and-services-binding"}: cannot change roleRef
```

This is because **`roleRef`** is immutable, so we cannot update this **`pods-and-services-binding RoleBinding`** using **`kubectl apply`**.

To tackle this situation, **`kubectl auth`** reconcile comes into the picture. If you use **`kubectl auth reconcile -f <filename>`**, it will delete and recreate the role binding. It is a good practice to use the **`--dry-run`** flag before executing the command and making cluster-level changes of RBAC.

Aggregating cluster roles

There are situations when you want a cluster role that is a combination of already created cluster roles. One way to counter such a situation is to create a third role with all the configurations copied from the two roles. Another option is aggregating **`ClusterRole`**.

The way to achieve this is very consistent with Kubernetes's way of grouping other objects, that is, label selectors. Following is a snippet of how you will attach labels to a custom **`ClusterRole`**:

```
1. apiVersion: rbac.authorization.k8s.io/v1
2. kind: ClusterRole
3. metadata:
4.   labels:
5.     rbac.authorization.k8s.io/allow-aggregate: "true"
6.   name: some-aggregation
7. rules:
8. (…)
```

Look at line number 5, where a label is attached to the **`ClusterRole some-aggregation`** (line 6). With the preceding **`ClusterRole`** in existence, you can define aggregation as follows:

```
1.  apiVersion: rbac.authorization.k8s.io/v1
2.  kind: ClusterRole
3.  metadata:
4.    annotations:
```

```
5.      rbac.authorization.kubernetes.io/autoupdate: "true"
6.    name: demo-clusterrole-aggregation
7.  aggregationRule:
8.    clusterRoleSelectors:
9.    - matchLabels:
10.       rbac.authorization.k8s.io/allow-aggregate: "true"
```

Pay attention to line 5; the auto-update is marked as true, meaning any change will be overridden when the API server restarts. Line numbers 7 to 10 set up the aggregated **`ClusterRole`** based on the labels earlier.

A general best practice is to create smaller granular cluster roles and then aggregate them for broadly defined roles.

User groups for bindings

In organizations, a large number of people can access a cluster. It can also happen that multiple teams are using the same cluster segregated via namespaces. You can add individual identities of users to each **`ClusterRole`** and user **`RoleBinding`**. Technically, it is fine to add all your users one by one into a **`RoleBinding`**, but this approach becomes cumbersome when you want to add a new user or delete a user from the **`RoleBinding`**. It is always a case that people come and leave, and rather than giving individual identities a mention in **`ClusterRole`** and **`RoleBinding`**, it is recommended to create User Groups and use them for Bindings. It makes management very simple and less error-prone.

These user groups are provided by the authentication providers; groups defined in authentication providers are not just for Kubernetes but for other scenarios as well. Assume that we have a group **`devops`**; this **`devops`** group could have access to run all pipelines and act as cluster admins. Hence, the same group could be used to give **`ClusterRole`**.

You have already seen how this is accomplished when you created **`RoleBinding pods-and-services-binding`**, a snippet of which is as follows:

```
1. (…)
2. subjects:
3.   - apiGroup: rbac.authorization.k8s.io
4.     kind: User
5.     name: jack
6.   - apiGroup: rbac.authorization.k8s.io
```

```
7.      kind: Group
8.      name: dev
9. (…)
```

Lines 6 to 8 use a group 'dev' provided by authentication providers.

Introduction to Zero Trust Architecture

Zero Trust Architecture (**ZTA**) is a security framework that takes care of all the challenges described in the earlier section of *Kubernetes security challenges.* It enforces each user entity, inside or outside an enterprise network, to be authenticated, authorized, and evaluated/validated against security rules/policies based on its current role. ZTA not only talks about securing infrastructure but also the data (in transit and at rest) in the system. The key aspects that need to be investigated/evaluated from the implementation point of view with this framework are as follows:

- Defining user identities and grouping and allocating the right credentials to them
- Identifying the area to create privileged accounts and allocating those accounts to the right user groups
- Studying and understanding the behavioral patterns of how different components in the system interact/communicate with each other, and defining policies governing those behaviors
- Using authentication protocols and mitigating associated risks
- Identifying libraries/artefacts that can bring in vulnerabilities, and being able to apply security patches across the system, as and when needed
- Managing and securing applications installed on endpoints
- Evaluating the validity of the defined controls in real time

And with all the mentioned aspects in place, it is also important to identify the security or incidents involving suspicious activities and recognize and take action

This discussion is formally captured by the **National Institute of Standards and Technology** (**NIST**), which defined the way ZTA should work and published it as a standard SP 800-207 (**https://csrc.nist.gov/publications/detail/sp/800-207/final**).

In this document, *Section 3.1* describes the three requirements of ZTA:

- The first one is designed using the network infrastructure and software-defined parameters applied to Kubernetes-based applications. According to the document, *When the approach is implemented at the application network layer*

(that is, layer 7), the most common deployment model is the agent/gateway (Section 3.2.1). In this implementation, the agent and resource gateway (acting as the single PEP and configured by the PA) establish a secure channel used for communication between the client and resource.

In Kubernetes, this requirement is handled when Ingress controllers and service mesh manage authentication. These components operate as gateways, keeping complete authentication away from Kubernetes APIs and applications that run inside containers.

- The second requirement is covered in *Section 3.4.1*, which is concerned with network requirements for ZTA. An application needs to acquire traffic from an external network, apply authentication and authorization to the external traffic, and hand over the requests to Kubernetes APIs.
- The third requirement describes the authentication and authorization model to determine the identity of users and machines, and their responsibilities for what they are allowed to do. Integrity ensures that data and configuration settings are managed and modified by authorized entities.

Table 8.2 describes each of the requirements for how it can be implemented inside Kubernetes:

Requirements from NIST SP 800-207	Implementation In Kubernetes
The enterprise can observe all traffic.	This is accomplished in Kubernetes by maintaining an observable state based on logging, metrics, and tracing.
Enterprise resources should not be reachable without accessing a PEP (policy enforcement point).	Reachability from external systems is managed via Ingress and service mesh configurations. Reachability within Kubernetes is managed by Kubernetes constructs such as Network policies, which impose control and restrictions between communications happening between two pods.
The data plane and control plane are logically separate.	This requirement is managed by Kubernetes internal design. The Kubernetes control plane includes the API server, scheduler, and controller managers. The data plane consists of nodes and their containers.
Enterprise assets can reach the PEP component. Enterprise subjects must be able to access the PEP component to gain access to resources.	This is managed by Ingress controllers and service meshes.

Requirements from NIST SP 800-207	Implementation In Kubernetes
The PEP is the only component that accesses the policy administrator as part of a business flow.	To ensure this, it is essential for ingress controller and service mesh to manage authentication and authorization rather than nodes, containers, or any other Kubernetes components.
Remote enterprise assets should be able to access enterprise resources without needing to traverse enterprise network infrastructure first.	This is managed and taken care of by well-configured ingress.
The infrastructure used to support the ZTA access decision process should be made scalable to account for changes in process load.	Ingress and Service mesh deployed in production systems should support this.
Enterprise assets may not be able to reach certain PEPs due to policy or observable factors.	Ingress and Service mesh, coupled with efficient observability, ensure this requirement.

Table 8.2: *Mapping NIST SP 800-207 to Kubernetes*

With this, let us dive into the details of each of the types of threats described previously and see why they are so critical.

When the described aspects are cast completely into the Kubernetes world, it boils down to the following list of security recommendations for your Kubernetes components. Note that attackers keep innovating their strategy and the type of attacks they can come up with. So, the following list is by no means the complete list of recommendations. There could be new situations based on the type of project you are working on.

Recommendations for Kubernetes Pod security

For ensuring the security of pods and ensuring that the attack surface is minimum, one should think of the following aspects while building and running containers inside pods:

- **Use containers built to run applications as non-root users:** Many times, the docker users either forget or do not see the benefit to change the user privileges inside the launched Docker image instance and keep performing the code execution as root users instead of non-root. This bad practice always poses a threat when the application is deployed and made public. This results in a threat not only to the application but also to the underlying file system of the docker container, resulting in weird behavior for other applications running inside the container.

```
1. FROM ubuntu:latest
2. RUN apt update && apt install -y make
3. COPY . /code
4. RUN make /code
5. RUN useradd devuser && groupadd devgroup
6. USER devuser:devgroup
7. CMD /code/app
```

In the preceding oversimplified docker file, refer to line numbers 5 and 6, which create a user and user group (line 5), and run the application as newly created users (line 6).

- **Run containers with immutable file systems:** Immutable implies that a container will not be modified but will be updated and redeployed. A container modification could be updates, patches, and configuration changes. Immutability enables deployments to be fast and secure. In Kubernetes, you can use both ConfigMaps and Secrets to inject configurations inside containers as environment variables or files. If you need to update a configuration, you should ideally deploy a new container (based on the same image) with the updated configuration.

  The following example is a Kubernetes deployment template that uses a read-only root file system:

```
1. apiVersion: apps/v1
2. kind: Deployment
3. (...)
4. spec:
5.   (...)
6.   template:
7.     (...)
8.     spec:
9.       containers:
10.       (...)
11.        securityContext:
12.           readOnlyRootFilesystem: true
13.        volumeMounts:
```

```
14.        - mountPath: /var/directory
15.          name: immutableVolName
16.      volumes:
17.        - emptyDir: {}
18.           name: immutableVolName
```

Line numbers 11-12 showcase how to define read only root file systems. Lines 13-18 are the volume treated as root file system.

- **Static and dynamic scan of images for vulnerabilities:** Static scans help identify the vulnerabilities in your software artifacts while building them. However, it is generally seen that a new vulnerability is observed in the libraries over a period; hence, it becomes important to run a dynamic scan as well. A static scan is performed by analyzing and examining the source code or compiled code for vulnerabilities, which tests the internal structure of the application (Whiteboxeb). Dynamic scans analyze applications from outside, that is, by running and manipulating applications from outside to discover vulnerabilities (Blackbox).

- **Technical control to enforce security**: In this category, you can apply technical governance to not create vulnerable/insecure conditions for deployments. The key ones are as follows:

    - Do not create privileged containers. Privileged containers are those that have all root capabilities on a host machine. When a cyberattacker gains control over such containers, the potential attack surface is very wide.

    - Restrict features in the container, such as **`hostPID`**, **`hostIPC`**, **`hostNetwork`**, and **`allowedHostPath`**, which are frequently exploited to intrude system and gain access.

    - Ensure hardening the applications by using security tools like SELinux, AppArmour and seccomp. **SELinux** is a set of alterations in the kernel of many Linux distributions.

    - Isolate critical workloads to support multi-tenancy, which refers to multiple suites of applications running on the same cluster. For critical suites, you can segregate the boundaries of execution using namespaces, RBAC, quota, and limits.

A different use case can have more such technical controls defined as per the need for compliance and security objectives. Kubernetes offers a component named pod security admission controller to create and manage the security standards for pods. Pod security restrictions are applied at the namespace level at the time of creation.

You need to first create a pod security standard and then assign it to a namespace. The same pod security standard could be used for multiple or all namespaces. For more details, visit **https://kubernetes.io/docs/tasks/configure-pod-container/enforce-standards-namespace-labels/**.

It is the responsibility of administrators to create custom policies to meet the requirement of their organization.

Recommendations for Kubernetes network security

The following recommendations and actions help teams to harden their network on Kubernetes deployment and ensure that unwanted access is not allowed to the control plane of Kubernetes:

- Restrict access to control plane nodes by using a firewall, and in case access is needed (for admins), it must be done via **role-based access control** (**RBAC**). It is a best practice to use a separate network for the control plane and data plane. Use private subnets to deploy control plane and worker nodes. Use the public subnets to deploy internet-facing load balancers. In terms of cloud deployments, you can have one virtual private cloud for hosting the control plane and other virtual private clouds for hosting the data plane.
- In addition, it is recommended to control plane components to use authenticated and encrypted communication via **Transport Layer Security** (**TLS**) certificates.
- Enforce limited access to the **`etcd`** server. Rather, use Kubernetes API to access details from **`etcd`**. Moreover, encrypt the data at rest and a separate TLS certificate for communication.
- Create and enforce network policies. It is recommended to create an explicit deny network policy. Moreover, set up network policies to isolate resources. By default, pods and services can communicate with each other, and if there is a need to restrict it, it can be done by applying network policies.
- Network policies are applied on namespace and hence, all pods inside the namespace adhere to the network policy. Policies can include one or multiple ingress rules or egress rules. To tie a pod in a namespace to the network policy, use a pod selector. Within ingress and egress rules, we configure different pod selectors to apply the rule. Consider the following ingress policy:

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

```
4.      name: allow-in-same-namespace
5.      namespace: default
6.    spec:
7.      podSelector:
8.        matchLabels:
9.          app: database
10.     ingress:
11.       - from:
12.           - podSelector:
13.               matchLabels:
14.                 app: orders
15.         ports:
16.           - port: 80
```

In the preceding YAML network policy configuration, all the incoming traffic to application database is allowed from pod with label order (line 14) on port 80 (line 16).

Refer to the **`deny-all-ingress-egress.yaml`** file in the codebase. This network policy, when applied, will deny all ingress and egress traffic to pods.

- Use Kubernetes secrets instead of configuration to secure all the sensitive information of your application. You can configure strong encryption around your secrets to prevent unwanted exposure and use.
- Restrict public access for worker nodes. Refrain from exposing services via node port; rather, expose them through load balancer or ingress controllers.

Recommendations for authentication and authorization

Following are some recommendations for allowing a user or a group of users with responsibilities to perform actions on the Kubernetes cluster:

- Disable anonymous login (which is enabled by default).
- Enforce strong user authentication mechanisms, especially for non-Kubernetes-managed users.
- Create and enforce RBAC policies for roles such as user, administrator, developer, infrastructure team, and other similar groups.

Recommendations for auditing and threat detection

Following are a few recommendations related to audit logging and threat detection:

- Enable audit logging; Kubernetes has an auditing feature that provides a timestamp-based collection of records for the sequence of actions performed in a cluster. For more details, visit **https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/**.
- Ensure that logs are persisted in case of node, pod, or container failures. This will help ensure tracking of cyberattacks in failed components.
- Enable logging throughout the environment. Cluster API audit event logs, metric logs, application logs, Pod seccomp logs, and repository audit logs are a few common ones among many.
- Not just enabling, aggregation of all these logs to an external location is also recommended. In case any component fails and a new one is created, the logs should remain available to detect threats.
- Enabling alerting and monitoring on the aggregated logs helps detect not just the dominant and obvious but also hidden and silent ones.

Recommendation for application security practices

Some recommendations for application security practices are as follows:

- Make sure no library/component that is not needed is installed in your system.
- For the library/components installed, ensure that all the recommended security patches are applied regularly.
- Perform penetration testing and dynamic scanning of all artifacts periodically.

To ensure that all the preceding recommendations are well implemented, action needs to be taken at multiple levels. Actions like the creation of multiple networks/VPC are part of infrastructure creation. Similarly, to implement recommendations inside a pod or container, one must develop the application code adhering to best practices. For example, to not run an application as root, the Docker file must be written in a way. The third area where the controls are applied is via Kubernetes constructs. In the rest of the chapter, you will develop an in-depth understanding of Kubernetes constructs to enable Zero trust.

Zero trust in Kubernetes

In this section, you will get an understanding of some key needs of zero security when it comes to applications and services interacting with each other. Enabling secure infrastructure, data, and access is becoming a complex topic, in the age where enterprises are moving toward multi-cloud and hybrid infrastructure. With these new-age shifts, the traditional way of security is also changing rapidly. Zero trust architecture is key when it comes to modern applications running across hybrid clouds, different environments, and so on.Notion is to not even trust their own services before allowing access. The following three best practices principles of zero trust will help you secure an application in today's complex deployment models and strategies.

Identity-based service to service accesses and communication

Modern services should be built around service identity. Service identity should be used for authorization rather than IP identities or any other. Two services willing to interact must mutually authenticate their identities and apply authorization to let the connection be established.

Kubernetes, by default, allows all containers and services to interact with each other and assumes that the container network and applications running on the cluster are trusted. For instance, a logging service and a domain service running on the same Kubernetes cluster can access each other at the network level. You can create policies in Kubernetes to apply default rules denying ingress and egress to the cluster. However, there is still a need to establish service-to-service authentication and authorization to ensure only secure working.

The preceding service-to-service communication is established with the help of service mesh. Service mesh allows you to create service identities for every service running on the cluster, and based on the service identity, mesh authenticates service identities using **`mTLS`**. You will see a complete section talking about Istio later in this book, which will cover the way traffic is managed across services, in *Chapter 12, Traffic Management Using Istio*. Istio uses a side car pattern, that is, in a pod apart from the application container, a sidecar Envoy Proxy container is also running. Take a look at *Figure 8.1* to understand the basic service-to-service communication:

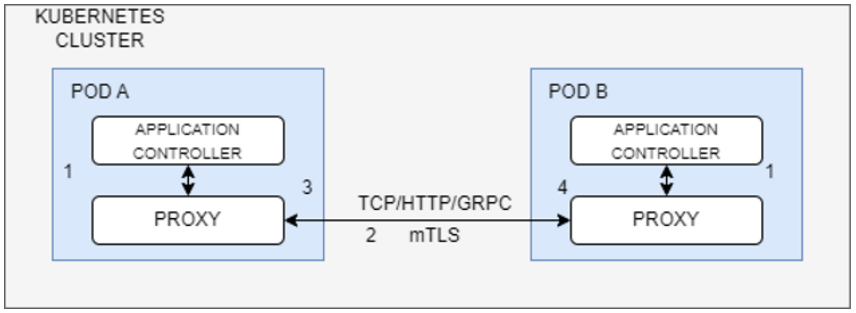


Figure 8.1: *Istio mTLS*

Refer to the numerical labeling in *Figure 8.1* with the corresponding numerical explanations:

1. Sending and receiving the network traffic across services via proxy.
2. When mTLS is enabled between the client-side (assume **POD A**) service and server-side (assume **POD B**) service, proxies on each side verify each other's identities claims before sending requests.
3. On successful verification, the client-side proxy (**POD A**), encrypts the traffic and sends it to the server-side proxy (**POD B**).
4. The server-side proxy decrypts the traffic and forwards it to the destination service.

The service-based calls can be authorized and blocked using intentions, which allow service-to-service communication permission via service names.

Include secret and certificate management and hardened Kubernetes encryption

Secrets used by applications deployed in Kubernetes should be encrypted and time-bound (have an expiry date) and should have the ability to work with a global service identity that facilitates encryption of data in transit.

Kubernetes secrets (*Chapter 5, ConfigMap, and Labels*) do not comply with the needs of Zero Trust Architecture. By default, the secrets are just base64 encoded and not encrypted. Secrets do not expire and hence, once compromised, they can lead the application to risks, till such scenarios are identified and fixed. Last but not least, Kubernetes manages these secrets within the cluster. If you want a set of secrets to be managed across clusters, they must be managed separately.

To encrypt secret data at rest, the following yaml configuration file provides a simplified example to specify encryption type and the encryption key. The secrets will be encrypted, but the key will be accessible in the **`EncryptionConfiguration`** file. Below YAML snippet shows how to use a custom encryption key for encrypting secrets:

```
1. apiVersion: apiserver.config.k8s.io/v1
2. kind: EncryptionConfiguration
3. resources:
4. - resources:
5.   - secrets
6.   providers:
7.   - aescbc:
8.       keys:
9.       - name: key1
10.         secret: <base 64 encoded secret>
11.  - identity: {}
```

Save this file to a location that will be used to enable encryption at rest by restarting the API server with the **`--encryption-provider-config`** flag set with the location of the configuration file.

Generally, a service mesh with a secret broker can easily handle this challenge. For example, Istio has integration with Hashicorp's Vault – a centralized secret management system. A secret is added manually or by system process to the vault. A service account is created and is added to vault and the pods/services accessing the vault. Vault fetches the credentials and mounts them automatically in the pod.

This setup helps ensure that secrets are encrypted via a centralized access control and auditing-capable system. Workflows are expected to support single and hybrid multi-cluster deployment. In addition to this, features like time to live and rotation of secrets without impact on workflows improve the trust in the system.

Enable observability with audits and logging

To understand and improve security, it is vital to understand what requests have been made to services deployed in the Kubernetes cluster. Audit logs help inspect which credentials have been used, how they were used (for performing which action), and when (the timestamp) they were used. This provides insights and accountability for security teams.

The challenge is that the applications (running inside pods) are generating logs inside the container. Since there could be multiple containers running, the logs are distributed across multiple containers/machines. Kubernetes has no native way to aggregate these logs at the cluster level. However, there are a few proven ways to accomplish cluster-wide collection of logs to a central location. Widely used methods are as follows:

- **Configure an agent on every Kubernetes node:** Isn this approach, an agent is installed on every node as Daemon set in Kubernetes, which pulls the data from within the container to a central location. The advantage of this approach is that it needs no changes to the Kubernetes cluster, but on the other side, the daemon set needs elevated privileges to access the log files.
- **Include a sidecar that gets added to every Kubernetes pod:** A sidecar container is a container that runs with the main container in the same pod. There are two industry-standard ways of implementing sidecars for logs: the first one is that the sidecar diverts the logging traffic to a central repository, and the second is where the sidecar diverts all the log traffic to a custom **`stdout`** file and then it is shipped off to a central location.
- **Configure every containerized application individually to ship its logs:** The third strategy is to configure the application itself to write the logs to a central location. While the advantage is that you do not need to maintain any other daemon or sidecar components, with a pain point that you must write and maintain the logic to push logs to the central location.

When using a service mesh, generally the sidecar method is used to aggregate logs. Service meshes have well-defined integration with the most common, capable, and efficient monitoring tools like Prometheus and Grafana to make it easier to review, understand and analyze networking patterns inside services and improve security.

Conclusion

Kubernetes is capable enough to support all needs and restrictions suggested in Zero Trust Architecture. K8S APIs have RBAC and TLS support for efficient authentication and authorization. Kubernetes has documentation to implement various aspects of ZTA. However, none of them are enabled by default; an admin must enable them explicitly. Kubernetes can be very secure, but it is not secure out of the box, and you explicitly must modify config files to support security features.

Service mesh is an important construct that helps enable some key features of security mostly with a side care design pattern. It generally introduces a sidecar container to perform all administrative tasks, while the main container runs as is. This chapter describes the ZTA features based on the NSIT document. However, in practical scenarios, there could be more aspects that need consideration, so it is

recommended to review and audit the security setting periodically to prepare for the worst.

Points to remember

- Zero trust is all about enabling data transmissions and interactions with authentication at all possible levels in your entire cluster. To deploy **Zero Trust Architecture** (**ZTA**) for Kubernetes, we strategize to effectively authenticate and authorize entities making calls to API servers or applications.
- ZTA requires encrypted and monitored access to users (system and real) to prevent all the data from unauthorized tampering and to preserve the data integrity aspects of the system.
- ZTA requires setting up an observable state for all the events and metrics in your application and making sure states are visible to security controls and other user entities who manage them.
- ZTA requires minimizing the size of the resources your namespaces, clusters, and users can access in your Kubernetes cluster.
- Effective ZTA must also ensure robust access monitoring systems. Monitoring how access control runs in your cluster is as important as authenticating requests.
- ZTA enforces all access in your Kubernetes-based network to be encrypted.
- By default, Kubernetes constructs need to be switched on constructs needed for ZTA support.

Questions

1. What activities are involved in ensuring that Kubernetes hosts are secure as per ZTA guidelines?
2. What are a few key data security activities done on Kubernetes cluster as per ZTA guidelines?
3. It is recommended to have either static scan or dynamic scan, but both is overkill. Is this true or false?
4. What is the key use case to deploy critical applications in a different namespace from the ZTA perspective?

Answers

1. The number of activities depend on the use case. However, some must-do activities are listed here: ensuring updated OS with the latest patches, implementing firewall rules and other security measure related to environment security.
2. Key activities involve encrypting data at rest and ensuring that RBAC aspects are followed to ensure authorized delivery on untampered data.
3. False. Static scans and dynamic scans serve different purpose, so both should be enabled.
4. Key aspect is a different network policy that could be applied to enable strict governance to compliances.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 9

Monitoring, Logging and Observability

Introduction

Till now, we discussed and learned how a microservice can be deployed on Kubernetes. We learned how multiple Kubernetes objects combine to support effective hosting and deployment of business workflows. This tectonic shift in the way of hosting applications brought in changes as to how an application must be monitored, logged, and observed when compared to traditional ways. Adopting the microservices architecture needs the implementation of new observability practices to efficiently monitor the deployment setup. The old parameters capturing the health of a system have changed, and so has the way to interpret them. This changed interpretation has impacted how contracts are signed and managed between the application owners and end users. To ensure that the promises in the contract are kept, the site reliability engineer is expected to perform a set of duties.

While you can do a lot of things on one side, such as have hundreds of metrics created, hundreds of dashboards to monitor them, and multiple alarms/alerts on threshold breach of metric values, the key is to understand the right things to be done to accomplish effective observability practices, in order to save the application from critical downtime. This chapter will not only uncover the technology tools at hand to accomplish observability, but it will also take you through the process of managing observability in modern-day highly dynamic workloads.

Structure

In this chapter, we will discuss the following topics:

- Kubernetes observability deep dive
    - Selecting metrics for SLIs
    - Setting SLO
    - Tracking error budgets
    - Creating alerts
    - Probes and uptime checks
- Pillars of Kubernetes observability
- Challenges in Kubernetes observability
- Exploring metrics using Prometheus and Grafana
    - Installing Prometheus and Grafana
    - Pushing custom metrics to Prometheus
    - Creating dashboard on the metrics using Grafana
- Logging and tracing
- Defining a typical SRE process
- Responsibilities of SRE
    - Incident management
    - Playbook maintenance
    - Drills
- Selecting monitoring, metrics and visualization tools

Objectives

After studying this chapter, you will learn and understand the basics of Observability in Kubernetes using open-sourced and famous tools like Prometheus and Grafana, and Site Reliability Engineering in general. You will learn in detail how to build and host the SRE automation stack on the Kubernetes orchestration engine. You will understand the *how* of aspects like selecting metrics, setting objectives, tracking, alerting, and so on. In the last section, we will discuss the responsibilities of SRE teams in detail.

The chapter does not intend to make you an expert in SRE practices. Rather, it will make you an expert in handling the infrastructure part of SRE. The right value allocation to metrics, the frequencies of the run, and other aspects defining the reliability of the system are out of the scope of this chapter. We will, however, study the aspects of scaling the stack and how the tech stack behaves when the applications scale.

Kubernetes observability deep dive

The observability of any software system, in its simplest form, is the potential to measure the current state of the system. This state is generally measured by checking the data generated, which includes data getting generated via logs, metrics, and traces. The workaround observability is not new; in traditional systems, teams observed the monolithic systems. However, these monolithic systems, being restrictive in deployment footprints, made life easy. With a distributed system like Kubernetes coupled with Microservice architecture, the application got broken into multiple pieces (*microservices*) deployed over multiple nodes (nodes in the cluster), which opened a wide gap between traditional versus new ways of observing systems. This helped observe small pieces of applications (microservices) and aggregating results versus observing one large monolithic application and the infrastructure related observability. In this section of the book, we will discuss aspects to be taken care of for effective monitoring of your deployments on Kubernetes.

Before deep diving into the technical aspects of why and how of observability, let us quickly understand the process around observability. We will be having an elaborate discussion around the roles and responsibilities of the **Site Reliability Engineering** (**SRE**) team. However, for now, let us define the role at a high level to understand a few terms. A SRE team does an automated evaluation of the reliability, by bringing in a change in the system/services. With this basic definition, let us learn about the three most common terminology used in SRE worlds:

- **Service level objectives** (**SLO**): Availability is key to successful system/services. A system that is not available cannot do what it is intended to, and by default, will fail. Availability in terms of SRE refers to the service/system performing the intended tasks. We define a numerical percentage, which is the threshold target for availability. Under no circumstances can the service be below this amount of availability. This threshold is known as SLO. All non-functional requirements are defined with the intent of at least improving the SLO for the service. It might look like it is best to keep the SLO at 100%, so we all aim for that. However, there is a generally very high cost associated with this. Not just the cost, even the technical complexity increases exponentially as we inch closer to 100%. The rule of thumb is to define the lowest possible value that you can get away with for each service. Many teams implement periodic downtime to prevent the service from being overly available. This

downtime of services can be beneficial in identifying the inappropriate use of services by other services. The value of this metric is usually decided in collaboration with product owners and with the engineering team in advance. This results in lesser confusion and conflicts in expectations in the future.

- **Service level indicators** (**SLI**): We need to ensure that we understand availability (availability means functionally available) and have clear numerical indicators for defining availability. How we do that is not just by defining service-level objectives but SLIs. SLIs are more often metrics over time. Example metrics could be to request latency, batch throughput in case of batch systems, and failures per request. These metrics are aggregated over time, and we typically apply a function like a percentile or median. That is how we learn whether a single number is good or bad. An example of a good SLI could be the 99th percent latency, when the latency of request over the last 5 minutes is 300 milliseconds. Another example could be that the ratio of error requests to the total number of requests over the last 5 minutes is less than 1%. These numbers aggregated over a longer time, let's say a year, will tell for how long the application was down. If the total downtime comes out to be less than 9 hours over a year, it corresponds to 99.9 percent availability.
- **Service level agreement** (**SLA**): This is a commitment between the service and the customer about how reliable the service/system is going to be for end users. It is promised that the availability SLO will not go below this number, or else there will be a penalty of some kind. Because of the penalties involved, the availability SLO in SLA is a looser objective in comparison to internal SLO. The teams should feel 100% confident in maintaining this availability. You may have an agreement with a client that the service will be available 99% of the time each month. You can configure an internal SLO to be 99.9% and configure alerts when this number is breached. The calculation is as follows:
    - **SLA with clients**: 99% availability; 7.31 hours of acceptable downtime per month
    - **SLO**: 99.9% availability; 43.83 minutes of acceptable downtime per month
    - **Safety buffer**: 6.58 hours

No time could be enough in case the service faces a major disruption. However, the bucket of 6 hours and more between the internal and external objective provides peace of mind when you deploy.

An **error budget** is the maximum amount of time that a technical system can fail without contractual consequences. In the preceding example, the time of 43.83

minutes, which is the acceptable downtime per month, is also known as the error budget.

Selecting metrics for SLIs

Though the number and type of metric depends on your use case, as there is no limit to how many metrics/SLI there can be, there are three major categories of Service Level Indicators that are most widely used in the industry. These are, availability, latency, and correctness, and they are as follows:

- **Availability:** An extremely important SLI is availability, which is the fraction of time the service was unusable. It can also be described as the ratio of the number of successful responses to the total number of responses. Although various enterprises aim for high value for this SLI, achieving 100% is near impossible, and it is expressed in terms of "nines" in availability percentage. For example, the availability of 99.9% and 99.99% can be referred to as three and four nines.
- **Latency:** Most use cases consider request latencies, that is, how long it takes to return with a response as key SLI. Sometimes client-side latency is more often the metric that impacts the end user. Nonetheless, sometimes it is only possible to measure latency, as the non-measurable part is out of the boundary of SLI calculation. For example, network slowness or API gateway latencies.
- **Correctness:** Correctness is the measure of how correctly your application is behaving within a given time. You must trigger a synthetic workload at a periodic interval and check the outcome of the processing of the workload with the configured result expectations.
- **Error rate:** Error rate is defined as the ratio of requests that errored out to the total number of requests.
- **Data freshness:** In case of an update of data to a system, how quickly is the update available across all queries? For example, if a customer does a hotel booking and opts for a room, how quickly is this information available to others so that they do not book the same room again?

There are a few more SLIs, such as throughput and durability, which are self-explanatory. Although we can measure many such metrics, it would be best if we do not have more than a handful of SLIs to measure. It is best to stick to 4 or 5 SLIs that directly relate to customer satisfaction. Good SLI ties with user experience, that is, a low SLI will represent low customer satisfaction and a high value will represent more satisfied customers. If an SLI fails to achieve that, there is no point in capturing and measuring that SLI.

Setting SLO

SLOs are the targeted SLI indicator values that the engineering team intends to achieve. The time window due to the difference in the numbers of SLA and SLO depends on the confidence of the engineering team to resolve the error. For example, if the SLA is 99% (7.31 hours of acceptable downtime per month), SLO can be of a value like 99.9% (43.83 minutes downtime per month), which leaves a safety buffer of 6.58 hours. This 6.58 hours of downtime depends on the engineering team's inputs, and if they need more, the SLO (99.9%) could not be increased further, which leaves us with lowering the value of SLA.

The given steps follow the complete SLO creation process:

1. Identify the critical user journeys and arrange them in order of impact.
2. Identify the metrics that could be used as SLOs to accurately gauge the customer satisfaction.
3. Identify the SLO target goals and SLO measurement period.
4. Create the SLI, SLO and Error budgets on consoles.
5. Create alerts.

Point numbers 1, 2, and 3 are points where the whole engineering team collaborates with the Business Team to identify the numbers. In this section, you will investigate how you will perform step 4 on the examples that we had already seen in the current chapter (refer to previous sections of this chapter). For this, we will use the log-based metrics example. Following are the steps to configure an SLI and then an SLO.

Tracking error budgets

In a software system, there are several metrics that can be tracked, including infrastructure monitoring and service monitoring. The art of tracking lies in the fact of identifying the key SLIs and setting appropriate SLOs, that is, tracking the metrics that really affect end-user journeys. For example, tracking CPU usage as an SLI might not be a good idea, but tracking them for the internal infrastructure team to tackle the need for additional CPU might be crucial for debugging purposes. Tracking too many metrics will result in a noisy system, where there are high chances that the risky problems may get missed simply because of the number of metrics you are tracking.

There are multiple ways to track, such as creating dashboards for metrics, creating automatic alerting, and so on. In the case of SRE, you are working on metrics that affect customer satisfaction. Thus, any metric which can affect that becomes important, and we want to have sufficient bandwidth allowed by the system for us to work upon before an issue starts impacting the customers.

The SLI, SLO, and SLA combination results in the definition of error budgets. It is never ideal to utilize our entire error budget. This means if the error budget allows the service to be down for 7 hours (99.9% SLI and 99% SLA) in a month, it makes sense to distribute these 7 hours across the month, that is, 7/30 = .23 hours of the service being down every day. If the service is burning the error budgets at more than .23 hours a day, there are high chances of missing the SLA.

This rate becomes an important dimension to track and generate alerts if the rate increases. It is of prime importance to investigate the failures happening as early as possible so that the SLAs are not breached. There might be a situation when the resolution might need multiple teams to react, so it is key to start working on these deviations early. As an SRE, the following aspects should be clearly defined for each SLA committed to the customer:

- A very well-created incident response plan and rehearsals to recover from it. For example, if a Blob store location going down results in your service going down, it is important to have a well-defined plan for how this will be tackled. Not only a plan, it is equally vital to perform that exercise at regular intervals for the team to feel confident in the plan. These regular activities are called **SRE drills**.
- It is important to trigger alerts at the right time. For example, there is no point in alerting when the budget is exhausted. Rather, alerts at 25%, 50%, 75%, and 90% could help the teams plan the resolution well and smoothly.
- Try to automate the actions as much as possible. For example, if the service is behaving slow in certain regions, create a few instances in the backup region to quicken the process.

Rather than tracking everything, tracking the service level indicators that determine meeting SLO and hence, SLA, is the key to success.

Creating alerts

You can generate alerts based on the exhaustion of your error budget on SLOs. In this section, you will see how to generate alerts based on different percentages (25%, 50%, and 75%) of exhaustion of error budgets.

While configuring the alerts, you need to set a **lookback duration**, which is a time window for which we are going to track the error budget. Smaller lookback durations (fast burn scenario) result in faster detection of issues, but with a caveat that the error rate over the course of the day may result in over-alerting by the system. A longer duration (slow burn), if not alerted, may result in exhaustion of the error budget before the end of the compliance period.

The second parameter is the **Burn Rate threshold**, which is the percentage of the error budget burned. If the burn rate exceeds the time window of lookback duration, an incident is generated. A good starting point for a fast-burn threshold policy is 10x the baseline with a short (1 or 2 hour) lookback period. A good starting point for a slow-burn threshold is 2x the baseline with a 24-hour lookback period.

An example configuration could be a situation where the lookback duration is 24 hours (that is, 1440 minutes) and burn rate threshold is 3.33. That is, in 24 hours, if the burn rate is above 3.33 percent, you burnt more than what was expected in a 30-day month. This alert makes sense, since if this continues, the chance of breaching SLA is high. Along with the preceding mandatory parameters, you can set the optional parameter of the alert notification channel, that is, how to inform the concerned team about the incident.

You can also go ahead and optionally mention the steps that could be taken when this kind of an incident occurs. This could be as simple as just informing a team XYZ about the incident to instructing the SRE to take some technical action. One good practice is to configure different levels of alerting mechanism. For example, the preceding daily alert on per day basis (fast burn) could be suited for the engineering team, but it will be too much for management. Hence, you should ideally set up an alert over larger lookback (slow burn) duration.

Probes and uptime checks

Testing specifies the acceptable behavior of an application with known data, while monitoring specifies the acceptable behavior in the condition of unknown user data. It might seem that the major risks of both known and unknown are covered by testing and monitoring defined for the system, but unfortunately, the risk is more complicated. The known bad request should error out, and known good requests should work. Implementing both as part of integration tests is a good idea, as you can run the same tests again and again at the time of each release.

However, it makes sense to think about setting up such checks as monitoring probes. It might seem that it is over-engineering to do so and therefore, pointless to deploy such monitoring because the exact check has been applied as integration tests. However, there are good reasons to think like that, and they are as follows:

1. The release test most probably has wrapped the server with the front end and fake back end.
2. The tests wrapped the release binary with a load balancing front end and separate scalable back end.
3. Front end and back end have different release cycles, and there are high chances of these cycles being different.

Therefore, monitoring probes is a configuration that was not tested. Probes should ideally never fail, but if they do, it means that either the back end or front end is not consistent in release and production environments. One can argue that there are more sophisticated ways that provide better system observability. However, probes are a lightweight way to determine what is broken, by providing insights into whether the service is reachable. Probes give only two answers: available or not available. This simple answer does not provide the overall health of the system, but it does provide a critical first-level insight: reachable or non-reachable.

For any service involving request/response, reachability is a prime prerequisite. Probes are tailor-made for this. For a reactive queue-based service, the same rules do not hold. Services that are not exposed to the client's throughput, other system indicators (such as memory usage and CU usage) are a better indicator of system health.

Generally, in the real world, it is not one service but a user journey involving multiple services working together, which makes sense for clients. Hence, rather than probing one service, a probe at the complete user journey makes more sense. For example, let's say we have a user journey where we ingest some data. Then, an example user journey could be as follows:

- User is authenticated.
- Ingestion process starts.
- Records are saved properly.
- Records are searchable and available for other systems.

Rather than triggering probes for individual services, it is wise to trigger a probe for the entire flow. You can configure synthetic data ingestion, which has a dataset that matches all your probable datasets expected to be ingested by this journey. This collection of data sets will be used by the periodic trigger of the workflow (probes), and the pass or fail of each of the preceding steps will be checked. The automated trigger can happen via a scheduled job on the orchestrator or cron job existing inside Kubernetes, or even by tools like cloud probe. You can have an additional check of how much time this probe will take, that is, (end time - start time). If this is below a threshold, it means your application is scaling well and in a predictable manner.

Triggering probes explicitly might not be a need in case of situations where the user journey has explicitly been used by end users. When the users are using the data, you can track the aspects of your metrics on live data, so there is no need to put extra burden on the platform.

Consider the pictorial representation of the preceding user journey, as shown in *Figure 9.1*:

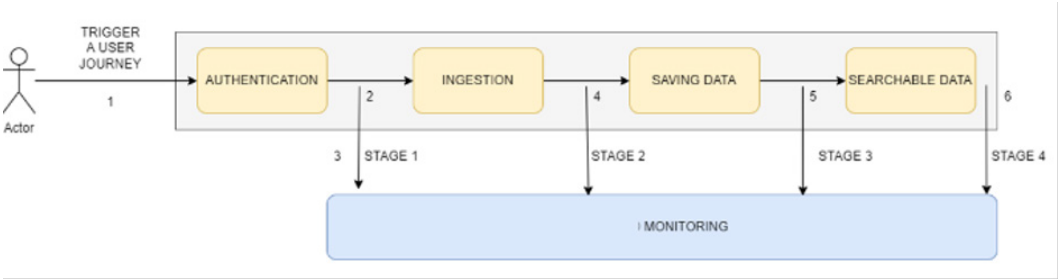


***Figure 9.1**: User journey*

The complete user journey is broken into four stages, as shown in the preceding figure. After the completion of each stage, an entry is made in cloud logging, as *stage: 1*, which means that in the user journey, stage 1 is completed successfully. You can go ahead and create a metric on this log entry, the way we did for HTTP status codes in the example under the *Log based metrics* section. On the metrics, you can also create SLO and assign error budgets.

In the preceding example, the customer satisfaction will be met only when we get stage 4 in the same count as stage 1, that is, the entire user journey is completed successfully. If that is not happening, you can easily count which stage has a problem and try to fix it.

Pillars of Kubernetes observability

Complete observability concepts/strategies in the case of Kubernetes depend on four key aspects: metrics, logs, tracing, and visualization. You will go through the hands-on experience of each pillar, with the help of one tool in each pillar. For now, let us see what each of the pillars is meant for:

- **Metrics**: Metrics are a quantitative measure of an aspect of the system. For example, a running pod involves CPU and Memory being allocated. The moment you decide to start tracking the CPU or memory, it will become a metric for us. There could be many aspects, different per application which you can wish to track. Tracking these metrics help the owner of the application identify unwanted situation being under control, and make key decisions such as scaling, to meet the need of modern workloads. The tools which are used to accomplish the collection and maintenance of metrics are Prometheus and Influx DB. Both these tools store the value of a metric (CPU usage) in a time series format.

- **Logs**: Another key pillar for observability is logs (application, system, and network logs) of activities within the system. These logs are very important for identifying issues with the general working of the application, security loopholes, and auditing. The tool generally used to perform log collection is Fluentd.
- **Tracing**: Tracing helps you get visibility into the flow requests and the components of the system. It helps identify performance bottlenecks, diagnose issues, and provides details into the component that is not scaling up/down as per the need. Tracing could be performed with Tools like Jaeger and Zipkin.
- **Visualization**: The data collected from metrics, logs, and tracing needs a UI for effectively identifying the trends and performing analysis or investigating an issue. Famous tools that are used to perform visualizations in Kubernetes are Grafana and Kibana.

Whichever tool you pick and choose for your Kubernetes deployments, it needs to be identified and known to applications, as some of these tools need an initial handshake with the deployed applications. The key is to select at least one tool for each of the mentioned five pillars. It is important with regard to debugging and identifying problems quickly and also helps resolve them.

Challenges in observability

A dynamic system like Kubernetes brings several challenges, and the story of Kubernetes for observability is also not different. While on one side, teams will love to have lots of observability aspects added to deployment, the more the observability, the more assured team members can be. However, the nature of the application and Kubernetes environment makes observability for everything, difficult to accomplish. Let's take a look at a few key challenges:

- The first key challenge with an application deployed on Kubernetes is the fact that it has a lot of moving components, both physical and logical. Physical components include nodes, networks, and disks. Logical components include Pods, services, and scalars. For the system to run properly, all these components (physical and logical) must work together. When an issue occurs with any of these components, it can become difficult to determine which component is responsible and which component is causing the issue. For example, there could be low performing network acting as a bottleneck for the applications. This could very easily be confused with some issues with the application.
- Kubernetes system is dynamic, meaning new elements of infrastructure get added or removed. Some of these are the nature of applications and the

number of instances of deployment of an application can keep changing (horizontal scaling/vertical scaling). The observability stack should be able to dynamically add and remove such components. It is challenging to keep the monitoring components up to date with the changes this dynamic system undergoes all the time.

- Another key challenge to performing effective observability is applications being deployed and updated quickly, making it challenging for the observability stack to monitor behaviors in real time. If these are not captured in real time or near real time, the whole purpose of monitoring could be defeated. For example, if the average CPU consumption is not interpreted immediately, the scaling decisions could be delayed, reducing the effectiveness of an action.

This is just a small list of challenges and is by no means a superset of all the challenges. Challenges depend on the use case. The type of application also brings in different challenges.

Exploring metrics using Prometheus and Grafana

There are multiple options available in the market when it comes to collecting metrics, monitoring, and alerting systems. A few common ones are Graphite, Nagios, Open TSDB, and of course, Prometheus.

Each of these tools score over each other given a use case. For example, Graphite is useful when applications are capable enough to send data proactively, rather than Graphite pulling the data. However, if you want the monitoring tool to pull data transparently from the applications, Prometheus serves the purpose well. In this section, you will understand the basics of how Prometheus acts as a monitoring tool and how to visualize the data on Grafana.

Prometheus is quite new in the market, so it overcomes most of the shortcomings of traditional systems in the category. It scores over its counterparts in terms of scaling, flexible query language (PromQL), push gateway (to collect metrics from batch jobs and ephemeral jobs), multiple exporters, and other tools. It actively scrapes data, stores it in a time series database, and supports queries, visualization, and creating alerts when the value for a metric goes above thresholds. It also provides API to integrate well with other visualization tools, such as Graphite and Grafana. To understand how all this is achieved, let us first discuss the architecture of Prometheus. Take a look at *Figure 9.2*:

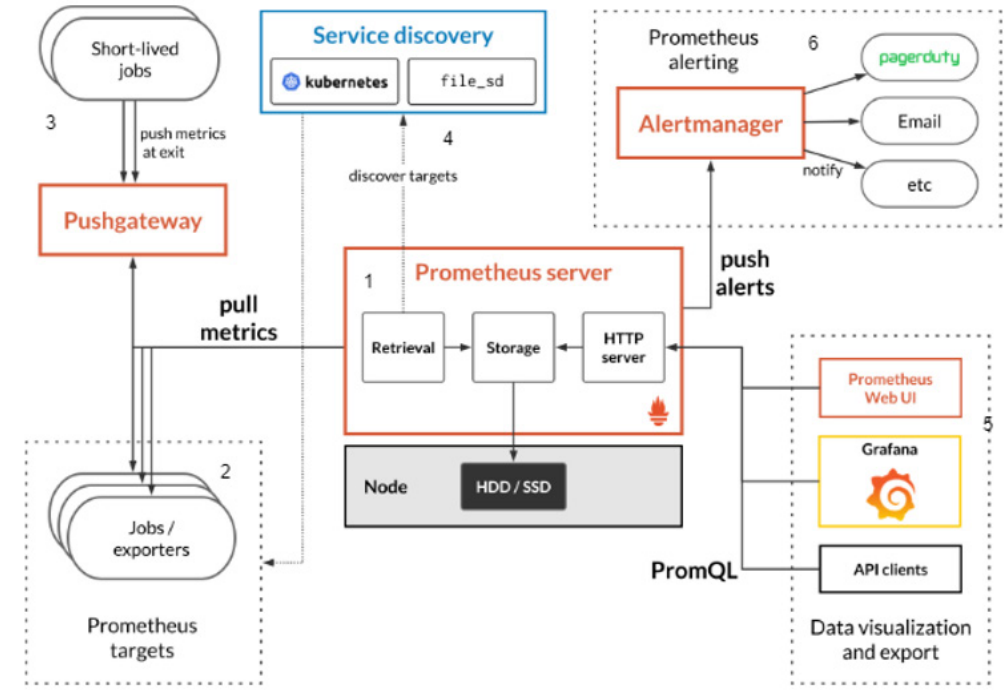


Figure 9.2: *Prometheus Architecture*
Source: https://prometheus.io/docs/introduction/overview/#architecture

Consider the figure's numerical labeling with the given numerically labeled bullet points:

1. The first component is the Prometheus server. The Prometheus server represents the core binary of the system and is divided into three components: Retrieval, Storage, and HTTP server. The retrieval component scrapes the data from target nodes, which can be a system or applications, and stores the fetched data in the storage. Prometheus stores the data locally into a custom time series database, backed up by either HDD or SSD. The stored data is made available to systems like visualization tools via an HTTP server.

2. Prometheus pulls the data from target systems using the Pull method. Applications (target system) need not send metrics to Prometheus. On the contrary, Prometheus pulls (scraping) the metrics by itself. This pull feature allows applications to easily develop and incorporate changes without worrying about how they will be monitored. Targets could be in the form of jobs or exporters. Job could be your custom-defined jobs, and exporters could be a system like a Windows machine or Linux machine to export metrics. Not only an application but a Prometheus deployment can also pull metrics from other Prometheus servers as well using the HTTP server endpoint.

3. For short-lived jobs, Prometheus supports a push gateway that could be used by these short-lived jobs to push the metrics (time series format) to an intermediary job when they had completed the processing and were about to terminate. The Prometheus pull mechanism pulls the metrics from these intermediary jobs for its purpose.

4. Once the application is ready and running to get scraped, how will Prometheus know where they reside? Service discovery is used to locate such dynamic (which adds nodes as and when needed) modern applications and their infrastructure where they are running. To make targets available, you can either hard code your targets (not a recommended approach as it might result in new nodes not getting picked for scraping), or you can use the service discovery mechanism (recommended way), which will allow automatic scraping of dynamic infrastructure. Prometheus has integration with some common service discovery mechanisms, such as DNS, Kubernetes, and Consul.

5. After the metrics are saved in the Prometheus time series database, it is made available to the end user using the three primary methods. The first one is the Prometheus-supported web UI, using which you can query raw data with the query language PromQL. Prometheus UI has features to present them as graphs too. The second is integration with third-party visualization tools like Grafana, which could be used to enhance the graphical features of metrics. The third and the last is the ability to use the API for any custom clients.

6. The last component is alert manager. Alerts are pushed from the Prometheus server into alert manager. The alert manager collects the alerts, aggregates them into groups, applies filters, silences, and throttles, and eventually goes on to use one or multiple notification channels, such as email, PagerDuty, and others, to notify the end user.

Though Prometheus pulls the metrics, for an application to allow Prometheus to pull metrics, we must add some instrumentation. Once these instrumentations are set, Prometheus can pull the metrics from your applications.

Installing Prometheus and Grafana

There are multiple strategies and ways to install Prometheus. For example, Prometheus can be installed on a Linux machine. There are multiple resources available on the internet that set up this flavor of Prometheus. Since you are reading a Kubernetes book, you are going to see the way Prometheus setup is done with Kubernetes. Prometheus setup on Kubernetes can be done in multiple ways:

1. Create all configuration files for each component of Prometheus, such as Prometheus server, alert manager, config maps, and so on, and run them in the correct order because of dependencies. This way is very inefficient and tedious and to accomplish this, you need to know the ins and outs of Prometheus well.

2. Another way to set up Prometheus is to use operators. It is better than the first one as it is easier to accomplish. An operator manages the life cycle of each component of Prometheus and ensures that all of them are up and running as one unit.

3. The third and most efficient way to deploy Prometheus is via Helm charts for the Prometheus operator. This helm chart is maintained by the Prometheus community. You will be looking at this approach in more detail. The helm chart will do the initial setup, and then the Prometheus operator will ensure that the components are running. Here's the set of commands to install Prometheus and Grafana using the community Helm charts:

```
1. helm repo add prometheus-community https://prometheus-community.
   github.io/helm-charts
2. helm repo add stable https://charts.helm.sh/stable
3. helm repo update
4. helm install prometheus prometheus-community/kube-prometheus-
   stack
```

This command assumes that you have Helm installed on the client machine. Line 1 adds the community repo of helm charts, and line 4 install one of the helm charts available in repository. It will take a couple of minutes, and once the command executes successfully, you can use the following command for listing down all pods for Prometheus deployment:

```
kubectl get pods
```

The command will produce the output shown in *Figure 9.3*:

```
NAME                                                     READY   STATUS    RESTARTS      AGE
alertmanager-prometheus-kube-prometheus-alertmanager-0   2/2     Running   1 (39s ago)   46s
prometheus-grafana-84dc5dcc8f-rpvm4                      3/3     Running   0             52s
prometheus-kube-prometheus-operator-587f9855f6-dbht5     1/1     Running   0             52s
prometheus-kube-state-metrics-5b6c6b4bd4-fdfbz           1/1     Running   0             52s
prometheus-prometheus-kube-prometheus-prometheus-0       2/2     Running   0             45s
prometheus-prometheus-node-exporter-7f9hg                1/1     Running   0             52s
prometheus-prometheus-node-exporter-qfhlq                1/1     Running   0             52s
prometheus-prometheus-node-exporter-t756p                1/1     Running   0             52s
```

Figure 9.3: *Output*

As you can see in the output, various components of Prometheus started as pods. Grafana is also running, and you can port forward it:

```
kubectl port-forward deployment/prometheus-grafana 4000
```

When you visualize this in a browser, you will see the login page for Grafana. You can feed the username as **`admin`** and password as **`prom-operator`**. When you log in successfully, you will go to the panel on the left and browse the dashboard, which is created by default. Refer to *Figure 9.4*:

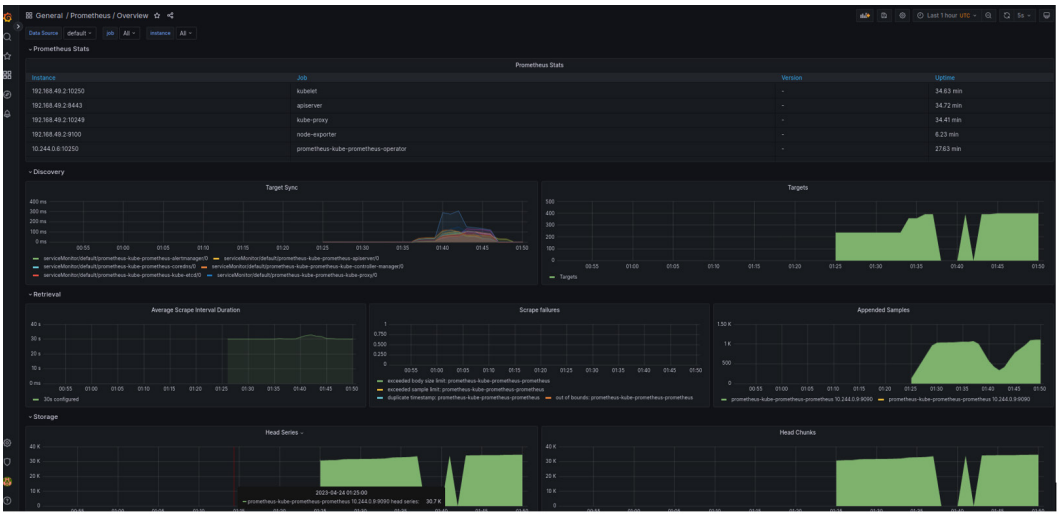


Figure 9.4: *Grafana Dashboard*

In *Figure 9.4* featuring the Grafana Prometheus overview dashboard, you can see that different metrics are been plotted. As you can observe, the Prometheus pods are running in K8s and are being pushed here as well. That signifies that Monitoring setup is complete.

You can port forward the Prometheus UI as well:

```
kubectl port-forward prometheus-prometheus-kube-prometheus-
prometheus-0 9090
```

Key page to visit on Prometheus UI is the Service Discovery option, as shown in *Figure 9.5*. This page shows which Kubernetes services are identified by Prometheus for scraping:

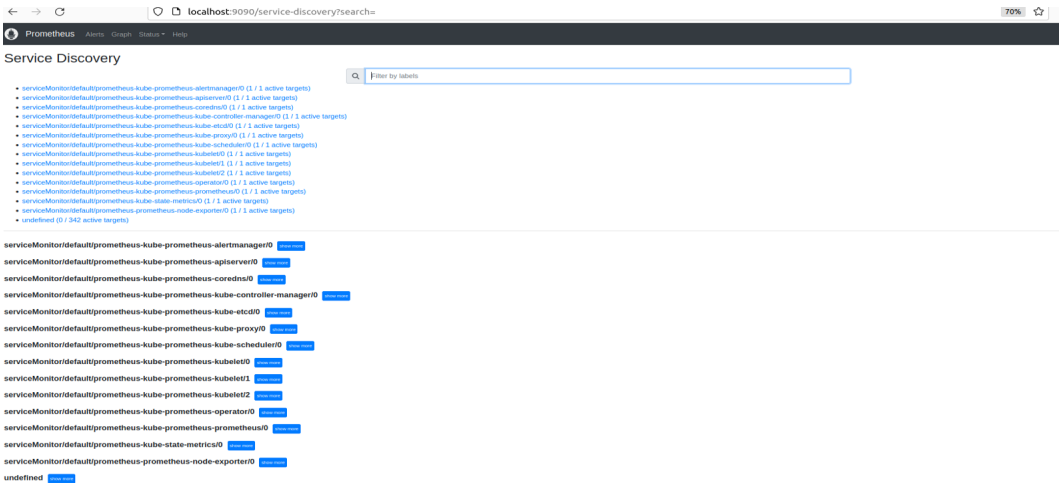


Figure 9.5: *Service Discovery option page*

As you can see, all the Prometheus-related services are discovered and pushed to Grafana UI.

The preceding helm chart installations perform several actions under the hood. If you want to take a look at what exactly is happening under the hood, refer to the official documentation. With this, the installation part is complete. In the next section, you will see how a real-time application's metrics are pulled in Prometheus and how can you create dashboards in Grafana.

Pushing custom metrics to Prometheus

In this section, you will deep dive into how to create a custom metric and expose them for scraping to Prometheus. Generally, metrics are created using Prometheus client libraries. These client libraries are available in several languages, such as Python, Go, and Java. The complete list can be found at **https://prometheus.io/docs/instrumenting/clientlibs/**.

Figure 9.6 captures the infographic representation of the strategy:

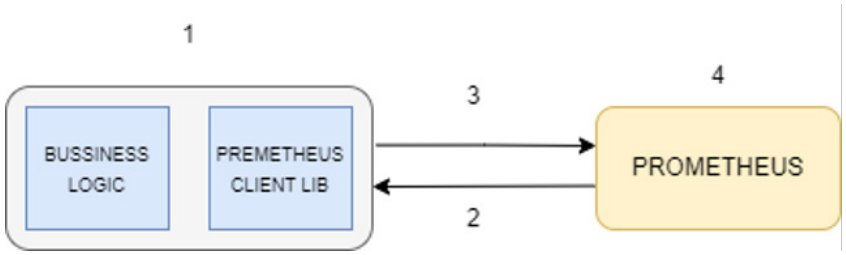


Figure 9.6: *Prometheus Custom Metric*

Consider the figure's numerical labeling with the given numerically labeled points:

1. It represents a Kubernetes service, which has pods that use the Prometheus client library to publish metrics to a location. Kubernetes also creates a service monitoring object.
2. Prometheus discovers the service using the service monitoring object, and once a service is discovered, the metrics are requested by Prometheus based on configuration parameters like scrape interval.
3. Metrics are pulled to Prometheus servers.
4. On the service discovery dashboard, the service starts appearing.

For step number one, consider the Python code shared in the attached codebase with the name **`custom-metric-standalaone.py`**. It is an oversimplified example, where we simply create a metric to track time spent and requests made. Install Python and **`prometheus_client`** (using the **`pip install`** command). Run the Python script and visit **http://localhost:8000/metrics** to see the metric it is producing in the preceding URL. This example is taken from the official documentation.

Now, to use the example in an actual real-world scenario, you must package this application as a docker image. This is already done and pushed to the docker hub repository.

After this, you can use the preceding image to create pods and define service. Refer to the **`custom-metric-service.yml`** and **`custom-metric-deployment.yml`** files. Apply them one by one using the **`kubectl apply`** command. It will result in a Kubernetes service with the name **`custom-metric-python-svc`** being created.

```
kubectl apply -f custom-metric-deployemnt.yml

kubectl apply -f custom-metric-service.yml
```

Now, the last step is to add this service to the service monitor so that Prometheus registers this service for scraping. Following is the service monitor yaml file:

```
1.  apiVersion: monitoring.coreos.com/v1
2.  kind: ServiceMonitor
3.  metadata:
4.    name: custom-metrics-servicemonitor
5.    labels:
```

```
6.      app: custom-metrics-servicemonitor
7.      release: prometheus
8.  spec:
9.    selector:
10.     matchLabels:
11.       app: custom-metric-python-prometheus
12.   endpoints:
13.   - port: metrics
14.     interval: 15s
```

While most of the tags are self-explanatory, this service monitor is tied to the service you created before with the label **`custom-metric-python-prometheus`** (line 11), and the scraping frequency is set to 15 seconds (line 14).

Creating dashboard on the metrics using Grafana

As you have already seen a snapshot of the Grafana dashboard, you can create a new dashboard using the UI. However, in real-world scenarios, creating a Grafana dashboard using the UI is not a practical approach. Grafana does provide an automated way to create and propagate the Grafana dashboard to all environments in a multi-environment setup. Grafana provides a mature set of APIs to create not only dashboards but also configure alerts. The list of APIs can be accessed at **https://grafana.com/docs/grafana/latest/developers/http_api/**.

It is always recommended to create a dashboard using the UI first. Once the dashboard is created, export it as a JSON file and commit the JSON in version control.

Your deployment pipelines should read these JSONs and use the API to create the dashboards in higher environments. The same strategy is true for creating other features like Alerts.

Consider *Figure 9.7*, which shows creating a custom dashboard using Grafana UI:

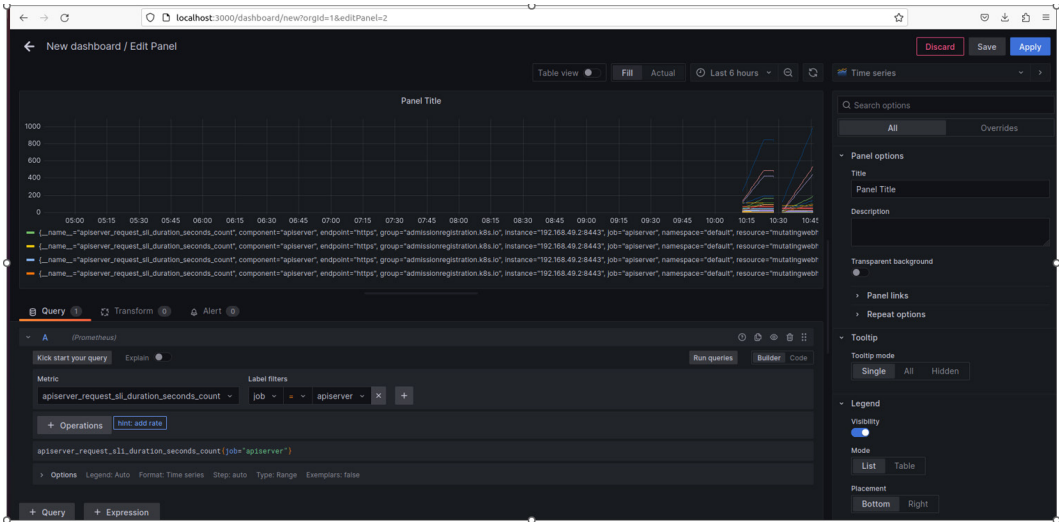


***Figure 9.7**: Configuring New Dashboard*

Next, you must save it and export the JSON. Save the JSON in Github or any source version control, and use the same JSON with the CRUD API defined earlier. Below is the curl, which allows you to pass the JSON in raw format:

```
curl --location --request POST '/api/dashboards/db HTTP/1.1' \ --header
'Content-Type: text/plain' --data-raw '<exported json>'
```

While running this command across environment, it is important to feed the requests with environment-specific variables.

Logging and tracing

Generating logs (application and system logs) from an application is a very common feature in software applications, which is done to identify what is happening inside your application as well as infrastructure.

While applications have some form of native logging mechanism, in the case of a distributed and containerized environment like Kubernetes, a centralized logging mechanism is preferred. A centralized logging mechanism facilitates logs from different applications, generating logs in different formats to be stored at a common logging back end, helping make the processing and analysis of logs possible. In this section, you will get to know the Kubernetes-supported resources needed to implement the logging functionality using Fluentd.

Tracing is a mechanism to identify and implicitly observe how a set of services interact with each other. It is done by each service emitting some information about

the actions performed. We call the single signals **spans** that have a name and an identifier. Multiple spans constitute a **trace**. A trace is associated with a **traceId**, and this **traceId** is used by the application to emit information as spans. While logs provide insights into what is happening inside a single microservice, which is useful for troubleshooting and auditing, tracing helps identify, troubleshoot, and audit information about the interaction between multiple microservices. You will look at how tracing is done in a distributed setup like Kubernetes using Open Telemetry and Jaeger.

Logging using Fluentd

Microservice deployed in Kubernetes containers (docker) emits logs to standard output (**stdout**) and standard error (**stderr**) streams. Docker containers direct the streams to a logging driver configured in Kubernetes to write the logs in JSON format. With this setup, users can fetch the logs of currently running containers via the **kubectl logs** command and the previous container by setting the previous flag to true. The previous container implies that the container got crashed and has now been restarted:

```
kubectl logs -c <container-name> <pod-name>

kubectl logs -c <container-name> <pod-name> --previous=true
```

The condition for this command to work is that the pod should be running. In cases where the pod is deleted from a node, the log details are also deleted. Similar is the situation when the node dies. The user in this case will also be unable to fetch logs. To overcome this loss of logs, logs must be shifted to a central location, whose life cycle does not depend on pods or nodes. Kubernetes does not provide this central storage location for logs, and this is where a tool like Fluentd comes into the picture.

To enable a centralized collection of logs, there are three patterns widely used in the industry:

- Push logs to the centralized location using a logging sidecar run inside the applications pod.
- Push logs to the centralized location by introducing a node-level logging agent on every node.
- Push logs to a centralized location by applications by directly writing logs to a remote centralized location.

Among these three patterns, for the first and third patterns, your application or pod definitions must introduce some constructs that can push logs. In the first option, side car must be introduced by all the services. Similarly, in the third option, all services must write logs using APIs that directly write to the remote central location. However, collecting logs is more of a cross-cutting concern, and different services

should ideally have this available without making any changes. Here, the second option comes to the rescue.

For implementing the second option, you must deploy a node-level logging agent on all the nodes; this agent is mostly a container with access to all the logs. As you have seen, the Kubernetes cluster generally has multiple nodes, so ideally, each new node should have an installation. Moreover, when a new node is added, the agent must be up and running there as well. The simplest way to achieve this in Kubernetes is to create a Daemon set and let Daemon set controller ensure that each new node has the agent running as a container. Daemon Set controller will check the number of nodes periodically and create a new logging agent when a new node is added. Using Daemon Set for logging is very useful as you need just one agent running on all nodes, and with the approach, the application does not need to make any changes in its codebase. One limitation of this approach is that it works only for applications that have standard output and error streams. Consider *Figure 9.8* to understand how Fluentd pushes logs to a common location:

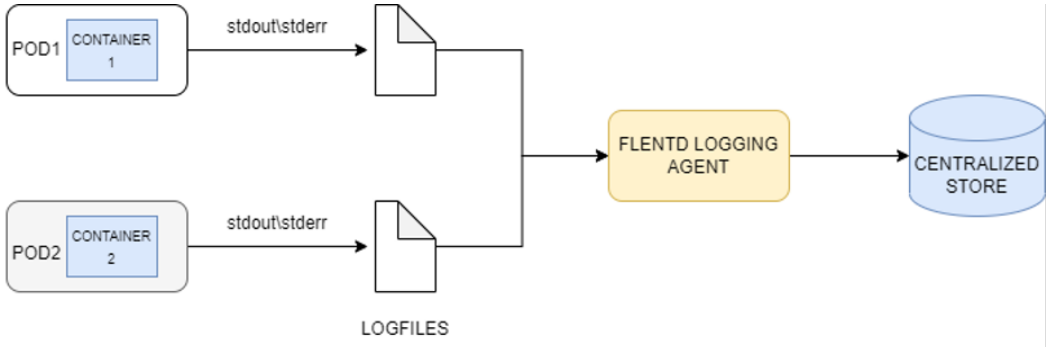


Figure 9.8: *Fluentd Logging Setup*

In the preceding figure, POD 1 has Container 1 running inside it and POD 2 has Container 2 running inside it. The **`stdout`** and **`stderr`** files are fetched by the Fluentd Logging agent and pushed to a centralized store, such as elastic search. Elastic search's search capability is then used to analyze the logs.

For setting this up, refer to the YAML file available in Fluentd-managed GitHub repository:

https://github.com/fluent/fluentd-kubernetes-daemonset/blob/master/fluentd-daemonset-elasticsearch-rbac.yaml

This aggregated YAML file has multiple Kubernetes object definitions. Let us go over them one-by-one.

- Line 1-7: Create a Fluentd identity by creating a service account
- Line 9-23: Grants Fluentd access to read, list and watch pods

- Line 25-36: Creates a cluster role binding, for Service account created above with the access
- Line 38-121: Definition of Daemon set; there are some key things to note in this definition:
    - The Daemon Set uses elastic search docker image (**`fluent/fluentd-kubernetes-daemonset:v1-debian-elasticsearch`**) configured as destination for Fluentd output at line number 66.
- You need to provide elastic search configurations (line 72-91). Key configurations are host, port, and credentials. Tweak these details if you want to use a separate elastic search cluster for pushing logs.
- Fluentd needs root permission to read logs in **`/var/log`** and write **`pos_file`** to **`/var/log`**.

With the preceding setup done, your applications/service deployed. This deployed application is producing logs, and the logs will be available in elastic search instances. Fluentd is just a good example of distributed log collection; you can use various other alternatives as per your use case, such as Logstash and Splunk. There is a very famous combination of tools known as ELK stack, which represents Elastic Search to store distributed logs, Logstash to collect logs (replacement of Fluentd), and Kibana to visualize the data and create relevant dashboards.

Tracing with Open Telemetry using Jaeger

A user journey triggered by some user action in a microservice environment eventually propagates to multiple microservices being called in some order. A microservice not producing results in an expected time will result in the overall performance of the application going down. Since these microservices are packaged as containers and run on multiple nodes, the method to track the flow of request as it is processed by different microservices in the system is known as Distributed Tracing. In this section, you will understand how to implement distributed tracing using Open Telemetry and Jaeger. Take a look at *Figure 9.9*:

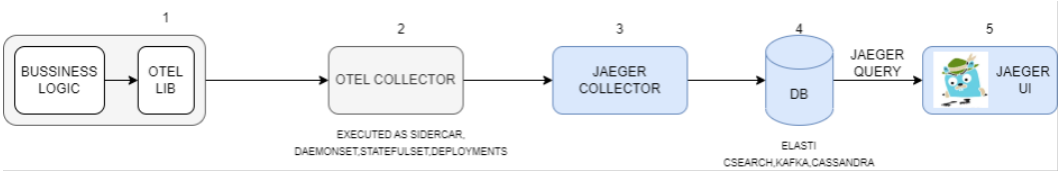


Figure 9.9: *Tracing with Otel and Jaeger*

Consider the figure's numerical labeling with the given numerically labeled points:

1. A microservice application that has business logic running inside it uses an **Open Telemetry Library** (**OTel**) to produce traces.

2. These traces created by the OTel library are collected via an OTel Collector, which is deployed in the Kubernetes cluster as a sidecar, daemon sets, stateful set, and deployment.
3. Open Telemetry Collector hands over the tracing details to Jaeger Collector.
4. Database to store the tracing information collected from Jaeger collector. There are multiple options available, such as Elastic Search, Cassandra, and Kafka.
5. Finally, the tracing information saved in the database in step number 4 is available to view and analyze by Jaeger UI.

Let us start the setup. You will see that helm charts have been used to describe installations:

1. Install Jaeger using the following code:

   ```
   helm repo add jaegertracing https://jaegertracing.github.io/helm-charts

   helm upgrade -I jaeger jaegertracing/jaeger
   ```

   Once the commands are successfully executed, run **`kubectl get pods | grep jaeger`** to get the details about the pods running. You get the following results:

```
jaeger-agent-5wl68                  1/1     Running     0                 8m38s
jaeger-agent-7xwmb                  1/1     Running     0                 8m38s
jaeger-agent-ln9jp                  1/1     Running     0                 8m38s
jaeger-cassandra-0                  1/1     Running     0                 8m38s
jaeger-cassandra-1                  1/1     Running     0                 6m34s
jaeger-cassandra-2                  1/1     Running     0                 4m29s
jaeger-cassandra-schema-vhnzd       0/1     Completed   1                 8m37s
jaeger-collector-57df656b86-bdq4q   1/1     Running     5 (7m11s ago)     8m38s
jaeger-query-5c85755697-srvh4       2/2     Running     5 (7m ago)        8m38s
```

Figure 9.10: *Jaeger Pods*

   As you can see, Cassandra is installed as the back end. You can also see that there are three agents running because the cluster has three nodes configured. The jaeger-query pod is the pod that holds the UI which lets you analyze the tracing data. To get to the UI, open the browser and type **http://127.0.0.1:8080/** after triggering the following command on CLI:

   ```
   export POD_NAME=$(kubectl get pods –namespace default -l "app.kubernetes.io/instance=jaeger,app.kubernetes.io/component=query" -o jsonpath="{.items[0].metadata.name}")

   echo http://127.0.0.1:8080/

   kubectl port-forward --namespace default $POD_NAME 8080:16686
   ```

2. Now, install Open Telemetry Operator using the following code:

```
helm repo add open-telemetry https://open-telemetry.github.io/
opentelemetry-helm-charts

helm repo update

helm install   opentelemetry-operator open-telemetry/
opentelemetry-operator
```

3. The next step is to install Open Telemetry collector. For this, refer to the file attached with codebase with the name **`opentelemetry-collector-demonset.yaml`** and apply the file using the **`kubectl apply`** command.

   Type the **`kubectl get po |grep collector`** command to see the result of the preeding installation. You will get the result shown in *Figure 9.11*:

```
jaeger-collector-57df656b86-bdq4q                                 1/1     Running
otel-collector-daemonset-collector-2grpc                          1/1     Running
otel-collector-daemonset-collector-v4pn9                          1/1     Running
otel-collector-daemonset-collector-vlwln                          1/1     Running
```

Figure 9.11: *Open Telemetry Pods*

4. The next step is to create an application. Refer to the **`tracing-app.yaml`** file attached with the codebase. It is a Java microservice that uses SDK to log data as per open telemetry standards.

```
kubectl apply -f tracing-app.yaml
```

5. If you run **`get kubectl get svc`** on the cluster, assuming there was nothing installed earlier, you will see the services shown in *Figure 9.12* running:

```
NAME                                              TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)                                        AGE
alertmanager-operated                             ClusterIP      None            <none>           9093/TCP,9094/TCP,9094/UDP                     26m
jaeger-agent                                      ClusterIP      10.8.13.172     <none>           5775/UDP,6831/UDP,6832/UDP,5778/TCP,14271/TCP  28m
jaeger-cassandra                                  ClusterIP      None            <none>           7000/TCP,7001/TCP,7199/TCP,9042/TCP,9160/TCP   28m
jaeger-collector                                  ClusterIP      10.8.15.53      <none>           14250/TCP,14268/TCP,14269/TCP                  28m
jaeger-query                                      ClusterIP      10.8.3.59       <none>           80/TCP,16685/TCP,16687/TCP                     28m
kubernetes                                        ClusterIP      10.8.0.1        <none>           443/TCP                                        33m
my-release-kube-prometheus-alertmanager           ClusterIP      10.8.7.232      <none>           9093/TCP                                       26m
my-release-kube-prometheus-blackbox-exporter      ClusterIP      10.8.4.228      <none>           19115/TCP                                      26m
my-release-kube-prometheus-operator               ClusterIP      10.8.10.12      <none>           8080/TCP                                       26m
my-release-kube-prometheus-prometheus             ClusterIP      10.8.5.116      <none>           9090/TCP                                       26m
my-release-kube-state-metrics                     ClusterIP      10.8.2.89       <none>           8080/TCP                                       26m
my-release-node-exporter                          ClusterIP      10.8.13.61      <none>           9100/TCP                                       26m
prometheus-operated                               ClusterIP      None            <none>           9090/TCP                                       26m
tracing-demo-app                                  LoadBalancer   10.8.12.20      34.121.184.199   8080:30124/TCP                                 84s
```

Figure 9.12: *Prometheus and Jaeger Services*

Observe **`tracing-demo-app`**; it is an app exposed outside. You can trigger multiple curl commands, as follows, to generate some load. The image used inside **`tracing-demo-app`** is an open-source image (**`quay.io/rbaumgar/otelcol-demo-app-jvm`**), which is built with the intention to demo the open telemetry tracing.

You can use the preceding application to generate some load. Here are the java files that serve the purpose: **https://github.com/rbaumgar/otelcol-demo-app/blob/main/src/main/java/org/acme/opentelemetry/TracedResource.java**.

To generate load, you can send the following requests from the command line:

```
curl http://34.121.184.199:8080/hello
curl http://34.121.184.199:8080/sayHello/swapnil
```

Defining a typical SRE process

There is no definition of what falls in the SRE process and what does not. However, looking at various use cases, a typical SRE process is more of a set of the best practices. *Figure 9.13* shows a very high-level responsibility for SRE:

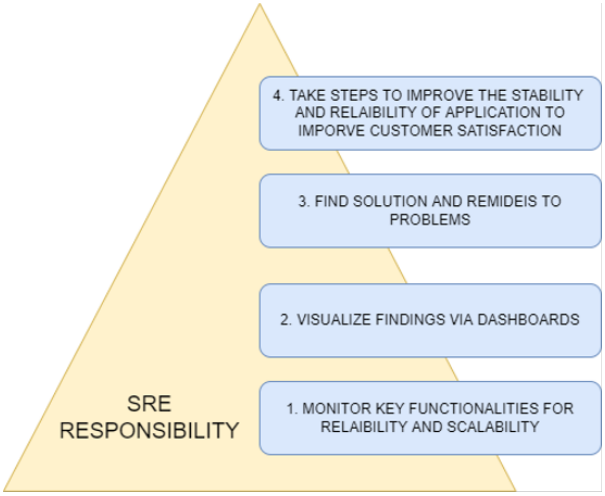


Figure 9.13: *SRE responsibility*

These best practices include the following:

- The SRE process involves monitoring data using some monitoring tools, such as *Datadog*, *Prometheus*, *Nagios*, and so on, to collect information from the system on how it is performing. Automated alerts are used when an abnormality is detected.
- When something goes wrong, SRE teams manage and apply the backup plan to deal with the issue and restore the appropriate state. This backup plan is known as **runbooks**.
- When an SRE team learns and matures by performing retrospectives of incidents, they eventually are better equipped to handle such situations better the next time they face it.
- By looking at an incident, the SRE team intends to identify the impact on customers with SLIs and SLOs. This helps them frame responses about such mishaps in the best possible way.

- SRE teams can easily facilitate and help in deciding how fast the development team can move, by releasing new features. They utilize the error budgets to make the best decision for the customers.

The mentioned practices help the teams in releasing newly developed software on production without impacting customers. Each SRE team matures with time, learning from the type, frequency, and cause of any issue, resulting in better handling of unwanted situations for your clients.

In the last section of the chapter, you will understand the responsibilities of SRE teams in depth. Although we did talk about them in the previous section, we will dive deep into this topic now.

Responsibilities of SRE

The term **Site Reliability Engineering** (**SRE**), conceptualized by Google Engineer Ben Treynor in 2003, aims to increase the reliability of the sites and services that a team offers. According to Treynor's definition, *SRE is what happens when you ask a software engineer to design an operations team.*

At the core, an SRE team is a group of software engineers who build and implement software to maintain the reliability of system/services.

Consider an example of a service: an online banking system that ideally has hundreds of financial transactions happening every second. If this service goes down frequently, it is considered as a loss of business for the bank. A new role of Site Reliability Engineer is introduced in software industry, with main tasks being to ensure that the reliability of the system is maintained without impacting the velocity of new features rolling out.

In a traditional system, there were two parties. One was the developer, whose aim was to build new features at a fast pace, and the other was the operator, whose intent was to deploy the new features without compromising the stability of the system. Because of the different intents, there was always a conflict between the two teams. To resolve this, a new role of DevOps was introduced. While DevOps culture did talk about deploying code fast and in high frequencies, reliability and stability were not generally given importance. In the DevOps team, there is no person whose responsibility is to ensure system reliability. This is where the SRE came in. At this point, you need to think about which actions can make a system/service unreliable. The major reason behind this is the change in platform, change in services themselves (new features), infrastructure changes, and so on. The solution to this could be to make no changes or to limit the number of changes to the system. However, that limits the business. Instead, we want to write more code and develop new features quickly, making the application better and thus, increasing business value. SRE tries to automate the process of evaluation of the effects the changes will have. Since this

automation has no discussions with other teams and no manual checks, it makes releasing the changes fast and safe. In this chapter, you will investigate how to build and deploy the automation aspects of assessing the effectiveness of changes.

The SRE team's primary task is to concentrate on engineering new products and services by automating and scaling applications and production environments. Even though SRE concentrates on operational stability, it also helps reduce the friction of handovers from product development teams. Each SRE team follows a few core principles:

- You cannot get away from failures; they are bound to happen. However, you can learn from them.
- Automate wherever possible to minimize manual works/reworks.
- The SRE team and the engineering team work together to find the issues that lead to system breaks. No blaming is involved. Both parties are equally responsible for the proper running of the system.
- While a lot of effort could be spent on making the service more reliable, it is important to put in the right amount of effort that satisfies the end user. The efforts saved could be used elsewhere.

Incident management

Although everybody wants their services to run without a hiccup, practically speaking, they may fail or go down. When such an unwanted event happens over a continued overtime window, it is known as an **Incident**. Though the primary aim of SRE is to ensure that such situations do not occur, if they do happen, how well they restore the system to normal depends on the capability of the SRE.

Resolving an incident means restoring the service to normal or mitigating the impact of the issue. Managing incident means coordinating the efforts of multiple teams efficiently and ensuring effective communication between the engineering parties/teams. The basic aim is to respond to an incident in a very structured way. Incidents could be confusing, and a well-thought-through strategy of action when such incidents occur reduces the time to recover.

To ensure the readiness of the plan and effectively apply the plan, SRE team members lead, perform, and ensure the following actions/best practices:

- **Prioritization**: The SRE team identifies the service-creating issue and may decide to shut down or restore the service, ensuring that they preserve sufficient information behind the cause of the incident.
- **Preparations**: The SRE team prepares for these unwanted situations or incidents by properly documenting the incident management process in advance, in collaboration with all the participants in the incidents.

- **Managing incident resolution**: When the indent occurs, SRE team members manage the whole resolution activity. Incidents are tough to resolve and might take days as well, and it might result in a few members' emotional states becoming cranky. They manage the environment so that your team is moving continuously in the right direction of resolution.
- **Periodic improvements**: The SRE team periodically re-evaluates the plan to improve it further and make it more effective. The plan gets updated with the learnings from each incident.
- **Drills**: The SRE team performs drills where they synthetically create an incident and situation of an incident to measure the effectiveness or correctness of the plan.

Playbook maintenance

The playbook contains high-level instructions on how to react to alerts. It is important to keep in mind that this critical alert is just a temporary thing and might fade in some time. Playbooks contain explanation about the why, how, and what of an alert, and attach the severity of it. It also has steps written to resolve the alert. Whenever an alert is generated, the playbook must be updated with every incident to make sure we document the reason for the latest alerts.

The contents of playbooks can very easily go out of date, so they need updates after every incident and reviews after every major release. If you had created a playbook that contains a lot of details, the frequency of change will also be high, as compared to when you create a generic playbook. It varies from team to team; some like to maintain lots of information and steps, which means they will rely more on the playbook to handle the situation rather than putting their mind to it. On the other hand, it could contain just the basic details, and the SRE who is working puts their mind to analyzing everything about an issue from scratch.

Drills

A drill is a periodic activity, where the SRE team synthetically tries to replicate a failure scenario and take the planned action against it. If after doing that activity, the team is able to restore the service completely, it is assumed to be a successful drill, or else the drill is marked as a failed drill. In case of a failed drill, introspection of the process is needed and optionally, a review of the documents (playbooks) available for correctness and effectiveness is also done.

Such drills help enterprises come up with Mean Time to Recovery (**MTTR**) numbers for the end users. The time taken by most of the drills is assumed to be the time that could be quoted for MTTR.

Selecting monitoring, metrics and visualization tools

We have already discussed all aspects of monitoring, metric and visualizations. This part of the chapter is going to cover some important qualities and expectations from a metric, monitoring and visualization tech stack. These qualitative expectations will help you choose the right system/tech stack for you monitoring needs. The expectations and qualities are as follows:

- **Independent life cycle**: One key requirement while selecting a monitoring system/stack is that it should be outside the boundaries of other services. Generally, each monitoring system has some effect on the business system it is monitoring, but its impact should be minimal. The monitoring system should be hosted separately, and the interaction between services and monitoring system has to be very short lived.
- **Reliable**: Another key expectation from the monitoring stack is that it is reliable. It produces the right metrics, alerts and dashboard, which could be used with complete reliance to take decisions. These systems are expected to be highly available and generally have self-healing and managing nature. These systems are expected to manage and maintain historical data of metrics, so as and when the application grows old, the data set size increases. This increase should not impact system performance.
- **Ease of usage**: This system has details about key insights into your applications. Generally, SRE when monitoring a system combines multiple such insights to reach a conclusion. It is imperative to have an easy-to-use system. Otherwise, rather than concentrating on analysis of monitoring data, emphasis will be on using the system, which could adversely hamper the results and eventually result in making the wrong decisions. Grafana is so popular because of its capability to produce charts on some very complex metrics.
- **Correlating factors from multiple sources**: Monitoring systems are expected to present a holistic view of your entire application. The system is expected to be a heterogenous one, that is, using different kind of tools and technologies and having different infrastructural footprints. An effective monitoring system should be able to capture the metrics from such heterogenous systems, the monitoring system has a mechanism to correlate the data from multiple sources and produce effective insights.
- **Automated detection of dynamic behavior**: One key dynamic behavior of a Kubernetes-based application is that the infrastructure is very dynamic. Nodes come and go; pod gets created and deleted. An effective observability

tools should be smart enough to detect such changes and start gathering metrices automatically. A manual intervention needed for such a situation will make the monitoring system difficult to use, and the insights will be inaccurate.

- **Powerful and low latency alerting**: Last but not least, the expectation from good monitoring systems is to have a powerful alerting mechanism. The duration between event generation and reporting should be minimal to take quick actions. Another important point is the channels supported, like emails, SMS, and mobile notifications.

Conclusion

The difference between monolithic application and microservice application results in the differences in observability of applications as well. The observability process and technology have advanced quite a lot with wide adoption of the architecture. Terms like SLI and SLO are more heard of and have more profound meaning in today's world, and the popularity of the role of SRE is also much more. The whole observability logic is based on systems producing metrics and tools evaluating metrics against a configured value to generate alerts. While each public cloud that supports managed Kubernetes cluster has a well-defined observability stack defined, the cloud native world has its own open-source stack defined. For scraping metrics, Prometheus is a market leading option. Grafana is used for visualization, and Fuentd is used for managing logs.

The technology stack keeps evolving over period, and by no means is it limited to tools discussed in the current chapter.

Points to remember

- Microservice has redefined the terms SLI and SLO in their own language, which is different from the traditional definition.
- With observability gaining importance these days, the role of SRE has gained lot of importance.
- It is needed to push metrics, logs and tracing data to a common centralized time series backed tool.
- For distributed logging, Fluentd is a good option. Fluentd collects logs by running an agent at each node of Kubernetes cluster; hence, applications are agnostic to the nature of Fluentd.
- Prometheus is used for the scraping metric; it scrapes the metrics periodically and stores the information into its own time series database.

- The metric collected in Prometheus can be connected to Grafana dashboard for visualization. All the tasks an SRE performs after that are smart slice and dice, and grouping of metrics on dashboard to generate right insights.
- A user journey comprises of multiple microservice being called one by one. A latency introduced in a microservice may lead to the entire user journey taking time. To tackle such bottlenecks, distributed tracing is done, which efficiently captures the knowledge of which microservice took what amount of time. This can be achieved via tools like Jaeger, Grafana Tempo and Zip kin.

Questions

1. SLAs are decided after engineering teams and business owners agree to an availability number. Is this true or false?
2. How does observability stack allow defining custom mathematical functions for SLI calculation?
3. Working on a resolution of an incident is the totally responsibility of SRE. Is this true or false?
4. Name tools for:
    a. Distributed tracing
    b. Distributed logging
    c. Visualization
    d. Monitoring

Answers

1. False. Ideally, a debate happens on SLO. If the service runs over a period meeting SLO numbers, then we define the SLA accordingly.
2. You can define custom functions using the Monitoring Query language. Once the metrics are aggregated, create an SLI and then SLO, and configure alerts. In case of Prometheus, it is done by writing smart queries using PromQL.
3. False. It is the responsibility of the entire team, including DevOps.
4. The tools are as follows:
    a. Distributed Tracing: Jaeger, Tempo, Zipkin
    b. Distributed logging: Fluentd
    c. Visualization: Grafana, Kibana
    d. Monitoring: Prometheus

Chapter 10
Effective Scaling

Introduction

A production-ready containerized application needs to be scalable. Scalable means not just adding more infrastructure but also decreasing it when there is no need for it. Generally, you can easily scale container applications by creating pods manually, as and when you feel that your SLAs are getting breached. The infrastructure needs of applications vary over time. At a certain time in the day, they might need X amount of infra (CPU and memory) and in a few hours of the day, they might need 5X or 10X amount of infra. If your application is unable to respond with the sufficient infrastructure required at any given time, it will be unable to finish the work in expected timelines and might just beat the entire purpose of application.

This kind of applications (micro service or any other containerized workloads) should be enabled with handling spikes in infrastructure requirements throughout the day, without any manual interventions. These applications generally have a qualitative or a quantitative parameter, which defines the need for scaling, and that parameter is used as a signal to increase or decrease infrastructures. Kubernetes provides a very well-created and matured scaling mechanism for container applications and thus, helps applications comply with targeted SLA.

In this chapter, you will look at the aspects of scaling discussed in depth, giving you enough expertise to design effective scaling for your application.

Structure

In this chapter, we will discuss the following topics:

- Needs of scaling microservices individually
- Principles of scaling
- Challenges of scaling
- Introduction to the autoscaling algorithm
- Types of scaling in K8s
- Best practices of scaling

Objectives

After studying this chapter, you will be able to identify the right parameters, tracking which, your applications should ideally define scaling. You will also see how you can combine multiple such scaling parameters aspects and how can you define your custom parameter-to-scale application. You will understand when to opt for horizontal scaling and when to go for vertical scaling. In the last section of this chapter, you will get insights into some key factors and some guidelines that will help your application scale in a production-ready environment.

Needs of scaling microservices individually

In case of microservices architecture, your application is broken down into logical and meaningful chunks; these chunks are sufficient to be running in isolation. Each micro service serves just one business function, and if this is done well, scaling can be done effectively. Take a look at *Figure 10.1:*

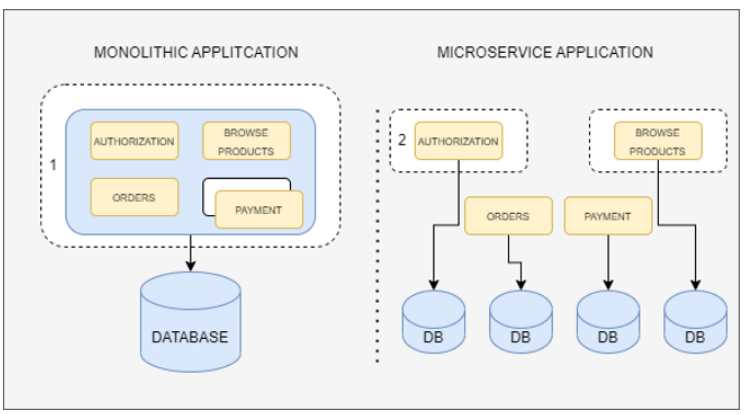


Figure 10.1: *Monolith vs microservices scaling*

In the preceding figure, the left side shows the monolith architecture, where all the modules (**AUTHORIZATION**, **BROWSE PRODUCTS**, **ORDERS**, and **PAYMENT**) are part of one application, and scaling of the application for one of the modules results in overall scaling of application (1), assigning unnecessary infrastructure to modules that do not need it.

When it comes to scaling microservices, there is a need to scale up/down different parts of the system at different times. For example, authentication and authorization services will be used more than order placement services in an e-commerce app (2). The primary objective of scaling such an application is to make sure authentication, authorization, and order placement service perform their duties without breaching SLAs. The infrastructure available to deploy is finite (with an upper limit) and hence, resources should not be over or underutilized at any point in time. Resources should be allocated to different areas of the application based on need.

When we use monolithic systems, you cannot break the system into chunks. Hence, when you scale, even the portions of applications that do not need scaling were allocated resources, which were often underutilized. This means wastage and potential breach of SLAs, and for meeting SLAs, you add more resources, leading to high costs. The cost aspect gains more importance with various cloud providers coming up with their pay-as-you-go model. The more you use it, the more you must pay for it.

Principles of scaling

The efficiency of the application is a key principle not just behind a microservice application but for any application. While it is much simpler to quantify the term efficiency for a single system (a monolithic application), it is very difficult to measure it in a distributed system or a system like microservices. However, the more distributed the system, or the higher the number of micros services, the less of a difference will one microservice have on the application. Two of the production readiness must-haves for a microservice setup are scalability and performance. Both factors help make microservices more efficient and increase the availability of microservices.

Modern workloads ensure one thing for sure: workloads are steadily increasing (and these can fluctuate at different times of the day) and to handle the workload, each microservice needs to be able to scale up independently, with the assumption that other microservices are also capable of handling scaling themselves.

To ensure that microservices are performant and scalable, you need to understand their growth scale, qualitative and quantitative, so that it could be planned well. Qualitative scaling means that the microservice could be made available to all the other services in the ecosystem, as and when needed. Quantitative means the microservices can handle the workload and responds in the expected time

(within SLA), even if the load fluctuates frequently. For ensuring that you met the quantitative parameters well, first of all, it is important to use the allocated hardware resources well, ensure that more resources could be added as and when needed, and ensure that you are aware of the resource bottlenecks outside the microservices. You must ensure that the dependencies on a microservice scale up as well when needed, that is, traffic is managed in a scalable way, data stored in a scalable way, and more important, tasks handled in a performant manner.

The key principles of scaling are as follows:

- One should know the growth scales (qualitative and quantitative), and looking at the plan, capacity provisioning is planned as well.
- Microservices are supposed to use allocated hardware resources optimally. Moreover, the microservice is written such that multiple parallel threads are not interrelated, and they support scaling.
- Resource requirements and associated bottlenecks are identified and planned.
- Capacity allocation is automated and performed on a scheduled basis, without manual intervention.
- When a microservice scales, all its dependencies should scale independently.
- Traffic patterns are understood well, and traffic could be re-routed in case of failures.
- Read and write of data should be handled in a scalable manner.

You can go ahead and identify more such points, but the rule of the game is simple: a microservice should scale up/down automatically when the workload increases or decreases, and in doing so, all possible type of bottlenecks needs to be identified and action needs to be taken to resolve them.

Challenges of scaling

Microservices scaling brings in a lot of challenges. For example, there are challenges related to traffic management, consistency, logging, monitoring, resource allocations, and many more. Among the plethora of issues that might arise while scaling microservices, in this section, we will talk about issues related to adding more infrastructure to microservices. Following are some key resource challenges of scaling microservices:

- **Flexibility and speed:** Modern workloads use various kinds of resources, CPU vs GPU, SSD vs HDD, and so on. Moreover, the usage patterns of modern workloads fluctuate quite frequently. At some point in time, we

need 2 instances serving the end users well, and at other times of the day, 10 such instances also do not suffice. This usage of varied hardware and the fluctuating pattern is making capacity planning very difficult. It is difficult to guess the right amount of infra capacity needed by an application. This problem, however, is less dominant if you are using public clouds, as they provide the pay-as-you-go model. Hence, adding more resources to your application simply means burning more money.

- **Convenience:** Another key challenge when it comes to adding resources for your microservice is convenience. Convenience here implies questions like, 'Do we have enough infrastructure available?', 'How well does the application utilize the allocated infrastructure?', and 'How convenient is it to scale the application up/down without breaking any flows?'. It also covers the ease with which the entire system scales up/down without bottlenecks.
- **Cost:** Adding resources to your microservice application comes with a cost, be it on-premise or cloud deployment. In some organizations, asking for more resources is not a big event, but it is in others. Hence, the scaling up down procedure needs to adhere to the budgets allocated, and it could become a challenge if the usage of certain microservices is showing an increasing trend.
- **Strategy:** By strategy, we mean how to identify the initial resource needs of microservices and what will be the scale up/down needs of the application. You can scale up your system when the average CPU usage goes above 80%. However, this 80% varies from application to application, and the best number will only be identified when the applications serve the production environment. Read the stats around the resource consumption and then take a call.

Note: Situations to prevent quick scaling, up and down, result in addition and removal request to pods. This sitiatuion, where a pod is created and by the time, it is expected to serve, get a signal to get killed, is known as Thrashing of Pods. This situation of Thrashing arises when the workload on your application fluctuates a lot. When the load increase, an increase in infra is requested (thanks to the strategy configured by you), but till the time infra is added, the load decreases. This results in pods getting added to the Kubernetes services and then being removed. This up and down in the number of pods is known as Pod thrashing. Your scale strategy should consider such situations and decide upon the right delays in scaling.

Introduction to auto scaling

We discussed situations where you can scale your applications deployed on Kubernetes up and down . To accomplish scaling up and down, there are two ways

to do it. One of them is manual: every time your application starts breaching SLA (unable to finish processing in the expected time), you can log in to your Kubernetes cluster and increase the number of pod replicas. It is easier said than done, as modern workloads do not show a consistent upward or downward trend in their hunger for resources. Another strategy to scale up/down is to monitor your workloads for a few days/weeks, predict the number of pods needed at any given point in the day, and have a CRON job that accomplishes scaling up and down for you. This analysis is based on the infrastructure need that has arisen for delivering services to end users. The scale up/down configuration is fixed (hardcoded), and any change will need somebody to analyze the workload manually and redo configurations. This strategy to scale up/down is referred to as predictive scaling.

While predictive scaling does better when compared to manual scaling, it does not capture the situation on the ground. It can identify and scale those major workload spikes, let us say at the hour level, but not the granular ones. For the right level of solution, the best would be to somehow keep monitoring your system for needs of scale (maybe, every 1 minute), and scale up and down accordingly. You may wonder why 1 minute; it is because you can set a time lesser than that, but reacting to such granular spikes in work might result in infrastructure bashing. It takes some time to scale up, to create a pod, and to add a node, and if your autoscaling has triggered creation, till the time infrastructure is ready, the need for infrastructure has reduced. This could result in unnecessary scale up/down operations happening.

Kubernetes supports scaling up/down of infrastructure out of the box, taking care of all the pitfalls of predictive scaling described above. It has a well-supported mechanism with sufficient capability to extend autoscaling aspects as per the need of applications. Kubernetes Autoscalar algorithm keeps monitoring the system for a metric value, for example, the number of HTTP requests coming, and it scales up and down based on it in near real time.

Autoscaling not only helps the application scale up/down in near real time, resulting in better serving without breaching SLAs, but it also prevents the application from acquiring unnecessary infrastructure. The freed infra could result in lower costs for the pay-as-yougo model. This facilitates justified and optimal use of infrastructure as well which is key incase of on premise deployments, where you cannot add more infra by a click of button.

Types of scaling in K8s

In this section, we will talk about the types of scaling Kubernetes provides and try to understand the benefits of one over the other with the help of a few real-world use cases.

Horizontal pod scaling

In horizontal pod scaling, the number of pods running for the application is increased or decreased to scale up/down. For example, say we want to scale up/down based on the number of incoming requests to a service, as shown in *Figure 10.2:*

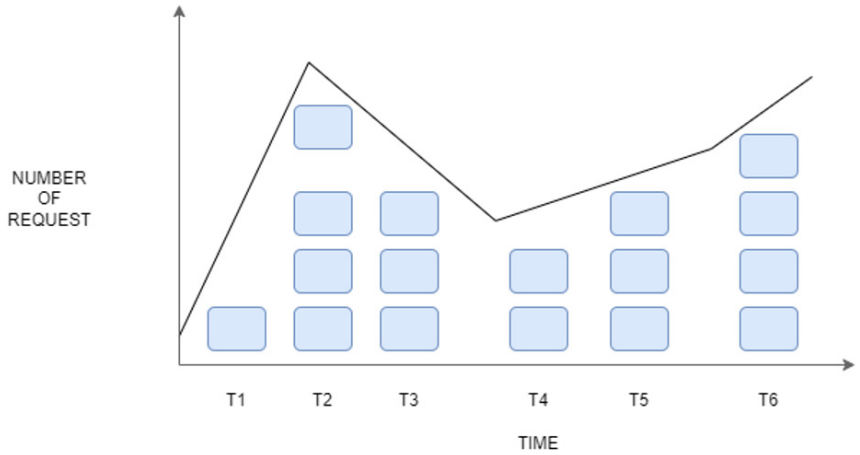


***Figure 10.2:** Horizontal Pod Scaling*

In the preceding figure, the x-axis represents the timestamp at which the infrastructure needs were monitored (represented by T1, T2….T6), and the Y-axis represents the number of incoming HTTP requests; the boxes represent the pods. As you can see, the moment that the number of requests increases, the number of pods to serve them also increases; and as they decrease, the number of pods to serve them decreases. Pay attention to the size of the pods; it remains the same.

Horizontal pods scaling can only scale objects that can be scaled inside Kubernetes, such as Pods, stateful sets, and similar resources. It cannot scale non-scalable resources like daemon sets.

`HoriziontalPodScalar` is implemented in Kubernetes as Kubernetes API and controller. The pod scaling controller runs in the Kubernetes control plane and adjusts the scale of the Kubernetes object (pod, stateful set) periodically, based on values of metrics like average memory and average CPU. You can define custom metrics as well and can scale based on the value of a custom metric.

Let us take a look at an example of how Horizontal scaling could be applied. For this example, it is recommended to have at least a couple of nodes apart from the control plane node running in your Kubernetes cluster.

You can apply autoscaling to any of the deployments we have created in the past. One such service created in *Chapter 3, HTTP Load Balancing and Ingress*, is **`my-appservice`**:

```
kubectl autoscale deployment my-app-service --cpu-percent=70 --min=1 --
max=10
```

The preceding command will set up an **Horizontal Pod Autoscaling** (**HPA**) configuration, which will check the current state of CPU usage for the pods aligned with the service, and a pod will be added the moment the average CPU utilization goes above 70%. Similarly, when the CPU utilization decreases, the pod will be evicted. With this configuration, the maximum number of pods to which an application scales up is 10, and minimum number of pods an application can scale down to is 1.

You can define HPA in the form of YAML as well. The preceding command can be defined in YAML as follows:

```
1.          apiVersion: autoscaling/v2
2.          kind: HorizontalPodAutoscaler
3.          metadata:
4.            name: my-app-service-hpa
5.            namespace: default
6.          spec:
7.          scaleTargetRef:
8.            apiVersion: apps/v1
9.            kind: Deployment
10.           name: my-app-service
11.         minReplicas: 1
12.         maxReplicas: 10
13.         metrics:
14.         - type: Resource
15.           resource:
16.             name: cpu
17.             target:
18.               type: Utilization
19.               averageUtilization: 70
```

In the preceding YAML, the line number shows the name of the HPA object created in the **`default`** namespace. **`scaleTargetRef`** (line numbers 7-10) defines the Kubernetes object on which the HPA is applied. Lines 11 and 12 define the minimum and maximum number of pods. Lines 13 to 19 configure the average CPU utilization

of above 70%, which is the criteria for scaling up and down. You can apply the preceding file using the **`kubectl create -f <filename>`** command.

The preceding example defines scaling based on one metric: average CPU utilization. You can define autoscaling based on multiple metrics as well, by giving multiple metrics (lines 14 to 19):

```
1.  (…)
2.  metrics:
3.  - type: Resource
4.    resource:
5.      name: cpu
6.      target:
7.        type: Utilization
8.        averageUtilization: 70
9.  - type: Resource
10.   resource:
11.     name: memory
12.     target:
13.       type: AverageValue
14.       averageValue: 500Mi
15. - type: Object
16.   object:
17.       metric:
18.         name: requests-per-second
19.       describedObject:
20.         apiVersion: networking.k8s.io/v1beta1
21.         kind: Ingress
22.         name: main-route
23.       target:
24.         type: Value
25.         value: 10k
```

For complete YAML, refer to the **`hpa-multiple-metrics.yaml`** file in the codebase provided with the chapter. Line numbers 15 to 25 define HPA based on an object.

This object is an Ingress object with the name main-route, and it adds a pod when the average number of requests goes above 10k. In addition to this, you can customize the scale in up and down behavior. You will explore that customization aspect, but before diving into the concepts, you need to understand the HPA algorithm that runs behind the scenes.

Kubernetes implements HPA as a control loop, meaning the loop execution happens intermittently rather than as a continuous process. By default, the interval is set to 15 seconds. However, you can configure it by setting the value of the **`--horizontal-pod-autoscaler-sync-period`** parameter to the kube controller manager.

With the time configured to **`--horizontal-pod-autoscaler-sync-period`**, the controller manager will run a query on the resource mentioned in HPA once in the interval. The controller manager will identify the target resource mentioned as **`scaleTargetRef`**, select a pod based on the target resource's selector labels, and get the metric from resource metrics API or custom metrics API. One more component that we have not talked about is the Metrics server. The metric server is launched separately and provides metrics for aggregated APIs **`metrics.k8s.io`**, **`custom.metrics.k8s.io`**, or **`external.metrics.k8s.io`**. **`HoriziontalPodScalar`** fetches these metrics and uses them for scale. The Kubernetes Metrics Server is an aggregator of resource usage data in your cluster and can be started with the following commands:

```
kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/
releases/latest/download/components.yaml
```

Take a look at *Figure 10.3* to understand how everything is stitched end-to-end:

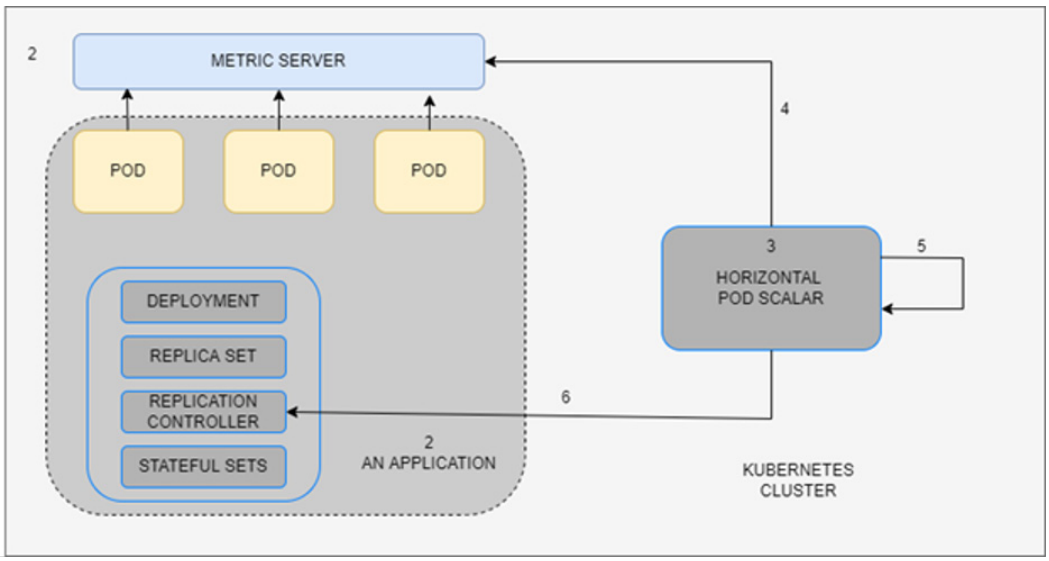


Figure 10.3: *HPA*

Consider the figure's numerical labeling with the following numerically labeled bullet points:

1. **Metric server**: Metric server accumulates resource usage data across the whole cluster. It collects data from kubectl (running on each node) and makes it available in the API server via the K8s Metric APIs.
2. **Kubernetes application**: This is an application running in Kubernetes, for example, the **`hello-java`** application. Each application has objects specified by the configuration:
    a. **Deployment:** A Deployment represents an application running where we give spec to specify the complete details. For example, the image to be used for the deployment should be **`hello-java:v1`** in the container registry.
    b. **Replica set:** A ReplicaSet aims to maintain a stable set of replica pods at any given time. For instance, if a pod goes down due to node failure, the replica set identifies that a new pod is needed to maintain the current pod replica requirements and starts the process of creating one on another suitable node.
    c. **Replication controller:** As the name suggests, it controls the number of replicas at any given time for deployment. It takes the final decision to increase and decrease a pod.
    d. **Stateful sets:** This object maintains the stateful applications. It manages the scaling and deployment of pods and provides ordering and uniqueness guarantees.
3. **Horizontal pod scalar:** The horizontal Pod scalar is an object that assesses the metrics in a Kubernetes cluster and informs the replica set to scale up or down.
4. Horizontal Pod Scalar scans the metrics server for reading the metrics for which it is configured. This happens by default after every 15 seconds.
5. The horizontal Pod scalar calculates the pods needed based on the metric defined for autoscaling.
6. Horizontal Pod scalar asks Replica Controller to create or delete the number of pods as per the latest calculations on current metrics.

Metric threshold definition

You can define the threshold for a metric in two ways:

- **Absolute/raw values:** You can specify absolute values as thresholds, for example, the threshold for CPU utilization can be set to 4 CPUs/mCPUs.

- **Percentage value:** You can specify the threshold in percentages as well, for example, CPU utilization above 80%, scale up.

If you specify an absolute value for metric (either CPU or memory), the value is used. If you specify a percentage value for metric (CPU or memory), the **`HorizontalPodAutoscaler`** calculates the average utilization value against the percentage of pods, CPU or memory utilization. This strategy holds for custom and external metrics as well.

With this, you are left with just one more concept to study: how to customize the behavior of **`HorizontalPodScalar`**. You can add the following sample configuration in the HPA YAML file to define the scalar behavior:

```
1.          (...)
2.          minReplicas: 2
3.          maxReplicas: 15
4.          metrics:
5.            (...)
6.          behavior:
7.            scaleDown:
8.              stabilizationWindowSeconds: 300
9.              policies:
10.     - type: Percent
11.         value: 10
12.         periodSeconds: 60
13.   scaleUp:
14.     stabilizationWindowSeconds: 0
15.     policies:
16.     - type: Percent
17.        value: 100
18.        periodSeconds: 15
19.        - type: Pods
20.          value: 4
21.          periodSeconds: 15
22.         selectPolicy: Max
```

In the preceding configuration (lines 6 to 22), you can define the scale-up and down behavior for HoriziontalPodScalar. **`stabilizationWindowSeconds`** (lines 8 and 14) is used to configure a time by which the scale up/down behavior will be delayed. This is particularly useful when there is a fluctuating workload that intends to increase and decrease the infrastructure too often. While scaling down, the value for this property is set as 300, meaning that the actual scale down will start after 300 seconds of the trigger. In the preceding configurations, you can see one policy related to scaling down the application and two policies related to scaling up.

According to scale-down policies, when Autoscalar in its periodic execution of the control loop identifies a need to scale down, it decreases the number of pods by 10 percent in the first 60 seconds. For example, if the number of pods is 100 and the need is to remove 50 pods, all will not be removed. Rather, only 10 of them will be removed in the first 60 seconds.

The scale-up policy has two policies. According to the first one, you should increase the number of pods running every 15 seconds till the maximum limit is calculated by Autoscalar. According to the second, you should increase the number of pods by 4 every 15 seconds. Among the two policies, the policy that results in the maximum value of pods to be added will be selected; in this example, policy #1 will be chosen, which will have a higher impact of having 15 pods in 15 seconds.

Limitations of HPA

Though HPA looks to be a silver bullet for handling all the scaling needs of the application, there are a few limitations associated with it:

- An application wanting to leverage HPA needs to be created in a way that supports distributed work models. It might need rework with your applications.
- HPA does not handle spikes in workloads, resulting in demand for scale instantly. It takes time to add a new pod or a new node to the cluster.
- HPA is bound by the overall cluster size, so the number of pods is limited. However, you can increase the size of the cluster via Cluster Autoscalar, but it takes more time to add a node to cluster.

Vertical pod scaling

Vertical pod scaling provides dynamic addition or removal of compute and memory resources to a pod, which belongs to a service. **Vertical Pod Autoscalar** (**VPA**) is a Kubernetes component that supports all types of deployments and works by tweaking the resource request parameter of the pod based on metrics collected from workloads. You have seen a YAML file with the request parameters mentioned. Following is the snippet of a deployment YAML that talks about resources:

```
1. (...)
2. spec:
3.   containers:
4.   - name: app
5.     (...)
6.     resources:
7.       requests:
8.         memory: "64Mi"
9.         cpu: "250m"
10.      limits:
11.        memory: "128Mi"
12.        cpu: "500m"
```

If you look at lines 6 to 12, you will see the specifications of requests and limits. The request section (lines 7 to 9) specifies the chunk in which the resources are going to be asked and allocated to a pod. The limit (10 to 12) determines the maximum resources that a pod can take. Kubernetes utilizes kernel throttling to impose CPU limits, that is, if the application requests for resources above what is configured, it gets throttled. Memory requests and limits, on the other hand, are implemented differently, and it is easier to detect them. You only need to check whether your pod's last restart status is **`OOMKilled`**.

The specification for request here is based on some initial guesstimate. It is very hard to come up with a good request specification and, more often than not, a developer misses it by a margin. VPA, when applied, takes into consideration the historical and current resource consumption by a pod and applies the analysis to determine the requests for a pod at runtime.

Consider the following YAML file that demonstrates the VPA:

```
1.   apiVersion: autoscaling.k8s.io/v1
2.   kind: VerticalPodAutoscaler
3.   metadata:
4.     name: my-app-service-vpa
5.   spec:
6.     targetRef:
7.       apiVersion: "apps/v1"
8.       kind: Deployment
```

```
9.        name: my-app-service
10.    resourcePolicy:
11.      containerPolicies:
12.        - containerName: '*'
13.          controlledResources:
14.            - cpu
15.            - memory
16.          maxAllowed:
17.            cpu: 1
18.            memory: 500Mi
19.          minAllowed:
20.            cpu: 100m
21.            memory: 50Mi
22.    updatePolicy:
23.      updateMode: "Auto"
```

Lines 10 -23 describe a policy that can be applied to requests specification of all the containers (Line 10) involved in the service **`my-app-service`** (line 9). This resource policy (10-23) applies to both CPU and memory (14-15) and defines the max allowed range for each request (16-18) as well as the **`minAllowed`** range for requests (19-21).

The **`targetRef`** specifies the deployment on which this VPA configuration is applied. The last section Update policy configuration, states how will these configurations will be applied to the workload. A value of **`OFF`** for **`updateMode`** results in the calculation of requests happening but not getting applied at run time. A value of Auto for **`updateMode`** results in applying these configurations right away.

VPA comprises three components:

- **Recommender**: This component takes into consideration the historical and latest metrics, and based on the observation, recommends value to be configured for new workload resource requests. This is done as per the VPA policy and the limits set for the workload.
- **Updater**: The recommendations from Recommender are applied to the workload, based on the Update policy in the VPA YAML. If the update is set to **`ON`**, the updater will evict pods and create new ones. If you have configured the pod disruption budget, the updater makes changes without disrupting the budget.

PodDisruptionBudget is a configuration that could be defined as a YAML file and applied using the **kubectl apply** command, and it ensures that a certain number or percentage of pods are not voluntarily evicted from the node at any given point in time. Voluntarily simply means that the eviction is delayed. You need to specify three things: the label selector, the **minAvailable** field and the **maxUnavailable** field. This will ensure that pods labeled with the label selector are going to maintain **minAvailable** number of pods and can afford to have **maxUnavailable** number of pods unavailable. Following is a snippet showing how you can mention budgets:

```
1. apiVersion: policy/v1
2. kind: PodDisruptionBudget
3. metadata:
4.   name: my-app-service-budget
5. spec:
6.   minAvailable: 2
7.   selector:
8.     matchLabels:
9.       app: my-app-service
```

In the preceding configuration, pods with label selector as an app, i.e., **my-app-service**, will at least maintain 2 pods. For example, assume that you have 4 pods running, and the VPA updater tries to update the resource requests. Two pods will be recreated first, and once the two pods are available, the leftover pods are affected.

Refer to the **maxunavailable-example.yaml** file, which defines the **maxUnavailable** budget for deployment.

- **VPA admission controller:** This component ensures that the new pods that are created have the newly proposed resource requests annotated in the pod definitions.

Cluster autoscaling

Cluster autoscaling, as the name suggests, is a mechanism to grow the size of your cluster when there is a resource crunch observed on the Kubernetes cluster. When you see pods not getting scheduled due to resource shortages, it indicates that more pods need to be created, based on scaling policies. However, there is no more space left to host new pods. Take a look at *Figure 10.4*, which explains the concept in detail:

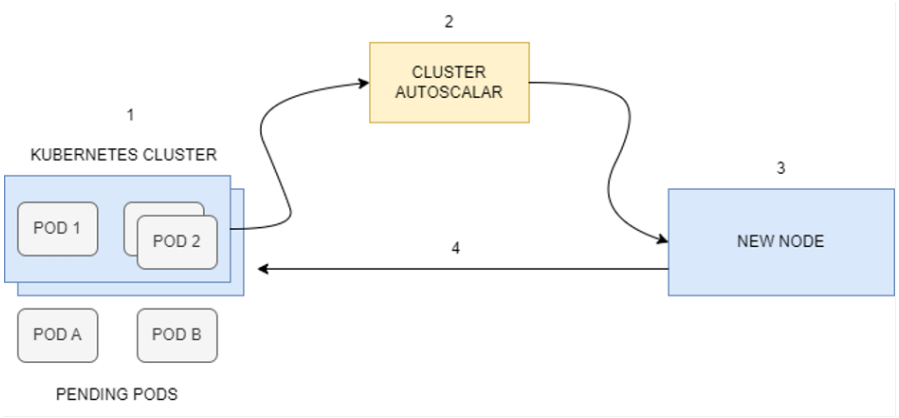


Figure 10.4: *Cluster Autoscalar*

Refer to the numerical labeling in *Figure 10.4* with the corresponding numerical explanations:

1. It is the Kubernetes cluster on which we have multiple nodes who host **POD1**, **POD2** and **POD 3**.On the other hand, **POD A** and **POD B** are scheduled on the cluster but are pending because of a resource crunch.
2. Cluster Autoscalar checks and finds that there are pods in the pending state; it starts the process of adding more nodes. It also ensures that it is not violating the maximum number of configurable nodes for a cluster policy.
3. A new node is provisioned. This newly provisioned node must do some pre-defined installations.
4. The newly created node is added to the cluster and pending pods eventually get scheduled on the new node.

When it comes to Kubernetes as managed services on the cloud, it is a straightforward process, and all public cloud providers have a well-defined mechanism in place. Generally, it is confined to setting a Boolean flag, which enables scaling, and setting up some basic properties.

Following is the list of clouds and their official documentation that talks about Cluster Autoscalar:

- Google Cloud Platform: **https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler**
- AWS: **https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/aws/README.md**
- Azure: **https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/azure/README.md**

However, when it comes to setting up cluster Autoscalar on-premises, the process is a little cumbersome. To add the support of Autoscalar to your cloud, you need to take the following steps:

1. You will have to write an implementation for the **`CloudProvider`** interface (in the Go Programming language). Link to interface: **https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/cloud_provider.go**
2. Add it to the builder: **https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/cloudprovider/builder/cloud_provider_builder.go**
3. Build an image (custom) of **`ClusterAutoscalar`** with all the preceding changes and set it up to start with your cluster APIs.

There are also a few prerequisites that need to be taken care of before the preceding logic kicks in. The important ones are as follows:

- Autoscalar assumes that all nodes within the same group are of the same machine type, have the same labels and taints, and are in the same availability zone.
- There must be a way to delete a specific node. If your cloud supports instance groups, and you are only able to provide a method to decrease the size of a given group, without guaranteeing which instance will be killed, it will not work well.
- There is a need to handle the mapping of Kubernetes to the actual instance it is running on.

As you can see, there are multiple development, deployment, and maintenance-related activities involved in hosting **`ClusterAutoscalar`** for your on-premises cloud. The community understands the pain point and hence, there are tools available in the industry. One such tool is Metal3.

Metal3 helps manage your bare metal infrastructure utilizing the Kubernetes APIs using Metal3 and machine-API CRDs. It is designed to work in a Kubernetes cluster and hence no special deployment is needed. It is also integrated with the Machine API from SIG Cluster-Lifecycle that allows your bare metal physical machines to manage their growth, like your cloud-based clusters.

Standard metric scaling

By standard metric, it is meant that these metrics are already available out of the box in your standard Kubernetes environment and its supporting infrastructure. In

standard metrics as well, you will observe two kinds of metrics: K8s-specific metrics and external metrics.

You can define scale-up conditions when the average CPU consumption inside your pods for a service goes above a threshold defined, for example, above 80%. Average CPU utilization is a metric provided by Kubernetes out of the box and is hence, known as K8S specific standard metric. As discussed, and demonstrated in the *horizontal pod scaling* section, you can either define scale up/down on one metric or a collection of metrics.

Sometimes, there are situations when you want to define scale-up conditions for your applications using a metric outside the cluster. For example, if your application is listening and reacting to data arriving in a queue, then you might want to scale up/down the number of pods based on the number of unread messages. In addition to the number of messages, you might want to track the lag (the time between a message being placed on the queue and it being consumed by the consumer) and scale the number of pods based on the metric. The higher the lag, the more is the infra needed to consume messages quickly. Another very common situation could be that you want scaling to happen based on the number of incoming HTTP requests. Both the number of HTTP requests and the number of unread messages are metrics that are not related to Kubernetes.

Metrics are exposed via the **`metrics.k8s.io/v1beta1`** API resource.

Following are the commands to fetch the metrics:

```
# Get the metrics for all nodes
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes

# Get the metrics for all pods
kubectl get --raw /apis/metrics.k8s.io/v1beta1/pods
```

The second one is to obtain specific resource:

```
# Get the metrics for all nodes
kubectl get --raw /apis/metrics.k8s.io/v1beta1/nodes/NODE_NAME

# Get the metrics for all pods
kubectl get --raw /apis/metrics.k8s.io/v1beta1/namespaces/
NAMESPACE/pods/POD_NAME
```

You have already seen how to install Prometheus and push the logs and metrics to Prometheus servers. You have also seen how you can use Grafana dashboard to visualize the metrics. However, for Kubernetes pods to use these metrics, you

will have to make sure these metrics in Prometheus are available to the metric server running as pod in Kubernetes. The metrics pushed in Prometheus can be made available to Kubernetes pods using the Prometheus adapter. For this example, assume that the log is already being pushed to Prometheus. The first step will be to define **`promethous-adapter.yaml`**, as follows:

```
1. logLevel: 4
2. prometheus:
3.   url: http://prometheus-server.monitoring.svc.cluster.local
4.   port: 80
5. rules:
6.   external:
7.   - seriesQuery: '{__name__=~"^kafka_consumergroup_group_lag"}'
8.     resources:
9.       template: <<.Resource>>    name:
10.      matches: ""
11.      as: "kafka_lag_external_metrics"
12.    metricsQuery: 'avg by (topic) (round(avg_over_time(<<.
   Series>>[1m])))'
```

It is expected that the community helm repo for Prometheus is already added, as done in the previous chapter. Apply the preceding configurations using the following command:

```
helm -n monitoring install prometheus-adapter prometheus-community/
prometheus-adapter -f promethous-adapter.yaml
```

The preceding configurations perform a series query (line 7 to 11) on Prometheus and exposes the metric as computed by the metric query (Line 12). The results are exposed as metric named **`kafka_lag_external_metrics`** (line 11).

Now you can go ahead and use this metric to scale an application, using a HPA configuration as follows:

```
1. apiVersion: autoscaling/v2beta1
2. kind: HorizontalPodAutoscaler
3. metadata:
4.   name: kafka-hpa-demonstaration
5. spec:
```

```
6.   scaleTargetRef:
7.     apiVersion: apps/v1
8.     kind: Deployment
9.     name: my-app-service
10.  minReplicas: 1
11.  maxReplicas: 5
12.  metrics:
13.  - type: External
14.    external:
15.      metricName: kafka_lag_external_metrics
16.      targetValue: 2
```

You can apply the preceding HPA configuration, and the deployment **`my-app-service`** (line 9) will scale up with target value 2.

Custom Metric scaling

Scaling based on custom metrics is like external metric. Custom metrics means that you could create your own metrics and define scaling based on it. Assume that you have an application that exposes metrics using **`/metrics`**. First and foremost, you have to let Prometheus know that this endpoint can be scraped by the following annotations on the pod:

```
1.  apiVersion: apps/v1
2.  kind: Deployment
3.  metadata:
4.    (...)
5.  spec:
6.    (...)
7.    template:
8.      metadata:
9.        annotations:
10.         prometheus.io/path: "/metrics"
11.         prometheus.io/scrape: "true"
12.         prometheus.io/port: "8000"
```

```
      (...)
    spec:
      (...)
```

The application deployed in the preceding code publishes a metric **`success_requests`**, which we wish to average every 5 minutes and then use this rate for HPA. Then, you must define the following **`promethous-adapter.yaml`**:

```
logLevel: 4
prometheus:
  url: http://prometheus-server.monitoring.svc.cluster.local
  port: 80
rules:
  custom:
  - seriesQuery: "success_requests"
    resources:
      overrides:
        kubernetes_namespace:
          resource: metricsdemo
        kubernetes_pod_name:
          resource: pod
    name:
      matches: "^(.*)"
      as: "success_request_metric"
    metricsQuery: "sum(rate(<<.Series>>[5m])) by (<<.GroupBy>>)"
```

Now you can go ahead and define an HPA object like the preceding configuration:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: custom-hpa-demonstaration
spec:
  scaleTargetRef:
    apiVersion: apps/v1
```

```
8.      kind: Deployment
9.      name: my-app-service
10.   minReplicas: 1
11.   maxReplicas: 5
12.   metrics:
13.   - type: Pod
14.     pods:
15.         metricName: success_request_metric
16.         targetAverageValue: 4
```

The preceding HPA configuration defines a scale up strategy. When the average value of **`success_request`** over 5 minutes goes above 4, a new pod is added.

Best practices of scaling

In this last section, you will learn about a few best practices for scaling and the key considerations to be made while scaling a microservice application effectively:

- **Node pool configurations**: It is important to mention the node selector attribute in your pod definition files so that the pods with special needs of infra could be allocated to the right node pools. First and foremost, there could be applications that need big pods; for example, if you are parsing a big file, you need big pods, and the size of the pod you want to have could not be created in other node pools due to the underlying size of the machines. Another situation is the special needs of the application; for example, if your application needs GPU to be used, a node pool enabled with GPU is the way forward. In short, the key is to understand whether there is any special need for scaling for your applications, identify/create a node pool where the special need could be met, and finally use **`nodeSelector`** in pod definition to leverage the node pool.

- **Ingress and Egress practices**: It is always recommended to understand the ingress and egress needs of the application to define the upper limits for scaling not just an application/micro service but the complete cluster. It is common to use the number of HTTP requests as a parameter to scale your microservices. Similarly, there is a need for your microservice to respond within a time limit. While on one hand, you can make your application scale aggressively, this always goes in conjunction with the capabilities of your configured ingress and egress.

- **Load balancing**: It is simple to scale up pods in microservices, but the key is to start utilizing them as soon as possible. Moreover, to transfer the load to the new pods, it is important to ensure that the load balancer knows about it and can route a request to the newly added pods. Generally, it is important to ensure that your network policies and load balancing configurations work in unison to leverage the power of scaling.
- **Storage**: Storage infrastructure should be configured and planned in a way that it is capable enough to cater to the needs of microservices when scaled to the maximum. Generally, it is advisable to use a storage option per microservice to avoid the impact of scaling one microservice from impacting the needs of other microservice. If you look back at *Figure 10.2*, each microservice discussed has its own back end in place. This kind of setup will not result in a noisy neighborhood, meaning a certain microservice scaled 10 times will not impact the IOPS (input/output operations per second) of other microservices.

Conclusion

Kubernetes provides a very mature framework for scaling up and down microservices automatically. You can configure a scaling strategy to increase the number of pods (horizontal scaling) as well as increase the size of individual pods (vertical scaling). Kubernetes supports cluster autoscaling as well. Scaling in Kubernetes happens based on indicators (metrics) and their associated values going above the threshold. These metrics could be the system metrics available out of the box from Kubernetes or external (standard metrics outside Kubernetes cluster), or they could be custom metrics created by applications. In the case of custom metrics, the metrics are pushed to Prometheus setup, and then you can define queries and finally use the data to define the threshold for scaling your application.

Points to remember

- Kubernetes supports scaling based on standard metrics like CPU and memory usage inside a pod.
- Kubernetes supports scaling based on external metrics like the number of unread messages in a queue.
- Kubernetes supports scaling based on custom metrics as well.
- You can scale up/down a Kubernetes deployment by adding and removing pods; this is known as horizontal pod scaling.
- You can also configure vertical pod scaling, kill a small pod and create a new pod with a bigger set of resources.

- To ensure that the system tackles the need for increasing infrastructure, Kubernetes supports Cluster Autoscalar, which sees the number of pending pods (due to resource crunch) and adds more nodes to the Cluster.

Questions

1. You can configure HPA on multiple metric thresholds. Is this true or false?
2. Kubernetes support scaling based on which of the following?
    a. Standard metrics
    b. External metrics
    c. Custom metrics
    d. All the above
3. Why is it recommended to have different datastores running behind each microservice from a scaling perspective?

Answers

1. True
2. d
3. Ease of scale of microservice and no impact on other microservices

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 11
Introduction to Service Mesh and Istio

Introduction

Using microservices for building web and desktop applications has become a norm these days. One of the benefits microservices provide is the range of programming language and technology choices across independent service teams, and many such teams also deliver a software, mostly software as a service.

However, as the applications grow, the number of microservices within the application also increases. Despite following the industry standard and/or proven design patterns for your services, things start becoming complex to understand and tougher to manage. Teams start finding it hard to manage the balance between the functional and non-functional requirements. If you are spending more time and efforts on traffic management, security, and observability of your services than on adding new features to your application to make it competing in the market, then maybe it is time to start looking at Service Mesh. The Service Mesh helps take the load off the development team for numerous non-functional activities.

You can consider Service Mesh as an infrastructure built to understand the existing network between microservices. Service Mesh does this by injecting a proxy into every network path so that cognizance about the traffic in the network is captured by all proxies and brought under a centralized control. If you are already using microservices in your organization, knowing about service mesh will definitely help you in discussions related to application roadmap and architecture evolution.

Structure

In this chapter, we will discuss the following topics:

- Why do you need a Service Mesh?
- What is a Service Mesh?
- What is Istio?
- Istio architecture
    - Data plane
    - Control plane
- Installing Istio
    - Installation using istioctl
- Cost of using a Service Mesh
    - Data plane performance and resource consumption
    - Control plane performance and resource consumption
- Customizing the Istio setup

Objectives

In this chapter, we are going to understand if and when a software development team should start looking at Service Mesh. We will discuss the benefits that Service Mesh brings and also the costs associated with it. We will get familiarised with Istio Service Mesh in this chapter and look at the setup of the Istio in the context of a Kubernetes cluster as well as outside of a Kubernetes cluster. After completing this chapter, you should have a fair idea about the concept of Service Mesh and the basics of Istio.

Why do you need a Service Mesh?

Let us begin with a fact: **not every** microservice-based application needs a service mesh. That is to say, yes Service Mesh is great, but it has its downsides, and we will look at them in this chapter. Readers are expected to understand the pros and cons of the Service Mesh before taking deciding whether they should use the Service Mesh.

That said, let us now understand why service meshes are getting more and more popular these days. Modern applications are architected as distributed systems, where each microservice or service has a fixed responsibility in the larger scheme of things. These services are part of a family. And just like members of any functional-

dysfunctional family, they need to communicate with each other, sometimes in normal voice and sometimes in a loud voice. This communication makes things interesting, and different scenarios emerge when the members, read as services, in the system grow. With a growing number of services, managing, observing, and securing the communication between the services to make the application production-grade requires as much effort as it takes to keep adding new features in your application.

Consider *Figure 11.1,* where an application is made up of 5 services and where each service communicates with different number of services to function. There is also a database to store the state required for the application. Let us consider that these services are deployed in a single Kubernetes cluster.

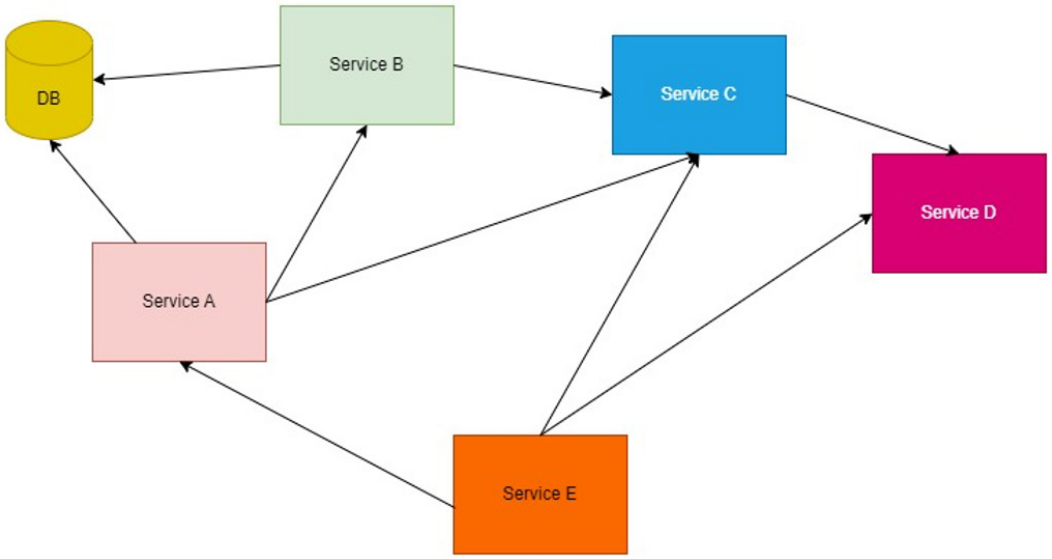


Figure 11.1: *A complex application*

In real-life applications, there can be a messaging system here in the figure as well, where services communicate with each other asynchronously with the help of messages. However, it is not put here for the purpose of simplicity.

As the deployment for this application grows, let us say in the number of instances per service or just the number of services in the application, it becomes harder to understand and manage because you cannot get the whole view of the system easily. You start facing problems related to service discovery, load balancing the traffic between services, graceful recovery from failures of services and how to monitor the service communication and collect metrics.

Let us quickly touch upon these problems one by one.

Service discovery

In an autoscaled cluster, the number of replicas for a service keep changing based on the criteria set. Similarly, it is quite common to add new services to provide new features in the application and sometimes remove the services that are providing deprecated functionalities. For the existing services in the cluster, it is necessary to know what services they need to talk to and how to find their IP addresses, endpoints and so on. This is where the service discovery algorithm is required.

Load balancing the traffic

Load balancing is about balancing the incoming requests among the available servers or replicas of the application. The advanced flavors of this could be that you would want more fine-grained control over the traffic. You would want to direct, let us say 70 percentage of traffic, to a new version of a service that is also known as A/B testing. Another scenario could be applying a different load balancing policy to the traffic for a particular set of services like round-robin, least-connection, resource-based and so on.

Monitoring the traffic between services

In modern systems, it is necessary to have knowledge of which service is calling which service and how often it is doing so. This helps in identifying areas for performance improvements and gaps in the system design, and it also gives a clear picture of how the system functions.

Collecting metrics

Collecting metrics is a foundational requirement when you want to adopt SRE practices and become more and more operationally ready. You need to build indicators first about how the system is functioning currently and then start setting customer expectations based on those indicators. At the same time, you need to gather insights from these metrics to improve the overall correctness and availability of the system.

Recovering from failure

Graceful recovery from failure is an important part of building applications with good user experience. In a growing application, a service may be getting called from multiple services within the same cluster, so failure of one service may have cascading effects. You also need to put system design practices like rate limiting and circuit breaker to avoid resource starvation and make the services more resilient.

So, when you start handling these problems in your services with the code that you manage, the picture starts looking like *Figure 11.2*:

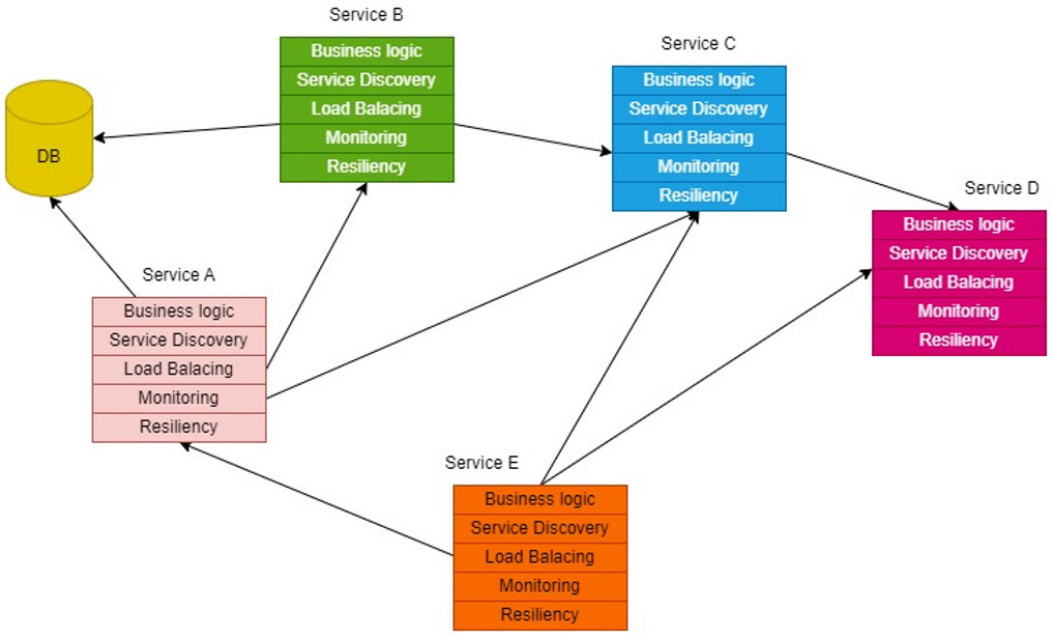


Figure 11.2: *A very complex application*

Not a very good picture, right? This is where Service Mesh comes in; it helps simplify the picture and reduce the complexity while easing the strain on development teams.

What is a Service Mesh?

First things first, we will be exploring service mesh in the context of containers and microservices and Kubernetes. However, Service Mesh as a concept exists beyond these and is independent from them. It is about communication between services and is not limited to Kubernetes deployed services. It is an architecture pattern to effectively manage the communication between services. It is not specific to any cloud, nor is it a microservices pattern.

Figure 11.3 shows the same set of services as in the preceding example, after a service mesh like Istio is installed with the services in the Kubernetes cluster:

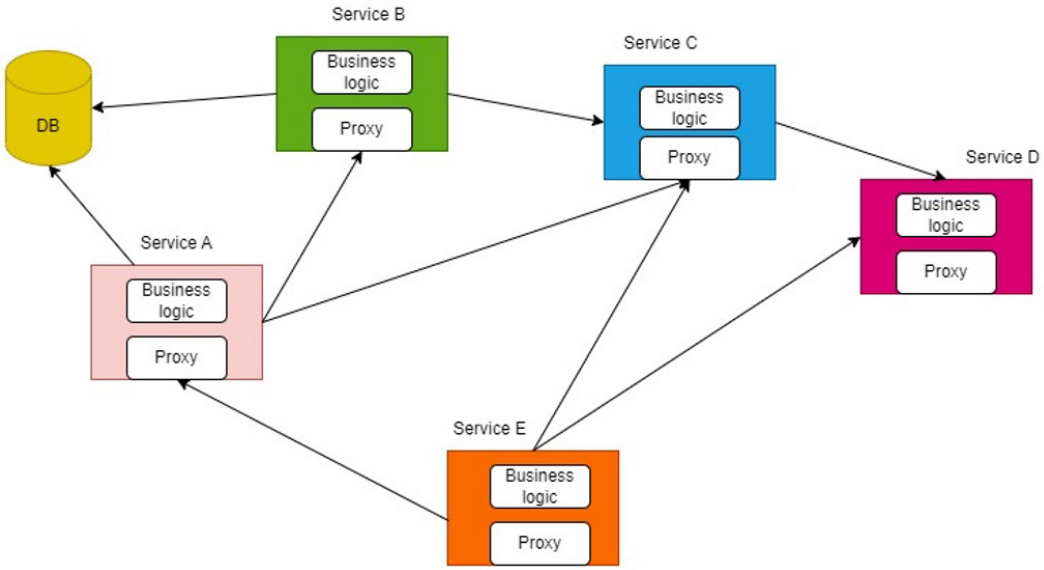


Figure 11.3: *A complex application with Service Mesh*

If you are getting to know the idea of the service mesh for the first time, you might wonder why is this extra complexity needed? Adding services mesh into the scheme of things adds new machinery plus the latency. Who would want that? There is also the fear of maintenance of newly added proxies, control planes and so on. All your thoughts are valid. Having said that, Service Mesh is designed in a way that adding new logic to the system becomes a lot easier. Moreover, the operational cost related to deployment and maintenance is not very high once you use the latest and recommended ways to do it.

Service mesh is like an infrastructure made up of small tracking devices kept in the pocket of each service. You must have heard of fitness tracker devices that help monitor movements, heartbeat, and activities like running and walking; you can consider proxy as a tracking device for your services. The proxy container tracks all the communication by your application container.

Now with the proxy injected alongside every service, all the incoming and outgoing communications from a service go via proxy. All these proxies are connected to each other via a Control plane component of service mesh, which is not shown here. However, this should reveal most of the magic done by the service mesh. Now that all the communication is *listened* by the service mesh components, it gets a lot of cognizance about the system as a whole and hence, it can provide rich features to your application. Thus, the cross-cutting concerns listed in the preceding image, such

as service discovery, load balancing, monitoring, resiliency and so on, are taken care of by the proxy, and business logic stays inside your application container, making it more fit-for-purpose.

In other words, a service mesh is an additional infrastructure layer that you can add to your application, and it provides a set of features. You get capabilities like observability, traffic management, and security, for which you would otherwise have to write code and maintain that on your own.

The service mesh architecture is a next step for the world of containers and orchestration. It lets you simplify your app development and configuration by connecting all the components of your application. The control plane component of the service mesh helps you tackle concerns like managing network traffic, and dealing with temporary faults and security. We will learn more about control plane in a while.

Popular implementations of service mesh, such as Istio and Linkerd, implement the service mesh by providing a proxy instance, called a sidecar, for each service instance. Cross-cutting concerns that are not tied with business logic are abstracted by the sidecars, and it is usually left to the DevOps teams or the Operations teams to manage the service mesh and get the best out of it. While the Service Mesh chapters in this book revolve around Istio, you can refer to the official documentation of Linkerd on **https://linkerd.io** to know more.

All the problems that we discussed, such as Service discovery, load balancing, monitoring, metrics collection and failure recovery, are handled with the service mesh. For example, a service mesh detects services as they come up and removes them from the system gracefully when they disappear, aiding Service discovery. This also helps improve resiliency in servicetoservice communication. A service mesh is also used to address operational requirements like A/B testing, canary deployments, and rate limiting.

Using a service mesh and getting the best out of it is not very straightforward. However, once you get the hang of it, you will embrace the power of service mesh with both arms. The developers will start using the features of the service mesh without explicitly knowing about them.

One fact that needs to be reiterated is that Service Mesh can be used outside of Kubernetes. Istio is being experimented with installation on Cloud Foundry, which is an open-source platform-as-a-service application. Many service meshes support the installation and management of data plane, that is, sidecar proxies, and the associated control plane on an infrastructure other than Kubernetes. HashiCorp's Service Mesh Consul supports connecting applications on multiple runtimes, multiple clouds, or on-premises environments.

What is Istio?

Istio is a very famous implementation of service mesh and the one that we are going to use as a reference in the remaining chapters of this book. This is not how Istio is defined of course.

It is defined as an open-source, platform-independent service mesh. Over the years, it has become THE Service Mesh that provides traffic management, policy enforcement, and telemetry collection.

A fun fact is that the term *service mesh* gained popularity through Istio. That is to say, many application teams and technology enthusiasts got introduced to the concept of Service Mesh after they got to know about Istio.

The Istio project was started by teams from IBM and Google in partnership with the team from Lyft that built Envoy proxy. Istio is now, a part of a Cloud Native Computing Foundation, which supports the development of cloud neutral software and helps cloud portability without the vendor lock-in.

Some of the key characteristics of Istio are as follows:

- Istio manages communications between microservices and applications, and hence, it helps take your services to the next level, which is the Service Mesh. Istio provides the service mesh features like automated baseline traffic resilience, service metrics collection, distributed tracing, traffic encryption, protocol upgrades, and advanced routing functionality for all service-to-service communication. And above all, it is provided without requiring any changes to the underlying services.

- Istio is an open-source software since its development. The source is located and maintained at a public GitHub repository **https://github.com/istio/istio**. This is the main repository for Istio that contains code for core components like istioctl tool, pilot, istio operator, and security.

- Istio uses extended version of the Envoy proxy, and the code for proxy extension is listed in a different repository: **https://github.com/istio/proxy**.

- Although initial stages of Istio supported Kubernetes-based deployment only, Istio is developed to enable supporting other deployment environments as well. Istio's control plane runs on Kubernetes, and you can add applications deployed in that cluster to your mesh. However, you can also extend the mesh to other Kubernetes clusters and connect VMs or other endpoints running outside of Kubernetes. Thus, Istio has extensible design.

Istio is a service mesh which, as we discussed, is a modernized service networking layer. The advantage is that it provides a transparent and language-independent way to automate application network functions flexibly and easily. Istio uses the battle-

tested Envoy service proxy for data plane components. Istio helps organizations run distributed, microservices-based apps anywhere, and it enables organizations to secure, connect and monitor services. It can be used with Kubernetes and with traditional workloads.

Istio architecture

Figure 11.4 features a simplified diagram of Istio architecture; the latest can be seen on their official website **https://istio.io**.

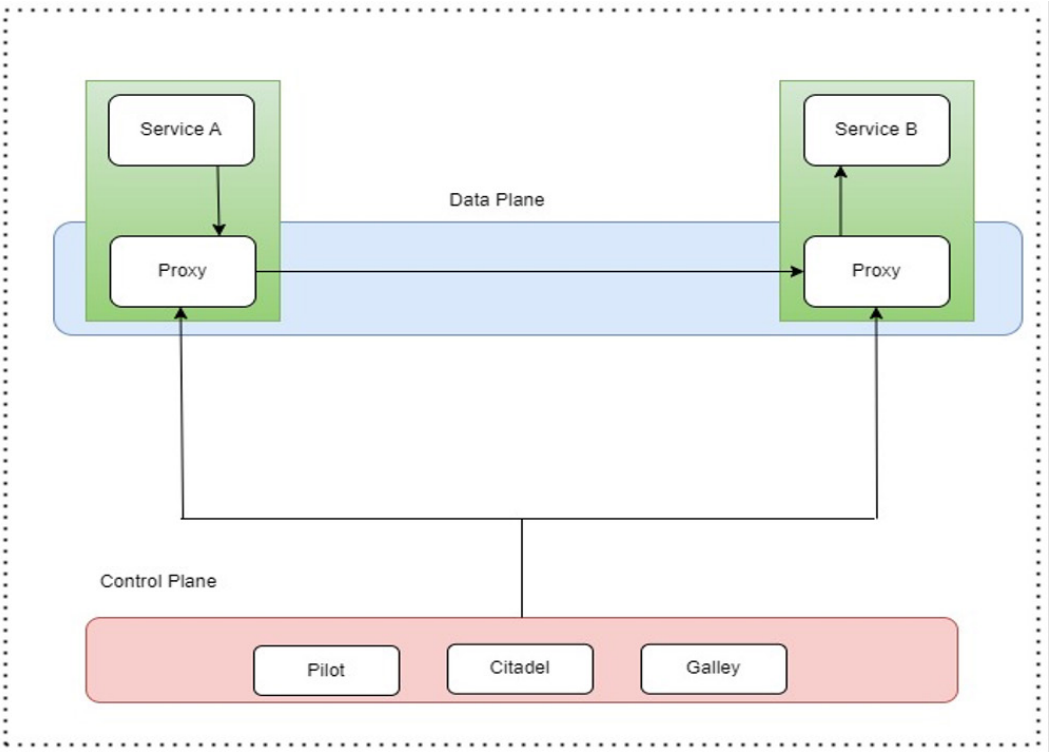


Figure 11.4: *Istio Architecture simplified*

There are two parts to the Istio Service mesh: a data plane and a control plane.

The data plane is composed of a set of Envoy proxies that are deployed as sidecars in your application containers. As shown in *Figure 11.4*, the request from **Service A** to **Service B** goes via the proxies. Thus, the proxies get to know and control all network communication between the services. The same is true for traffic going in and out of the cluster through any of the services within the cluster. Hence, the proxies are ideal candidates to collect and report telemetry on all the traffic in the mesh.

The control plane manages service discovery, service configuration and certificate management.

Let us get to know the components better.

Data plane

As mentioned earlier, the proxies used by Istio are the extended versions of the Envoy proxy. Envoy was built originally by Lyft in C++. It is a high-performance proxy developed to mediate all inbound and outbound traffic for all services in the service mesh. Envoy proxies interact with data plane traffic.

The Envoy proxy has many built-in features, as follows:

- Load balancing
- Dynamic service discovery
- TLS termination
- Circuit breakers
- Health checks
- Staged rollouts with %-based traffic split
- Fault injection
- Rich metrics

Hence, when the Envoy proxies are deployed as sidecars to services, the services also benefit from these built-in features. The presence of proxies and the mediation done by them allow Istio to enforce policy decisions. Istio can extract telemetry which can be used to monitor systems and provide information about the behavior of the entire mesh. There are other ways to extract telemetry of course, but Istio makes it easier and better.

For the application development teams, this reduces a lot of rearchitecting and coding, which would have otherwise required you to build the istio-like capabilities in the system.

The data plane helps build the following features for your application:

- Traffic management using routing rules for TCP, HTTP, gRPC, and WebSocket traffic to enforce fine-gained control over the traffic
- Resiliency features to set up retries, failovers, circuit breaking, rate limiting and fault injection
- Security features to enforce security policies, enforce access control

- Custom policy enforcement and telemetry generation for mesh traffic

Control plane

Control plane is provided by a single binary called **Istiod**. Istiod combines the control plane functionalities like service discovery, configuration, and certificate management for the service mesh. Before 2020, control plane components like Citadel, Mixer, Galley, and Pilot were built and managed independently, just like microservices. However, in real-world deployment, these components were not being deployed or scaled independently. Hence, the components are now combined into a single binary.

Istiod acts as a converter to convert the user-defined routing rules into the configurations specific to Envoy and propagates them to the sidecars at runtime. The Traffic Management API from Istio can be used to instruct Istiod to refine the envoy configuration for more granular control over the traffic between your services.

Pilot component abstracts the platform-specific service discovery mechanisms and converts them into a standard format understandable such that any sidecar, that conforms with the Envoy API, can consume. Thus, Istio can be used for services' discovery in Kubernetes deployed applications and VMs-based environments.

For security, Istiod enables strong service-to-service and end user authentication using its built-in identity and credential management. Istiod helps encrypt traffic between services as well. You can use the authorization features of Istio to control who can access your services.

Istiod's component Citadel is the built-in certificate authority, and it is used to generate certificates to allow secure communication with mutual TLS in the data plane.

Installing Istio

There are multiple ways to install Istio. You can install Istio in one of the following ways based on what best suits your use case:

- **Installing with istioctl**: You can install Istio using the command-line tool called **`istioctl`**. The **`istioctl install`** command installs the Istio components into the cluster. This command internally uses Helm Charts.
- **Installing with Helm:** Helm is like a package manager for Kubernetes, and you can use Helm charts to install the Istio. Once you have the **`istio-system`** namespace created, you can install the Istio components into the namespace with the following **`helm`** commands:

  ```
  helm install istio-base istio/base -n istio-system
  ```

```
helm install istiod istio/istiod -n istio-system –wait
```

You can optionally install an ingress gateway in a different namespace:

```
helm install istio-ingress istio/gateway -n istio-ingress –wait
```

Once done, use the **`helm status`** command to verify the status of the installation. For example,

```
helm status istiod -n istio-system
```

- **Installing using Istio Operator:** Instead of installing, managing, and upgrading the Istio on your own, you can use Istio operator, which does it for you. The **`istioctl operator init`** command creates the required custom resource definitions. Once you have the operator installed, you can create a mesh by deploying an **`IstioOperator`** custom resource.

More advanced or complex options for installing Istio also exist, where you can connect a workload running in a virtual machine to your Istio control plane. You can install a single Istio Service mesh across multiple Kubernetes clusters. The other way could be installing an external control plane and connecting multiple remote clusters to it or using multiple control planes in a single cluster.

Let us take a practical example of installing Istio using **`istioctl`** and then injecting sidecar proxies into the services running inside a cluster.

Installation using istioctl

Installing with the **`istioctl`** command line tool is the easiest and recommended way to install Istio. You should get the binary for latest release from **https://github.com/istio/istio/releases/.**

Install a binary called **`istioctl`** on your desktop; it opens a wide range of commands to use, just like **`kubectl`**.

Once you have **`istioctl`**, you can install the Istio into a cluster. The command is **`istioctl install`**.

This command installs the default Istio profile with three components called *Istio core*, *Istiod*, and *Ingress gateways*, as per Istio 1.16.1, which is the latest version at the time of writing of this book.

To take advantage of all of Istio's features, pods in the mesh must be running an Istio sidecar proxy. If you already have a Kubernetes cluster running a bunch of Microservices, then you can inject Istio's sidecar proxies by manual injection. This is not a recommended way, but it is a possibility.

Let us say you are connected to a Kubernetes cluster where you have a deployment that contains 2 pods with nginx image. Refer to *Figure 11.5* to see the sample output of the **`kubectl get pods`** command:

```
D:\>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6c6bdf4899-2srv4   1/1     Running   0          7m22s
nginx-deployment-6c6bdf4899-tt8bz   1/1     Running   0          7m22s
```

Figure 11.5: *Pods without sidecars*

You can observe that both pods contain 1 container each. The count under the READY column states how many containers are ready. Give yourself a pat on the back if your mind took you to back to the reference of readiness probes on Pods from *Chapter 2, PODs*.

With the **`istioctl kube-inject`** command, you can modify the deployment to contain 2 containers inside each pod, as shown below:

```
kubectl get deployment nginx-deployment -o yaml | istioctl kube-inject -f -|
kubectl apply -f -
```

You can verify the pods content again after the preceding command is successfully executed. Refer to *Figure 11.6:*

```
D:\>kubectl get deployment nginx-deployment -o yaml | istioctl kube-inject -f - | kubectl apply -f -
deployment.apps/nginx-deployment configured

D:\>kubectl get pods
NAME                                READY   STATUS        RESTARTS   AGE
nginx-deployment-65ddb4978d-7rj5f   2/2     Running       0          4s
nginx-deployment-65ddb4978d-xfjk7   0/2     Init:0/1      0          0s
nginx-deployment-6c6bdf4899-2srv4   1/1     Terminating   0          7h48m
nginx-deployment-6c6bdf4899-tt8bz   1/1     Running       0          7h48m

D:\>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-65ddb4978d-7rj5f   2/2     Running   0          21s
nginx-deployment-65ddb4978d-xfjk7   2/2     Running   0          17s
```

Figure 11.6: *Manual injection of sidecar*

The recommended way is to automatically inject the sidecars into the pods as they are created.

This is done by specifying **`the istio-injection=enabled`** label on the namespace before creating pods through a deployment or by any other means. When this label is set and an admission webhook controller is in place, the newly created pods have

sidecar container injected along with the main application container. You can verify this with the **`kubectl get`** pods command, as shown in *Figure 11.7*:

```
D:\>kubectl label namespace default istio-injection=enabled
namespace/default labeled

D:\>kubectl apply -f nginx-deployment-1.yaml
deployment.apps/nginx-deployment created

D:\>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6c6bdf4899-2bph8   1/2     Running   0          5s
nginx-deployment-6c6bdf4899-89pgd   2/2     Running   0          5s

D:\>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-6c6bdf4899-2bph8   2/2     Running   0          9s
nginx-deployment-6c6bdf4899-89pgd   2/2     Running   0          9s
```

Figure 11.7: *Automatic injection of sidecar*

Before we discuss an interesting detail about admission webhook controllers, let us understand in brief what they are. We discussed them briefly in *Chapter 2, PODs*; *Chapter 6, Configuring Storage with Kubernetes*; and *Chapter 8, Zero Trust Using Kubernetes*, and we can revise the concept again here.

One of the key characteristics of Kubernetes architecture is its extensibility. The admission controllers, as the name suggests, control the admission of objects into the Kubernetes cluster. Admission controllers are the piece of code that intercepts the requests to the API server and validates the related object, modifies it or does both. Thus, there can be validating admission controllers, mutating admission controllers, or the controllers who can do both. While Kubernetes provides its own admission controllers, these controllers can be developed as extensions as well, and they are run as webhooks configured at runtime. Mutating admission Webhooks are called first, and they can modify the objects sent to the API server in order to enforce some defaults. Validating admission webhooks are called to validate the object against predefined policies, and they can either accept or reject the request.

Now, coming back to the automatic injection of sidecar in your services, you should have a mutating admission webhook controller enabled on your cluster. Generally, admission controllers are enabled by default on the Kubernetes cluster. However, if they are disabled, automatic sidecar injection will not work unless you enable the **`MutatingAdmissionWebhook`** controller on your cluster.

One useful command that comes with **`istioctl`** is **`istioctl analyze –all-namespaces`**. This command helps you identify potential problems in your cluster.

Cost of using a Service Mesh

Service Mesh such as Istio provide a rich set of features that you get to use without making changes in your application. Although it helps you address a lot of cross-cutting concerns, it does not do so magically. There is a cost attached to it. The Envoy Proxy and Istiod both use some vCPU and memory from the cluster. All the communication is intercepted by the proxy, so it adds to the latency of the calls between the system.

Istio publishes the performance summary for the performance tests they perform with every release. Here are some of the performance numbers for Istio 1.16.1, which is the latest one available at the time of writing this book:

- The control plane binary Istiod uses 1 vCPU and 1.5 GB of memory.
- The envoy proxy uses 0.35 vCPU and 40 MB memory per 1000 requests per second going through the proxy.
- The Envoy proxy adds 2.65 ms to the 90th percentile latency.

Data plane performance and resource consumption

Following is a list of the factors that influence the performance of data plane:

- CPU cores
- Number of client connections
- Number of worker threads in the proxy
- Target request rate
- Request size and response size
- Protocol being used for communication like TCP and HTTP
- Number and types of proxy filters, specifically telemetry v2 related filters

Proxy does not buffer the data passing through it, so the number of requests does not increase the memory consumption. However, as the number of listeners, clusters, and routes increase, it can increase memory usage for the proxy.

Control plane performance and resource consumption

Istiod processes the combined configuration and system state from the Kubernetes environment, and the user-authored configuration to produce the expected configuration for the proxies.

The control plane can support thousands of services, spread across thousands of pods with a similar number of user-authored VirtualServices and other configuration objects. CPU and memory requirements for the control plane scale according to the number of configurations set up and the possible system states.

The following factors influence the CPU consumption for the Istio Control Plane:

- The number of proxies connected to or configured by Istiod.
- The rate of changes in the deployment.
- The rate of changes in the configuration.

If the number of pods in the cluster is huge, and hence the number of proxies, then increasing the number of Istiod instances helps reduce the amount of time it takes for the configuration to reach all proxies.

Customizing the Istio setup

While installing Istio, you can use different configuration profiles. These profiles provide the customization of both data plane and control plane components.

You can start simple with one of Istio's built-in configuration profiles and then customize the configuration for your specific needs . Let us first look at what built-in configuration profiles are available:

- **`default`**: This is the recommended profile for production deployments and for primary clusters. This profile enables components according to the default settings of the **`IstioOperator`** API. You can use the **`istioctl profile dump`** command to see default settings.
- **`empty`**: This can be used as a base profile for custom configuration, as it deploys nothing.
- **`minimal`**: This is the reduced version of the default profile, where only the control plane components are installed. This enables you to use separate profiles to configure the control plane and data plane components.
- **`demo`**: This profile has high levels of tracing and logging because it is designed for showcasing the Istio functionality. The official Istio website uses this configuration along with **`BookInfo`** application for quickstart guides. Note that due to the heavy tracing and access logs that come with this profile, it does not give the best performance.
- **`remote`**: As the name suggests, this profile is used for configuring a remote cluster that is managed by an external control plane. The other use case where you can use this profile is when you want to configure a remote cluster and manage it by a control plane in a primary cluster of a multi-cluster service mesh.

- **`preview`**: This profile contains upcoming features which are in experimental stage. Thus, stability, performance and security are not guaranteed for this profile and hence this profile is not suitable for production.

Along with installing Istio with the abovementioned built-in configurations, you can also customize it using **`istioctl install`** command. For example, to set the logging level to debug you can use the following command:

```
istioctl install --set values.global.logging.level=debug
```

To enable the control plane security, you can use:

```
istioctl install --set values.global.controlPlaneSecurityEnabled=true
```

These configurations can be found from official documentation of install options for Istio. There are too many installation options to list them all here. You can pick and choose different options for customizing the installation with regards to security, global settings, **`certmanager`** as well as the configurations for other software that is integrated with Istio service mesh like Grafana, Prometheus, Kiali and so on.

`IstioOperator` API is a recommended option for customizing the configuration. The API defines Istio in 6 components, as base, pilot, **`ingressGateways`**, **`egressGateways`**, cni and **`istiodRemote`**. Settings for each of these components are listed in the **`IstioOperator`** API documentation. The same API can also be used to customize Kubernetes settings for these components in a consistent manner.

Istio can be integrated with a bunch of software to achieve the required functionality. For example, Istio can be integrated with Grafana to show the dashboards, Kiali for observability and logs visualization and Jaeger for distributed tracing. You can integrate Istio with Prometheus for its monitoring capabilities. We will be using some of these tools in the upcoming chapters.

Conclusion

If you are building an application made up of microservices for a while and are working on its operational readiness, then you should know about Service Mesh, regardless of whether or not you choose to use it. Service Mesh is here to stay, and it is improving and maturing as you read this.

A lot of cross-cutting concerns, such as security, monitoring, and traffic management, are taken care of by the service mesh. It does not come for free; it has a cost for running, but wise usage of Service mesh will help you ensure that the benefits outweigh the cost. Istio is a very popular service mesh implementation, and it uses Envoy proxy for its data plane components.

In the later chapters, we will discuss some key advantages of Service Mesh or Istio in particular. We will also discuss traffic management, observability, and security.

Points to remember

- Service Mesh is an architecture pattern and is not tied to Kubernetes or Microservices.
- Istio is a very popular implementation of the Service Mesh, and it can be easily installed on services in the Kubernetes cluster.
- Use the **`istioctl analyze`** command to identify probable problems in your cluster and get some recommendations to fix them.
- Istio Service Mesh is made up of two components called data plane and control plane.
- Envoy proxies, which are installed alongside your application container, form the data plane.
- Control plane is a single binary that provides features like Certificate management, Service discovery and configuration for the data plane.
- You can customize Istio setup as per your needs using built-in configurations provided and the options provided with the **`IstioOperator`** API.

Questions

1. You have a monolithic application deployed using Virtual Machines in Microsoft Azure. There are a couple of new applications being developed, which are also being developed as monoliths in the initial phase. Should you be using Service Mesh to provide observability, improved security, resiliency, and traffic management among these monoliths?
2. In your Kubernetes cluster, you want to start using Istio. However, automatic sidecar injection is not working as expected, although you have put a **`istio-injectio=enabled`** label on the namespace. What could be the reason?
3. What are the advantages of using an external control plane outside your Kubernetes cluster and connecting remote clusters to it?

Answers

1. No, using service mesh for monoliths in the initial stage would not be very beneficial, and you are better off without them. Once you a have good reason or reasons to start breaking the monoliths into microservices and start defining communication patterns between them, you should start considering service mesh.

2. Check whether the **`MutatingAdmissionWebhook`** admission controller is enabled. If it is not enabled, you have to enable it using the **`kube-apiserver --enable-admission-plugins`** command.

3. Separation of concern is the biggest advantage with the external control plane deployment. The control plane and data plane are deployed on different clusters. With this model, mesh operators install and manage the control planes, whereas mesh admins configure the service mesh.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 12
Traffic Management Using Istio

Introduction

Istio manages every communication done by the services within your service mesh, which is why you can use it to manage the traffic between your services without making any changes in the services themselves. The control that you get with Istio makes the way to a lot of advanced patterns such as circuit breaking to increase resiliency of the system. It also simplifies deployment strategies like canary deployments, blue-green deployments and so on, which are otherwise not so easy to do in the absence of a service mesh.

What if you get to redirect the traffic between different versions of the same service based on the users who are calling it? What if you can mock the response from the services with a specific error code, without changing the source code? What if you can stop forwarding requests to the service after it has thrown enough number of errors? Does it not sound great? You can surely sense the flexibility to test the reliability and resiliency of the system if this is possible.

Istio does give you that power and flexibility to manage the traffic in the system with the help of custom resources that are deployed alongside your Kubernetes objects.

This is going to be the most interesting topic where we discuss Istio, as traffic management is one of the core features of Istio.

Structure

In this chapter, we will discuss the following topics:

- Traffic management via gateways
- Controlling Ingress and Egress traffic
- Shifting traffic between service versions
- Injecting faults for testing
- Timeouts and retries
- Circuit breaking

Objectives

This chapter will help you get the best of Istio by showing you multiple ways to manage the traffic using Istio. Knowing about custom resources like **`VirtualService`** and **`DestinationRule`** will help you move a lot of traffic management logic from your source code to the yaml files. Reading this chapter will give you a fair idea about custom resources like **`VirtualService`**, **`DestinationRule`**, **`ServiceEntry`** and Gateway. You should be able to manage traffic, put timeouts and retries for your service, inject faults for testing, and improve the resiliency of the system by putting circuit breaker logic, all through yamls and without requiring changes in the code.

Traffic management via gateways

In *Chapter 3, HTTP Load Balancing With Ingress*, we discussed Ingress and gateways. In this chapter, we will expand upon those concepts.

Ingress refers to traffic that originates outside the network and is intended for a service or endpoint within the network. The traffic is first routed to an ingress point that acts as a gatekeeper for traffic coming into the network. The ingress point enforces rules and policies about what sort of traffic is allowed into the network.

Istio allows you to configure ingress traffic using either an Istio Gateway or Kubernetes Gateway resource. It also provides support for Kubernetes Ingress resources. A gateway provides more flexibility and extension over customization than Ingress. Gateway allows Istio features like monitoring and route rules to be applied to traffic entering the cluster.

Istio's Ingress gateway plays the role of the network ingress point and is responsible for guarding and controlling access to the cluster from the outside-cluster-traffic.

Istio's ingress gateway can also be used for load balancing and virtual-host routing.

Following is a sample yaml for an ingress gateway:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: petservice-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 80
      name: http
      protocol: HTTP
    hosts:
    - "petservice.com"
```

The **apiVersion** informs us that this object is coming from Istio, and it is not a standard Kubernetes object but a custom object. What this yaml specifies is to create an ingress gateway for the incoming traffic on the specified host. You redirect the traffic to specific service using another custom resource called **VirtualService**, which is used to specify routing. Following is a sample yaml for a **VirtualService**:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice-vs
spec:
  hosts:
  - "petservice.com"
  gateways:
  - petservice-gateway
  http:
  - match:
    - uri:
```

```
        prefix: /pet
    route:
    - destination:
        port:
          number: 5000
        host: petservice
```

Take a note of gateways section that specifies the name of the ingress gateway. This binding denotes that requests from only this gateway will be allowed and others will be rejected with a **`404 Not Found`** error response.

In a test environment where you have no DNS binding for the host, you can pass the domain as header and use a curl command to access the service, as follows:

```
curl -s -I -HHost:petservice.com "http://localhost:80"
```

Here, **`localhost`** is the external IP for your istio **`ingressgateway`** and **`80`** is the port for that service. Generally, in case of a Kubernetes cluster setup on docker-desktop, **`localhost`** is the name of the load balancer service.

Before we look at more scenarios for traffic management, let us discuss two custom Istio objects in more details, that is, the **`VirtualService`** and the **`DestinationRule`**.

Virtual service and destination rule

What we have seen with **`VirtualService`** so far is just the beginning; it can do a lot more, and we will see many more features in this chapter. But first, let us get to the basics.

It is an object separate from the standard Kubernetes service that we saw in *Chapter 3, HTTP Load Balancing With Ingress*, and have used all along. Virtual service separates where clients send the requests from the actual workloads that implement them. Virtual service provides a way to specify different traffic routing rules for sending traffic to the workloads.

You can use a virtual service to specify the traffic behavior for one or more hostnames. You use routing rules in the virtual service that tell Envoy Proxy how to send the traffic coming toward the virtual service to the appropriate destinations. Route destinations can be different versions of the same service or entirely different services. The ability to route the traffic to different versions of the same service allows providing different deployment patterns for your service deployments. We will see how to shift traffic between different versions of the same service in this chapter.

The other way is to use a single **`VirtualService`** object for multiple services in the back end. This helps give consumers of a monolithic application the same experience

when the application is being transitioned into a composite of multiple services. For example, the requests to the route **http://mymonolithapp.com/pets** will go to a **`petservice`** service in the background, and the requests to the route **http://mymonolithapp.com/orders** will go to an **`orderservice`** in the background.

We will also see an example of how you can use **`VirtualService`** in combination with gateway object to control the ingress and Egress traffic for the cluster.

To use some of these features, you need to configure **`destination rules`**, another object that is used to specify the subset for the service. Having a separate object for destination rules improves reusability. Destination rules are used to configure what happens to the traffic for the destination. Destination rules are applied **after** virtual service routing rules are evaluated so that they apply to the actual destination of the traffic.

Thus, **`VirtualService`** and **`DestinationRule`** are the objects that play an integral role in traffic management using Istio. These objects sit in front of your actual Kubernetes service and do the magic.

Figure 12.1 features a Kubernetes service in the absence of **`VirtualService`** and **`DestinationRule`** objects:

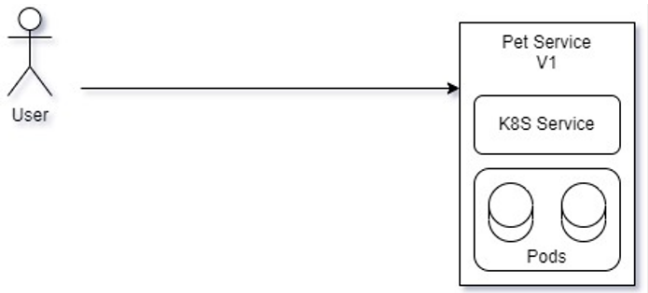


Figure 12.1: *Calling a service without VirtualService and DestinationRule*

When you create **`VirtualService`** and **`DestinationRule`**, you get something like what is shown in *Figure 12.2:*

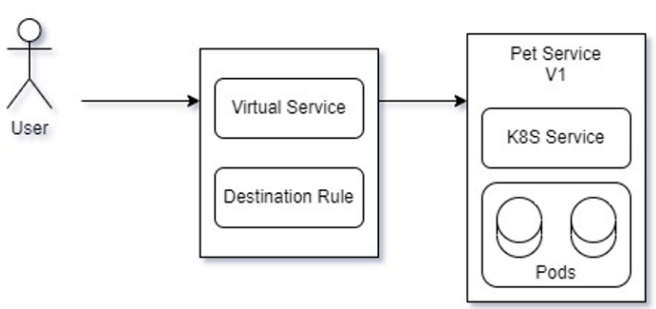


Figure 12.2*: Calling a service via a VirtualService and DestinationRule*

Destination rules also let you specify your preferred load balancing model, TLS security mode, or circuit breaker settings. Let us see the load balancing options that you can specify through the **`DestinationRule`** object:

- **Round-robin:** This is the default option, where service instances are sent the requests in turns.
- **Random:** With this load balancing option, requests are forwarded to the instances randomly.
- **Weighted:** This option can be used to specify that requests should be forwarded to instances in the pool according to a specific percentage.
- **Least request:** With this option, requests are forwarded to instances with the least number of requests.

You can use this option to specify the load balancing to be done for a single service or a subset of the service. Following is a sample yaml where you could see examples for both use cases:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: loadbalancing-dr
spec:
  host: petservice
  trafficPolicy:
    loadBalancer:
      simple: RANDOM
  subsets:
  - name: v1
    labels:
      version: v1
    trafficPolicy:
      loadBalancer:
        simple: LEAST_REQUEST
  - name: v2
    labels:
      version: v2
```

Here, the **`LEAST_REQUEST`** option is used for the v1 subset of the **`petservice`**, whereas the default policy for all subsets is specified as **`RANDOM`**.

Moreover, simple is one of the types for the **`loadbalancers`** supported by Envoy proxy. There are other types such as **`ConsistentHash`**, which is used to create session affinity based on certain properties and **`localityLoadBalancer`**, which is used to override mesh wide settings.

While we will look at many important specifications related to different Istio provided objects, official Istio documentation shall be referred to for a complete list of options for virtual service and destination rule specifications. Istio is an evolving software, and more and more features will keep on getting added.

Controlling Ingress and Egress traffic

`istio-ingressgateway` and **`istio-egressgateway`** are the services that are deployed when demo profile is installed for Istio. If you use the default profile for Istio, only ingress gateway is installed. Having said that, you can modify these gateways with your configurations or even create new gateways if required.

`ServiceEntry` is another custom resource that is used to allow Envoy proxies to send traffic to the external services as if they are part of the mesh. Thus, Service Entries help manage traffic for services outside the mesh. You can also define retries, timeouts and fault-injections in the Service Entries for external services.

Istio, by default, allows Envoy proxies to pass the requests to unknown destinations. However, you cannot use Istio features for these destinations unless you create a Service Entry and register them inside a mesh. Following is an example **`ServiceEntry`** for *google.com*.

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: google
spec:
  hosts:
  - www.google.com
  ports:
  - number: 443
    name: https
    protocol: HTTPS
  location: MESH_EXTERNAL
  resolution: DNS
```

The location tells whether the service is inside the mesh or outside. Resolution is used to determine how the proxy will resolve the IP address for instances of the service.

External destinations are specified under the **`hosts`** field. You can use a fully qualified name of wildcard prefixed domain name. The following entry is an example usage of wildcards in the domain names:

```
spec:
  hosts:
    - "*.example.com"
```

Shifting traffic between versions

Let us see how we can deploy new versions of microservices but only make them available to certain users, like a canary deployment. You have a service and you introduced enhancements to a couple of endpoints in the service. Now you have released the changes all the way up to production environment but want to enable the enhancements only for the test team.

Once the test team gives a green signal, we can enable it for a higher number of users in batches and finally, to all users. This is called canary deployment, wherein you roll out the change to all the users gradually. Thus, as shown in *Figure 12.3*, a traffic is sent to **V1** of pet services initially and then gradually shifted to **V2**:

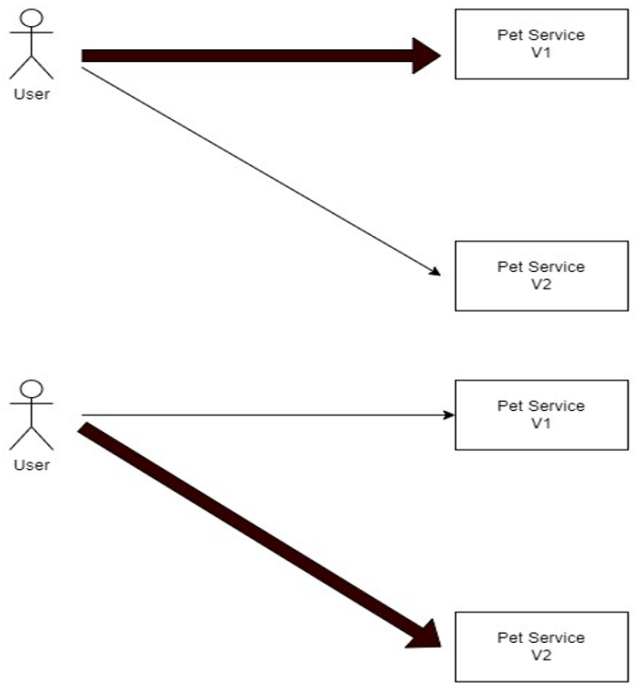


***Figure 12.3**: Canary deployment*

The other option could be following a blue-green deployment approach, where with the help of Istio, you can switch all the traffic to the newer version instantly. So, when the switch is off, the traffic is directed toward version **V1**, and when the switch is turned on, all the traffic goes to version **V2** of the service. If you find any issues with **V2** version, the switch will come in handy to revert the traffic back to version **V1**.

Refer to *Figure 12.4:*

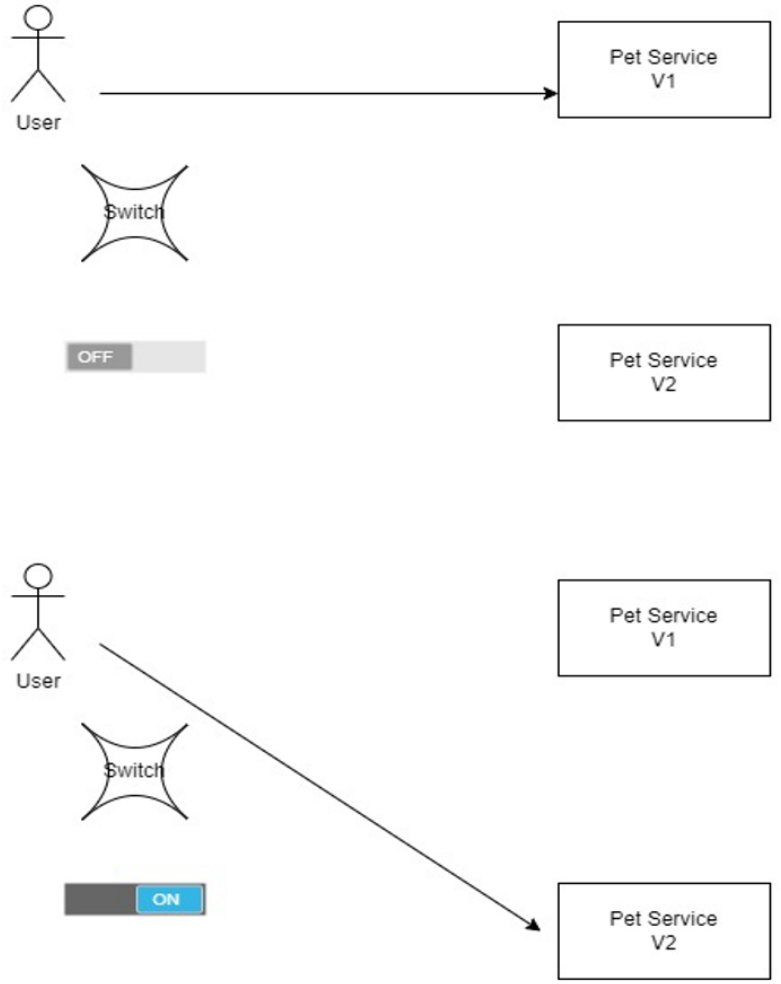


Figure 12.4: *Blue-Green Deployment*

This kind of traffic-shifting between versions is achieved via a **`VirtualService`**, which we looked at earlier, and another custom resource that we will discuss now in detail: **`DestinationRule`**. Let us look at sample yamls to understand them better.

Following is a yaml for a **`VirtualService`**:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice-vs
spec:
  hosts:
  - petservice
  http:
  - match:
    - headers:
        user:
          exact: testuser
    route:
    - destination:
        host: petservice
        subset: v2
  - route:
    - destination:
        host: petservice
        subset: v1
```

This yaml has a specification that if the value for user header is **`testuser`**, the subset **`v2`** is chosen for directing traffic. Note that the routing rules listed in Virtual Service object are evaluated in top to bottom order. The first route defined is given the highest priority, then the second and so on. In the preceding yaml, you can see that the traffic goes to subset **`v1`** by default. What the subset means is defined in the **`DestinationRule`** yaml, as follows:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: petservice-destinationrule
spec:
  host: petservice
```

```
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

subsets is the important specification here, as you may have noticed. What they specify is that if the subset v1 is chosen for routing the traffic, then the traffic will be routed to service instance with label version as **`v1`**, and if subset **`v2`** is chosen, then the traffic will be routed to service instance with label version as **`v2`**. This should emphasize how awesome labels are, and you can see how a simple label contributes to an important decision of routing the traffic.

Thus, **`DestinationRule`** works in tandem with **`VirtualService`**. You can use them to decide how to send traffic for a specific user or a set of users to a subset for the service. The subset can be considered as a group of instances of the service with specific value for a specific label. This can be a great alternative for feature flags in many software applications.

Now that we have got the idea of what a subset is, let us see one more interesting use case where you want to divert a percentage of traffic to a subset. That is let us say you want to send 50 percent of incoming requests to a subset v1 and the remaining 50 to subset v2.

Following is a **`VirtualService`** where you decide the percentage of the traffic by specifying the weight:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice
spec:
  hosts:
    - petservice.default.cluster.svc.local
  gateways:
    - petservice-gateway
  http:
```

```
  - route:
    - destination:
        host: petservice
        subset: v1
        port:
          number: 5000
      weight: 50
    - destination:
        host: petservice
        subset: v2
        port:
          number: 5000
      weight: 50
```

This is very cool, isn't it? You just put some specification in the yaml format, and the network obeys your commands. Think how complicated or weird it would be otherwise to code this through in your application logic.

Injecting faults for testing

Istio helps you do other interesting stuff like injecting faults into your network. For example, let us say you have a service called order service, and it internally calls pet service to get some details. Now, while you want to ensure that the pet service is working as expected and giving the correct response, you also want to prepare order service to handle the scenario in case there is no response from pet service or there is a delay in the response from the pet service. How do you do that using Istio virtual service? Let us take a look at another yaml:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice-vs-v2-delay
spec:
  hosts:
  - petservice
  http:
```

```
  - match:
    - headers:
        user:
          exact: testuser
    fault:
      delay:
        percent: 100
        fixedDelay: 2s
    route:
    - destination:
        host: petservice
        subset: v2
  - route:
    - destination:
        host: petservice
        subset: v1
```

The fault section under match rule states that there will be a delay of 2 seconds every time, that is, 100 percent of times, whenever the user header has a value of **`testuser`**. With this setting, your order service will get a 2-second delayed response from the pet service whenever you call it with the specific header. This is happening over the network, so it is very much equivalent to the mocking that is done in the unit tests, integration tests and so on. Such configurations enable different combinations of testing scenarios.

Let us look at another yaml first and then see what magic it does for you:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice
spec:
  hosts:
  - petservice
  http:
  - match:
```

```
    - headers:
        user:
          exact: testuser
    fault:
      abort:
        percent: 50
        httpStatus: 503
    route:
    - destination:
        host: petservice
        subset: v2
  - route:
    - destination:
        host: petservice
        subset: v1
```

This is very similar to the yaml we saw previously; the difference is that it is injecting a different kind of fault. Instead of delaying the response for some time, like before, the istio-proxy will now respond with 503 for half of the requests. Now again, with this kind of setup, you can test your order service to handle 5xx responses from the pet service.

Doing chaos testing for the services in your cluster is very easy with the help of fault injection capabilities provided by Istio. The changes you will have to make are in the yaml files, which should reach your cluster via the usual **Continuous Integration Continuous Deployment** (**CI/CD**), and then your cluster is all set to behave unexpectedly, although in a way you have defined. Note that you can try this out even in a production environment because you can control the behavior of a specific set of users.

Timeouts and retries

Every service should know itself well. The service should know how much time it takes to handle an incoming request in a worst-case scenario and in a best-case scenario. The services in Istio Mesh can specify how long the data plane proxy should wait for a response from the service, or else call a timeout.

Such specification ensures that the dependent services do not wait indefinitely for replies and the requests succeed or fail within a predefined timeframe. The setting of timeout for HTTP requests is disabled by default for the Envoy proxies in Istio.

As mentioned earlier, the developers and consumers of the service know the best or optimal value to be set for timeout. Setting it too high may cause unexpected latency, while setting it too low could result in a greater number of failures. For a service calling other services to perform the operation, the timeout for such downstream services should also be considered while defining the service timeout. Istio lets you easily specify timeouts for each service using the multi-faceted virtual service resource, and like earlier, you would not have to change your service code.

Following is a yaml for a **`VirtualService`** specifying a 12-second timeout for calls to the v1 subset of the host **`petservice`**:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice-timeout
spec:
  hosts:
  - petservice
  http:
  - route:
    - destination:
        host: petservice
        subset: v1
    timeout: 12s
```

The subset mentioned here is defined in a **`DestinationRule`**.

Just like timeout specification, Istio provides specification mechanism for retrying requests to a service in case of failures. Let us look at sample yaml first and then discuss it in detail:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: petservice
spec:
  hosts:
  - petservice
```

```
http:
- route:
  - destination:
      host: petservice
  retries:
    attempts: 3
    perTryTimeout: 5s
    retryOn: 5xx
```

The **retries** in the preceding yaml specifies that the Envoy proxy will attempt 3 retries at most to connect to the **petservice** after a call fails with response code as 5xx. The duration between retries would be determined automatically, depending upon the **perTryTimeout** and timeout values if they are specified. **perTryTimeout** specifies the timeout per attempt for a given request, including the initial call and the retries that may happen.

Retries are great because they enhance the service availability and application performance by making sure calls do not fail permanently because of temporary issues in the service. If the service can recover from transient problems and respond successfully, it improves overall availability. The interval between retries (25ms+) is variable and determined automatically by Istio, thus avoiding the called service from being overwhelmed with the number of requests.

The default retry behavior for HTTP requests is to retry twice, and an error is returned if both attempts fail. This default retry behavior may not suit every application need regarding the latency or availability. For example, too many retries to a failed service may not be required due to the impact on latency. Also, like timeouts, you can specify retry settings on a per-service basis in virtual services and fine-tune the overall system behavior. You can also fine-tune the retry behavior for your service by adding **perTryTimeouts**. That way, you can control how long you want to wait for each retry attempt to successfully connect to the service.

Circuit breaking

The circuit breaking design pattern is very useful in today's world of microservices where if an application is failing to serve the request, it can reject further incoming requests by breaking the circuit or, in other words, failing immediately and returning error response. The circuit breaker can be a proxy that monitors the recent failures in the application and uses this information to decide whether to allow further requests. If the number of recent errors is higher than the defined threshold, then the new requests are returned with exception immediately.

This kind of pattern helps make your application more resilient, saves it from Distributed-Denial-of-Service attacks, and enables fast failures.

Istio allows you to set the circuit breaking rules in the **`DestinationRule`** objects. You can set limits to how many calls to a service can fail before tripping the circuit or how many maximum concurrent connections can be set using TCP or HTTP for a service.

Let us look at a yaml to understand this better:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: petservice
spec:
  host: petservice
  trafficPolicy:
    outlierDetection:
      consecutive5xxErrors: 5
      interval: 1m
      baseEjectionTime: 3m
      maxEjectionPercent: 100
```

The **`outlierDetection`** specified under **`trafficPolicy`** is used by the Envoy Proxies to track the status of each instance of the services. These policies apply to both HTTP and TCP services. In the preceding yaml, **`consecutive5xxErrors`** specify the number of 5xx errors before an instance or host is ejected from the connection pool. The **`baseEjectionTime`** specifies the minimum time the host is kept out of the pool, where interval is the time for the sweep analysis. The **`maxEjectionPercent`** tells what percentage of hosts can be removed from the pool if they throw consecutive errors; 100% means all the hosts can be removed for the period decided by **`baseEjectionTime`**. You can, of course, have different kind of specifications for different services in the yaml files based on expected service behaviors.

Let us look at another yaml as an example to set maximum connections:

```
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
  name: petservice
spec:
```

```
host: petservice
subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
    trafficPolicy:
      connectionPool:
        http:
          http1MaxPendingRequests: 1
          maxRequestsPerConnection: 1
```

Here, the traffic policy is applied only to a subset under a destination rule. **`MaxPendingRequests`**, as the name suggests, is the maximum number of requests allowed in the queue while waiting for the connection. **`maxRequestsPerConnection`** is very useful to avoid services from failing if they accept more traffic than they can handle.

There are a lot of other settings that you can try out for circuit breaking, and you can refer to the official Istio documentation to understand more on the different traffic policies that you can set.

Conclusion

You can do wonders by effectively using custom resources like **`VirtualService`**, **`DestinationRule`** and Gateway provided by Istio. The custom resources help you move traffic management logic outside of your services.

With the help of specifications that you put in yamls, you can improve service resiliency and availability. You can control the traffic toward the services and the load balancing option you want to use for your services. You can use different deployment patterns and perform different types of system testing by injecting faults into the system.

In the next chapter, we will discuss observability, where you can visualize the magic of traffic management through logs, dashboards, graphs and so on using different observability tools with which Istio provides seamless integration.

Points to remember

- Virtual services define how you route the incoming traffic to a given destination and then use destination rules to configure what happens to traffic for that destination. These are custom objects created by Istio.
- With the help of **`VirtualService`** and **`DestinationRule`**, you can achieve different patterns for deployment, do chaos testing, and achieve circuit breaking functionality for stopping incoming requests to the services.
- You can use retries setting in the **`VirtualService`** to improve the reliability of the service.
- If you want the calls to a service automatically timeout after certain time, you can use the timeout setting in **`VirtualService`**.
- You can use *fault* specification in **`VirtualService`** to put delays in the response from a service, or you can return error for a fixed percentage of incoming requests to test the overall system.

Questions

1. What is the difference between **`VirtualService`** and **`DestinationRule`**?
2. How do you set up canary deployment using Istio?
3. What is the **`outlierDetection`** setting used for in **`VirtualService`**?
4. How do you ensure that the calls to a service do not wait beyond a certain duration?
5. What is the default load balancing policy used by Istio while forwarding requests to the services instances in the pool?
6. How can you do chaos testing for a system containing multiple services using Istio? What are the advantages of doing such testing using Istio?

Answers

1. A virtual service lets you configure how the requests are routed to a service within the service mesh. **`DestinationRule`** is used to create subsets, and those subsets are used in **`VirtualService`** to specify the rules for traffic management.
2. You can create different subsets of the service with different labels for the new and old versions. With the **`VirtualService`** and **`DestinationRule`** objects, you can shift traffic between different versions of the service for

specific group of users. You can shift more and more traffic from the old version to the new version gradually.

3. The **`outlierDetection`** setting is part of the **`DestinationRule`** object, not the **`VirtualService`** object. It is used to stop sending further calls to the service if there are more failures than the predefined threshold.

4. Create a **`VirtualService`** specific to your service and use the 'timeout' value to specify how long the Istio proxy should wait before replying with an error to the caller in the absence of the reply from the service.

5. Round-robin policy is used by default, where traffic is forwarded to the service instances in turns.

6. Using **`VirtualService`** objects, you can put delays and error responses for random services or random subsets of the services in the system. With random faults, you can check whether the system fails only for expected workflows or for all workflows. You can see what cascading effect such faults have on the system, if any. Because this is being done using yaml files only, you can restore the system relatively quickly as the source code is not changed.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 13
Observability Using Istio

Introduction

When you run your apps with Istio, all the communication between your components flows through the service mesh, and you can have Istio collect telemetry about all the service calls and forward the data to different types of services, which, in turn, can derive different types of visualization. This collected data gives you an insight into the health of your application and the status of your deployments. You can also use it to analyze performance problems and debug failures better. The telemetry collected by Istio helps DevOps teams and **Site Reliability Engineering** (**SRE**) teams a great deal with the observability that it offers, and there is no burden on the development team either. The teams get a clear visibility on how services are interacting not only with each other but also with the mesh components.

Structure

In this chapter, we will discuss the following topics:

- Understanding the telemetry flow
- Sample application and proxy logs
- Visualizing Service Mesh with Kiali

- Querying Istio metrics with Prometheus
- Monitoring dashboard in Grafana
- Distributed tracing

Objectives

In this chapter, we will discuss how to utilize Istio to observe the service mesh and get the complete view of the cluster. We will see how the telemetry flows between the service mesh and how you can collect the metrics and logs with Istio. As mentioned in *Chapter 11, Introduction to Service Mesh and Istio*, Istio supports integration with Prometheus and Grafana, which are popular visualization tools; we will also look at that integration in this chapter. You will be able to query metrics from Prometheus and monitor dashboards from Grafana. We will look at how distributed tracing works, and how to use Jaeger and Zipkin to identify latency problems. We will look at integrating the service mesh with Kiali to use it as an observability console for Istio.

We saw a lot of yamls in the previous chapter; in this chapter, we will look at lots of images to get a glimpse of the observability features that you get with Istio.

Understanding the telemetry flow

Istio uses a plugin model for gathering telemetry. The telemetry is classified into three categories, as follows:

- Metrics: Metrics provide a way of monitoring and understanding the behavior of the services in the mesh altogether. These are, in simpler words, things like the count of the number of requests and the count of the responses. Istio generates service metrics based on monitoring for latency, traffic, errors and saturation.
- TraceSpans: A trace span is an individual span in a distributed trace. Istio generates the trace spans for the distributed services. When such spans are seen together, they give a clearer sense of how the calls flow in the system and which services are dependent on which ones.
- Log entries: Services can choose to log metadata about the requests flowing in the system in a custom way. Envoy proxies can also print the access information to their standard output. Then, you can use the **`kubectl logs`** command to see this information. The messages related to the service can be seen in the logs of Istio proxies of both source and destination.

Istio uses different adapters to send the data to different back ends, and it has adapters for all the industry standard software. For example, you can send your metrics to Prometheus, your logs to Elasticsearch and your TraceSpans to Zipkin. After you have all your telemetry data captured using the back ends, you can visualize it through different front ends. Of course, Istio has support for all the leading visualization software, most of which are open source. So, Prometheus can power Kiali, which is a front end, to visualize your service mesh and see where and how the traffic is flowing. Prometheus also supports Grafana, which is another generic front-end tool for dashboards. Grafana Tempo is also an open-source tool that can be used as tracing back end, and it integrates well with Jaeger, Zipkin and Open telemetry. Istio also provides its own set of dashboards for monitoring Istio components and the services in the mesh. Zipkin has its own user interface for visualizing traces, but you can integrate Zipkin with Jaeger, which is another equally awesome front end. Jaeger helps you break down your end user requests and visualize all the service requests that were triggered due to that one user request.

Similarly, Elasticsearch has its own user interface called Kibana, which is excellent for visualizing log entries and aggregating data. We will see how these different front ends enable different aspects of observability to help you deal with different problems that you might have to debug in your applications.

Istio has support for these different types of adapters built-in, and if you use the Istio demo deployment with the sample application, it sends metrics to Prometheus and TraceSpans to Zipkin by default. So, if you want to look at the front ends for the metrics and the TraceSpans, all you need to do is enable the corresponding user interfaces.

Logging is a little different from metrics and traces, because you will typically have your own logging stack already deployed in your Kubernetes cluster, so Istio does not deploy a logging stack by default. But you can easily integrate Istio with your logging stack.

Sample application and proxy logs

Figure 13.1 features a sample application that we are going to refer to during this chapter. It consists of just three services: **Pet Service**, **Order Service**, and **Delivery**

Service. These could be written in any language, and when deployed in same cluster, they should be able to communicate with each other over HTTP:

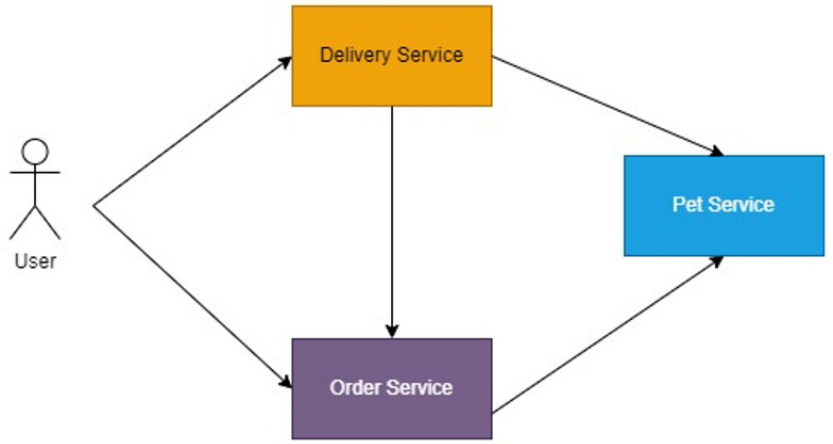


Figure 13.1: *Sample Application*

The user would interact with order service and delivery service to perform typical create, read, update, and delete operations on the objects. The Kubernetes cluster will have these three services installed, along with Istio and observability tools to collect metrics and logs from these services. Refer to *Figure 13.2* to see the number of services expected to be installed in your cluster under different namespaces as you reach the end of this chapter:

```
D:\>kubectl get svc -A
NAMESPACE      NAME                   TYPE
default        deliveryservice        ClusterIP
default        kubernetes             ClusterIP
default        orderservice           ClusterIP
default        petservice             ClusterIP
istio-system   grafana                ClusterIP
istio-system   istio-egressgateway    ClusterIP
istio-system   istio-ingressgateway   LoadBalancer
istio-system   istiod                 ClusterIP
istio-system   jaeger-collector       ClusterIP
istio-system   kiali                  ClusterIP
istio-system   prometheus             ClusterIP
istio-system   tracing                ClusterIP
istio-system   zipkin                 ClusterIP
```

Figure 13.2: *Services in the cluster*

The metrics for the services will be collected by Prometheus service. Envoy proxies inside the pods print access information to their standard output, that is, **`stdout`**. This can then be printed by the **`kubectl logs`** command. This is the simplest kind of logging. Logs in the **`stdout`** contain useful information like HTTP verb, HTTP path and the response codes.

The command to see the logs for the **istio-proxy** container within a pod is as follows:

```
kubectl logs pod-name -c istio-proxy
```

Visualizing Service Mesh with Kiali

Istio provides a basic sample installation to quickly get Kiali up and running. You can install Kiali into your cluster from the add ons provided along with Istio.

Use **kubectl apply** as follows:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/
release-1.17/samples/addons/kiali.yaml
```

You shall see Kiali service installed in your cluster under the **istio-system** namespace. Once Kiali is installed, you can check the details of the service under the **istio-system** namespace:

```
kubectl -n istio-system get svc kiali
```

Refer to *Figure 13.3*:

```
D:\>kubectl -n istio-system get svc kiali
NAME    TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)              AGE
kiali   ClusterIP   10.102.133.203   <none>        20001/TCP,9090/TCP   37s
```

Figure 13.3: *Kiali service*

Just a refresher on another **kubectl** command, **kubectl get pods -n istio-system** should show you the active pod for Kiali.

As you may already know, these **kubectl** commands are quite generic and can be used for checking the services and pods for observability tools that we use in this chapter.

One of the ways to launch the dashboards for add ons is by creating a gateway for the observability tool, like Kiali in this case, and then a virtual service on top of the actual service, as shown in the following yaml:

```
apiVersion: networking.istio.io/v1alpha3
kind: Gateway
metadata:
  name: kiali-gateway
  namespace: istio-system
spec:
```

```
  selector:
    istio: ingressgateway
  servers:
  - port:
      number: 15029
      name: http-kiali
      protocol: HTTP
    hosts:
    - "*"
---
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: kiali-vs
  namespace: istio-system
spec:
  hosts:
  - "*"
  gateways:
  - kiali-gateway
  http:
  - match:
    - port: 15029
    route:
    - destination:
        host: kiali
        port:
          number: 20001
```

The other way is to use the **`istioctl`** command to open the Kiali dashboard:

```
istioctl dashboard kiali
```

The Kiali dashboard is launched at port 20001 by default. In the dashboard, you can see different panes in the UI, such as **Overview**, **Graph**, and **Applications**. With the sample set of services that we are using, you can see the graph shown in *Figure 13.4*:

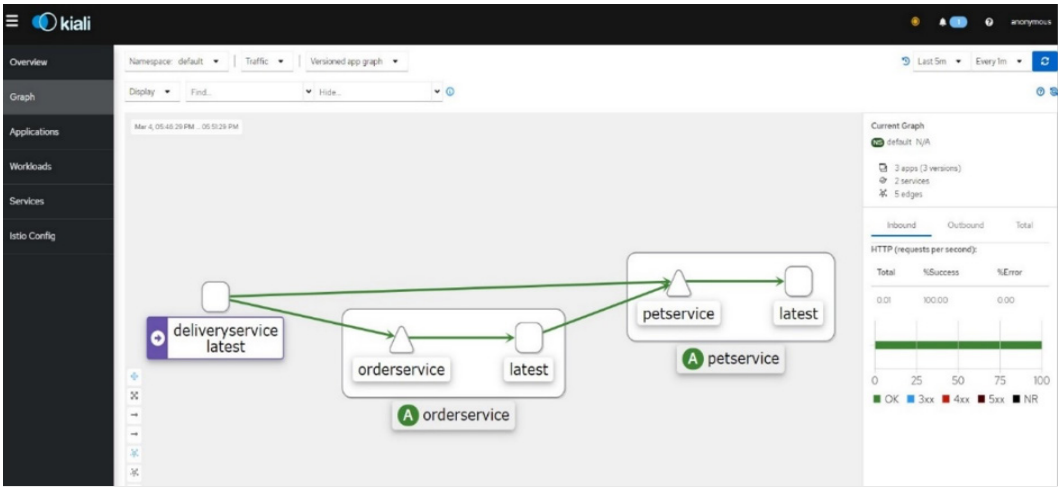


Figure 13.4: *Graph in Kiali UI*

Is it not a nice pictorial representation of the services in your cluster? You can see that the delivery service is calling order service and pet service. Order service gets called from delivery service and has dependency on pet service. Pet service does not have any dependency and only serves incoming requests.

For the sake of simplicity, we are using only three services, but even for a higher number of services in the cluster, Prometheus and Kiali keep doing the awesome job, and you get the overview of how different services connect to each other using the Kiali dashboard.

If you go to the 'Applications' pane and select a specific application, such as order service as shown in *Figure 13.5*, you can even see traces for the requests to that

application. The figure shows how requests to the order service application are distributed over time, and you can see some outliers as well:

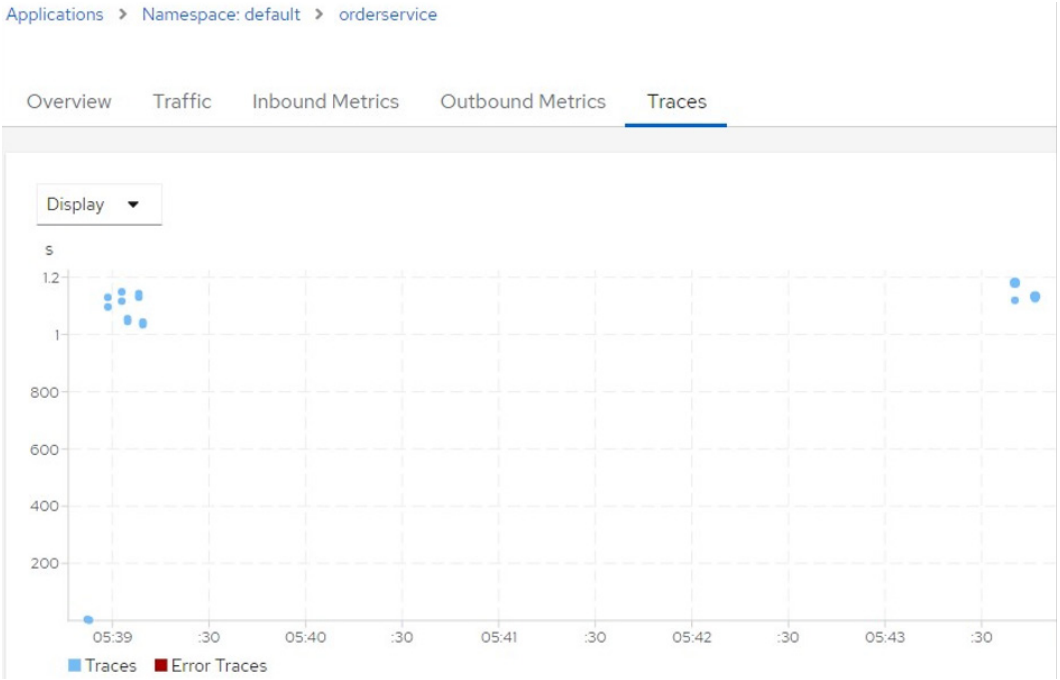


Figure 13.5: *Traces in Kiali UI*

The same can be observed for services in your cluster, and you can see more details when you hover on an individual trace. *Figure 13.6* shows traces for the delivery service and details for an individual trace:

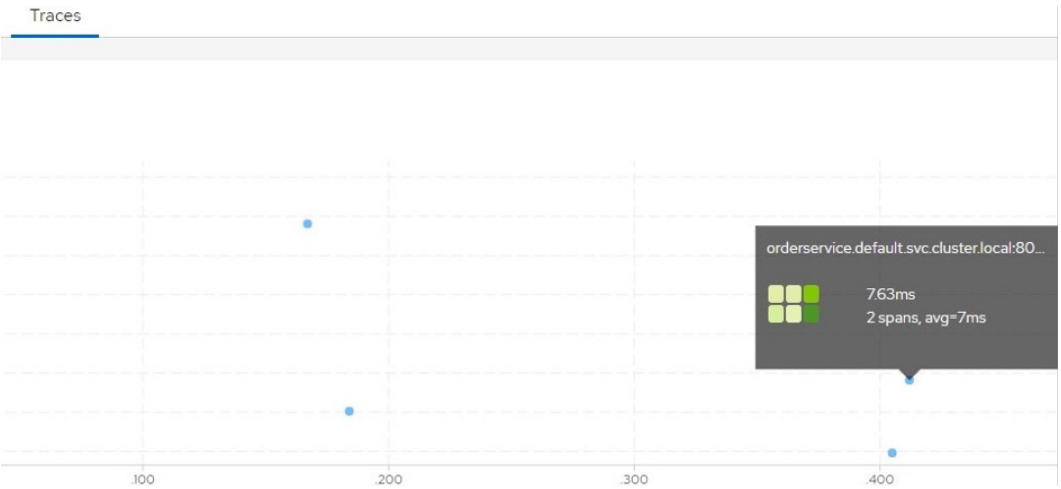


Figure 13.6: *Traces details in Kiali UI*

Now let us explore one more feature of traffic management again and visualize it through Kiali. Say you created a new version of pet service that does things differently and then you create a new deployment for pod with the **`version: v2`** label, which contains the newer image of pet service. If you remember from *Chapter 12, Traffic Management Using Istio*, we can use Istio objects **`VirtualService`** and **`DestinationRule`** to shift traffic between two versions of the same service.

Let us say you created a virtual service and destination rule to shift the traffic 50-50 between two versions of pet service. Now when you access the delivery service or order service that sends requests to pet service, you can see the traffic shifting percentage in the Kiali dashboard displayed over the connection edges in the graph. Refer to *Figure 13.7*:

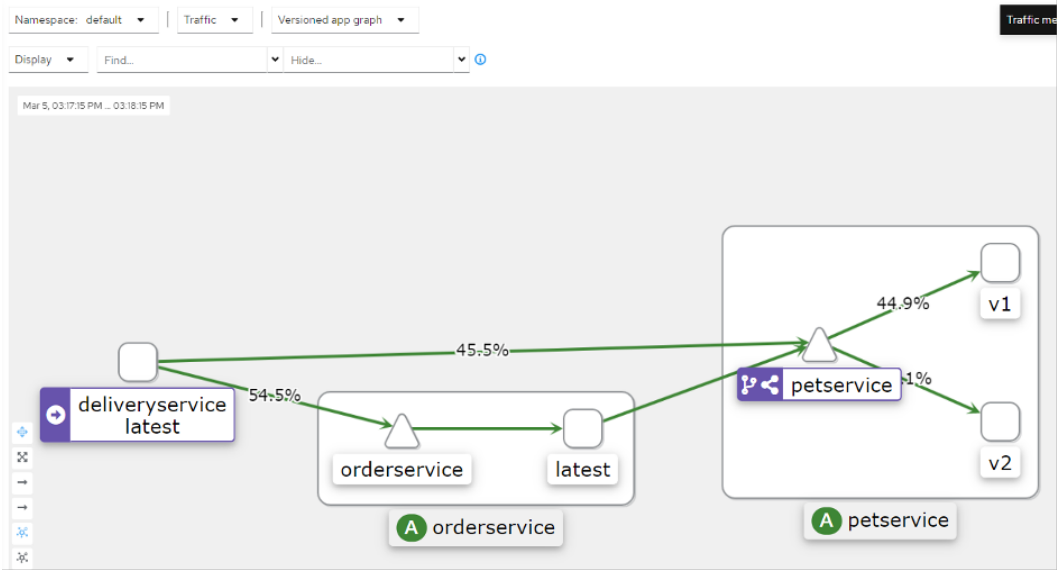


Figure 13.7: *Traffic percentage in Kiali UI*

Here you can see that the **v1** version of pet service has received 44.9% traffic, and the rest has gone to **v2**. You need to turn on the checkbox for traffic distribution under the **Display** dropdown to see the traffic percentage between the network calls.

You can also play around with the other options under the 'Display' dropdown and can see response times, traffic rate, and throughput for the network calls between

your services. Refer to *Figure 13.8*, where response time and throughput are displayed along with traffic percentage:

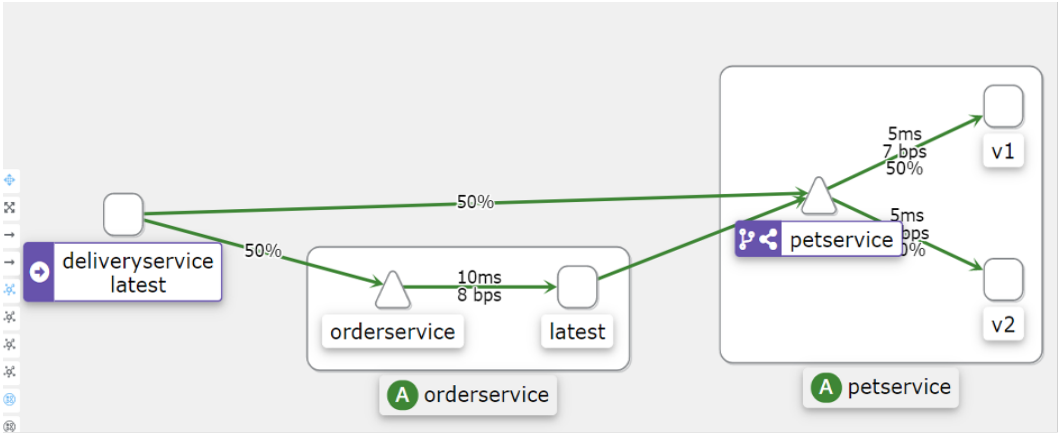


Figure 13.8: *Throughput, response times in Kiali UI*

You can see similar graph views by selecting individual deployment by going through **workloads | Namespace | Deployment-name** from the Kiali dashboard.

This should underscore how powerful Kiali dashboard is and how many different visualizations it provides for the services in your cluster. Do remember that Istio plays a key role in enabling these visualizations.

Finally, let us look at Istio config, which is a special pane dedicated for Istio objects; you can view and edit the Istio objects from this pane. *Figure 13.9* shows the **`VirtualService`** and **`DestinationRule`** objects that we created in this chapter:

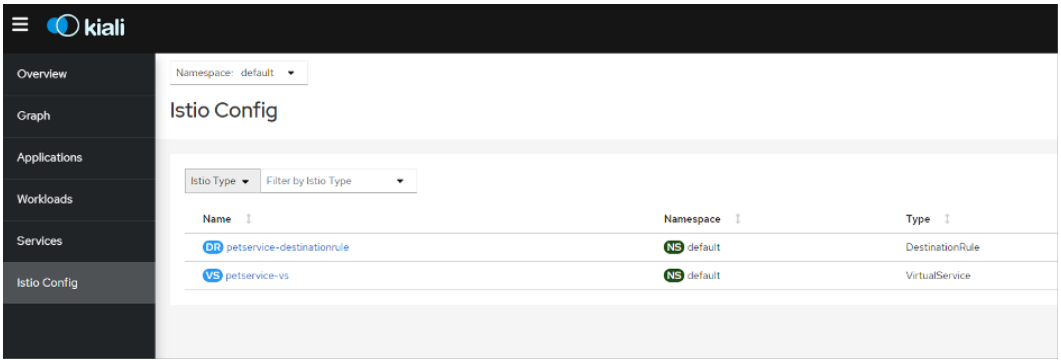


Figure 13.9: *Istio Config in Kiali UI*

Note that it is not advised to edit these objects through this dashboard, and you should follow best practices to store the yaml configurations in source control. They should be used from the repositories during continuous integration-continuous deployment processes.

Querying Istio Metrics with Prometheus

You shall have the Prometheus and Grafana add ons installed in your cluster if you use Istio demo installation, which is available in Istio's getting started documentation. If not, you can have them installed from **`istio-releases.cluster`** and **`istio-releases`**.

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/
release-1.16/samples/addons/prometheus.yaml
```

You shall see Prometheus service and deployment installed in your cluster under the **`istio-system`** namespace. The Prometheus dashboard can be launched with the **`istioctl dashboard`** command:

```
istioctl dashboard Prometheus
```

You can check that the pods for Grafana and Prometheus are running under the **`istio-system`** namespace with the **`kubectl get pods -n istio-system`** command. The Prometheus container has responsibilities like collecting metrics and putting them in a time series database. Prometheus provides an API to query the data, and it has a visual interface that is useful for exploring the data collected in storage.

The metrics explorer shows the huge list of metrics that are collected by Prometheus. You can see that there are metrics from Prometheus itself, then Kiali, and then Envoy

and Istio itself. *Figure 13.10* indicates different metrics listed inside a metrics explorer related to requests:

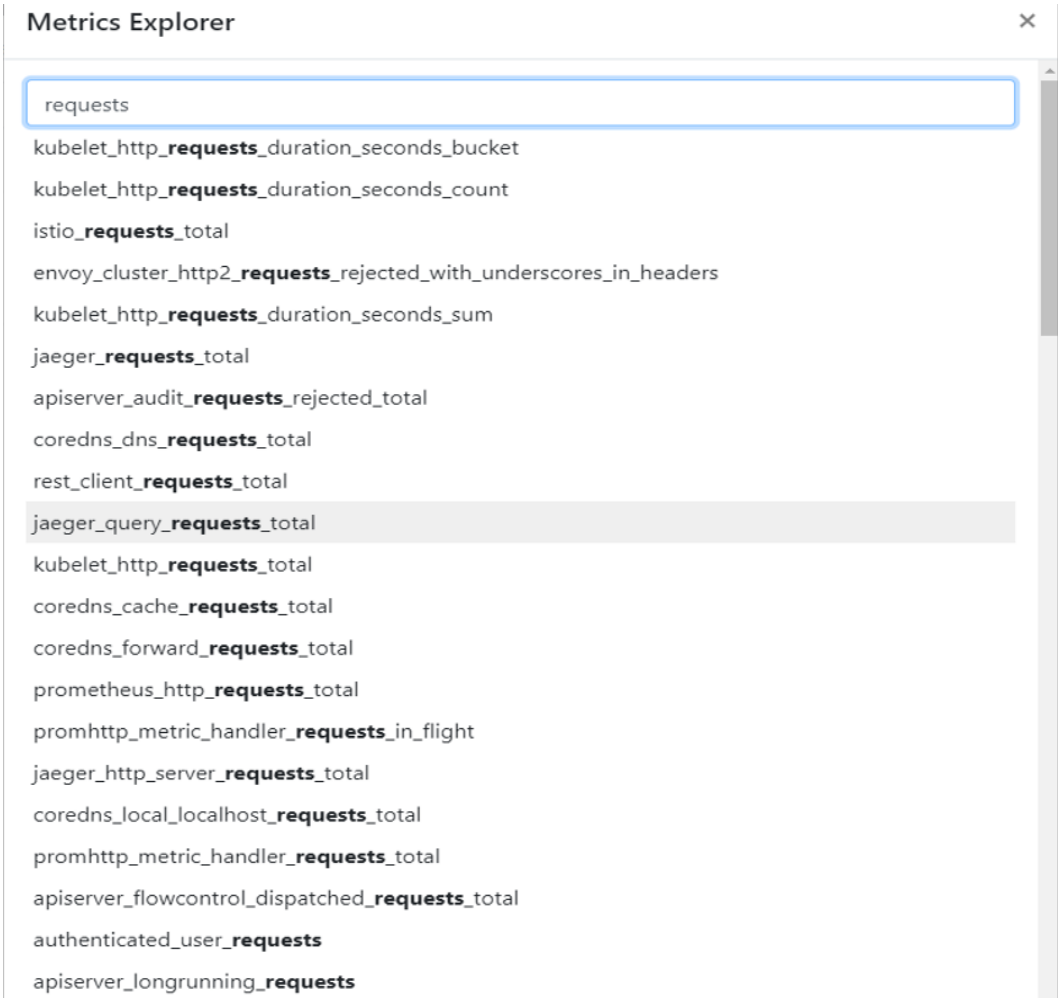


Figure 13.10: *Metrics Explorer in Prometheus UI*

You can filter them as you type what you are looking for.

If you just put **`istio_requests_total`** in the search bar and execute the query, you shall see the requests for all the services in the cluster. To filter the requests for a specific service, you can use the **`destination_service`** attribute, as shown in *Figure 13.11*:

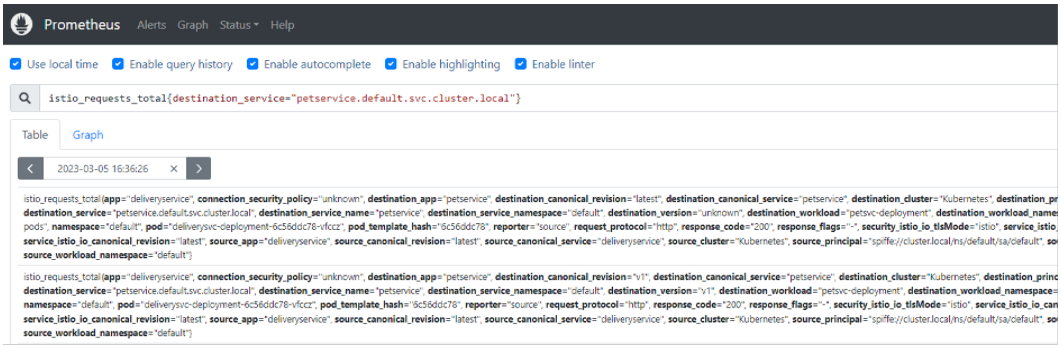


Figure 13.11: *Prometheus Queries*

How Prometheus works under the hood is that it has a pull model, and it gets the data from different endpoints. You can check the endpoints listed under 'Targets' from the 'Status' tab in the Prometheus UI. The **`scrape_interval`** property from **`prometheus configmap`** decides the pull interval. For the curious readers, it would be a good exercise to get the configmaps specific to the observability tools and check out some of the interesting settings. All these configmaps are under the **`istio-system`** namespace.

Monitoring dashboards with Grafana

While Prometheus provides querying capabilities, Grafana provides dashboards that are quite handy, as you shall see. The configured instance of the Grafana, which is part of Istio demo deployment, shows the dashboards for all the Istio components like Citadel, Mixer, Pilot and so on.

You can use the sample installation for Grafana add on from Istio directory on GitHub:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/
release-1.16/samples/addons/grafana.yaml
```

This installation provides standard dashboards for Istio. For example, Istio Mesh dashboard listed under the Istio folder gives a good overview of all the services in the mesh. Refer to *Figure 13.12*, where pet service and order service in the cluster have 100% success rate:

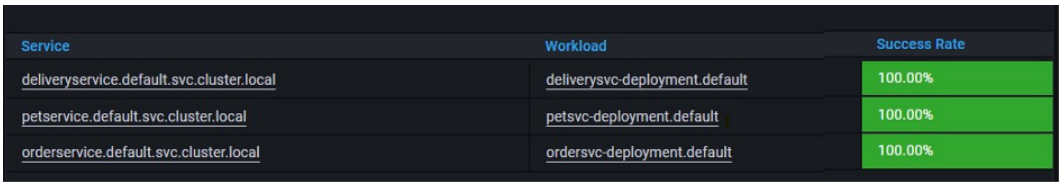


Figure 13.12: *Mesh dashboard in Grafana*

The actual dashboard also shows the requests rate and latencies at different percentiles, but they are edited from the preceding image. The Istio performance dashboard lists the CPU and memory usage of the Istio components.

In our sample application, the delivery service is calling order service. So, if you put some load on the delivery service, you can observe that in the client workloads dashboard under the Istio Services dashboard. Select the order **Service** in the service dropdown at the top of the dashboard. Refer to *Figure 13.13*:

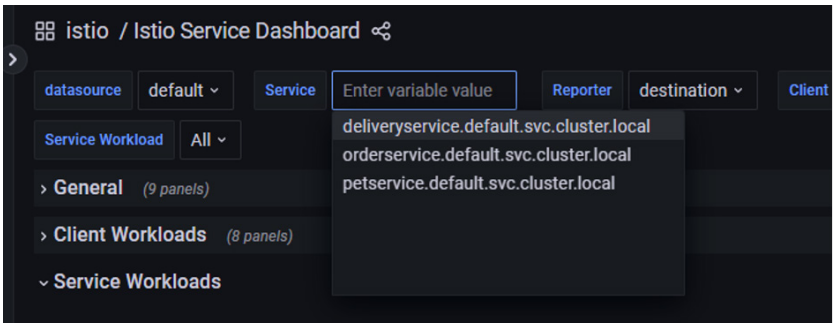


***Figure 13.13**: Services in Service dashboard in Grafana*

Then, look at the first chart under client workloads, as shown in *Figure 13.14*:

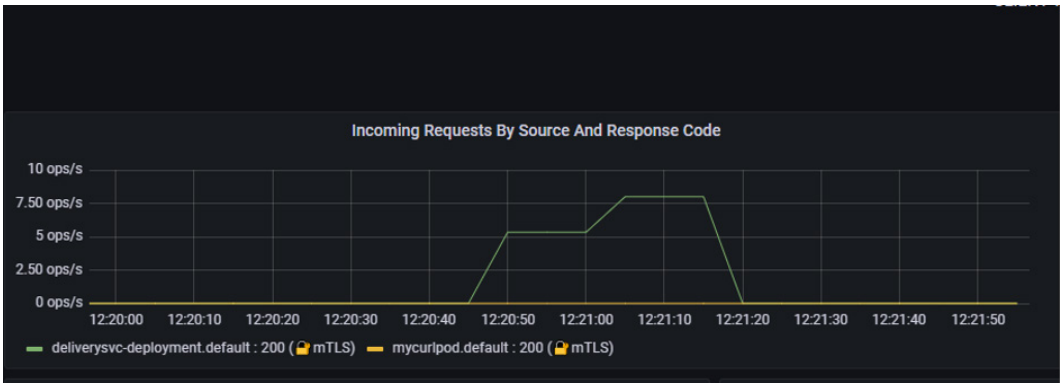


***Figure 13.14:** Incoming requests for a Service*

This shows the spike in the number of incoming requests from delivery service and all of those are responded with 200 success code from the service you have selected. It also lists **`mycurlpod`** deployment which is also one of the sources in the past for order service, and it can be seen that for the selected time range, there are no incoming requests from the **`mycurlpod`** deployment.

Then, the Istio service dashboard can be used to see the metrics for the requests on the order service. Refer to *Figure 13.15*, where metrics for incoming requests and success rate for the orders service are shown:

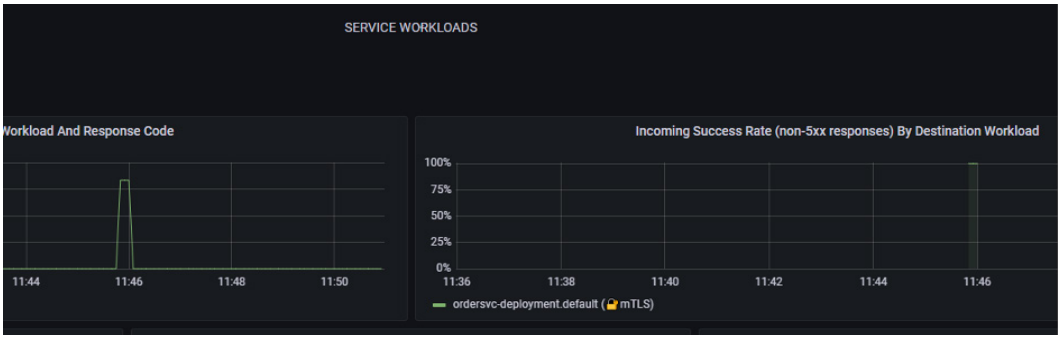


Figure 13.15: *Service workloads in Grafana*

There are several interesting dashboards that you get out-of-the-box with Istio demo deployment, and one interesting dashboard to check out is the outbound services dashboard under the Istio Workloads Dashboard. Run the following command to create a pod in your cluster that contains the curl image:

```
kubectl run mycurlpod --image=curlimages/curl -i --tty – sh
```

This will take you to the shell prompt in the pod, and you can send a different number of HTTP requests to the delivery service, order service and pet service via curl command.

The curl commands are:

```
for i in $(seq 1 200); do curl -s “http://deliveryservice:8080/delivery"; done;

for I in $(seq 1 150); do curl -s“"http://orderservice:8080/orde”"; done;

for i in $(seq 1 100); do curl -s "http://petservice:8080/pet"; done;
```

Now you can select **`mycurlpod`** as workload from the Istio workload dashboard and check the Outbound Services dashboards. You shall see a picture as shown in *Figure 13.16*:

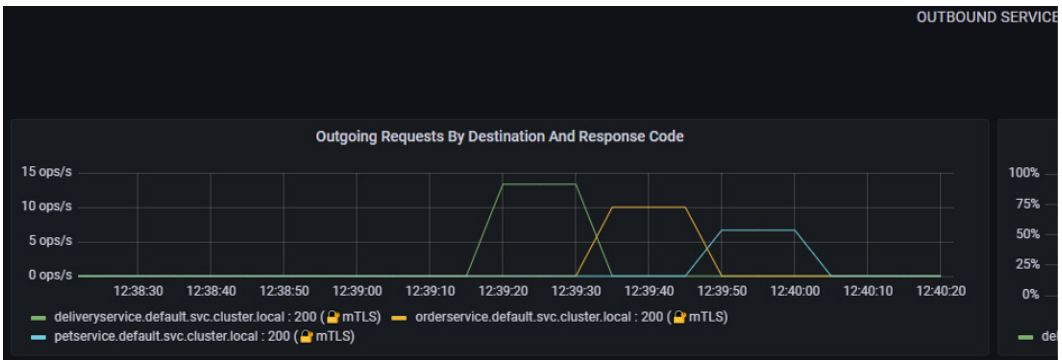


Figure 13.16: *Outgoing requests in workload dashboard in Grafana*

While the previous dashboard shows the number of outgoing requests to different services in the cluster, you can use the outgoing success rate dashboard to see which services are returning successfully and which ones are not. Refer to *Figure 13.17*, which shows that all three services have 100% success rate:

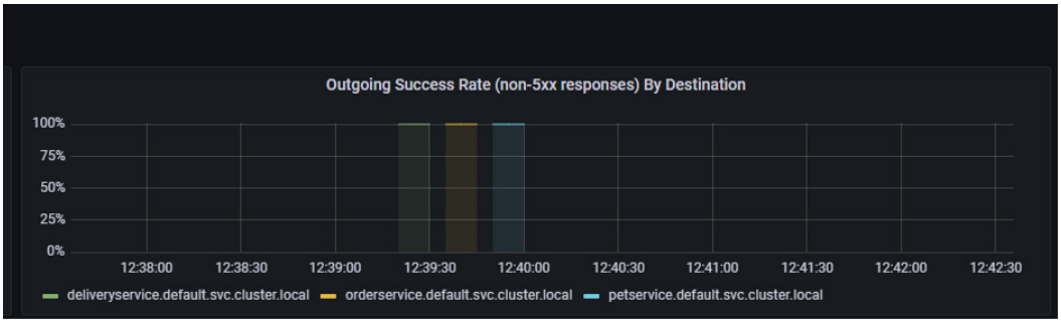


***Figure 13.17**: Outgoing success rate in workload dashboard in Grafana*

The number of dashboards available in Grafana for Istio may overwhelm you at the beginning, but once you start getting used to them, you will be the best judge to decide which dashboard to use for what purpose. Especially for the **Site Reliability Engineers** (**SREs**), such dashboards give a good overview of the system health and communication patterns without peeping into the source code for different services in the cluster.

One more interesting thing to note is you can create alerts from the Grafana UI. You can write queries on the metrics provided by Prometheus, write conditions, and then define alert behaviors like sending email, running playbooks and so on. Of course, it is advised to store your yamls in repositories and access them via version control tools.

Distributed tracing

In a microservices-based application, it is quite common for a user request to travel through multiple services to achieve the desired result. For example, in the sample application that we are referring to throughout this chapter, there is a very small number of services, so it is relatively easy to see how a request travelled from one service to another. However, in a complex application where the number of services is higher, it becomes tricky to determine which service took how much time to process a request, and developers may get lost in the services. Getting insights into how a request spanned over different services can help identify the performance bottlenecks in the system.

Istio uses Envoy's distributed tracing feature to provide tracing integration. Istio provides options to install various tracing back ends, and you can configure proxies to send trace spans to them automatically.

Just like the add-ons we used earlier in this chapter, refer to the Istio documentation for installing the Jaeger add on in your cluster. Following is a sample command:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/
release-1.17/samples/addons/jaeger.yaml
```

You shall see the jaeger-collector, tracing and Zipkin services installed in your cluster under the **`istio-system`** namespace after this installation is complete, which starts collecting the traces sent by the applications.

There is a little catch when it comes to getting the view of traces of the requests across services. You need to pass appropriate headers between services to ensure that Jaeger and Zipkin can link the requests together to give you an aggregate view. You do not get this for free. Some application work is required, which is not huge either. You need to use a bunch of client libraries, in the development language chosen for the services, in your application to include certain headers in the requests that you pass from your services.

For Zipkin and Jaeger, the B3 multi-headers that should be forwarded are as follows:

```
x-b3-traceid, x-b3-flags, x-b3-spanid, x-b3-parentspanid and x-b3-sampled
```

Passing these headers along is an easy approach to tracing. As long as all the services in your cluster carry forward these headers, Istio should be able to work out that all the requests belong to one user request or different users. Thus, when Istio proxies send spans, the back end joins them to form a single trace.

Before we look at traces going through multiple services, let us see how it looks like for a single service. Refer to *Figure 13.18*, which has 50 traces for **`orderservice`**:

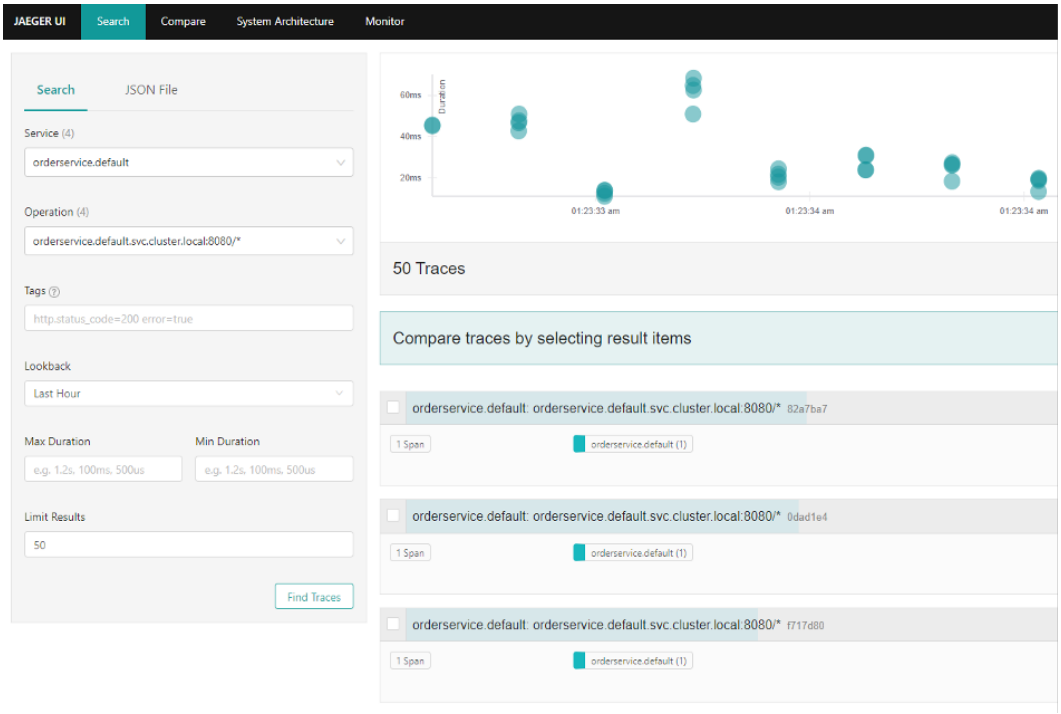


Figure 13.18: *Success traces in Jaeger*

Istio uses the Zipkin format, and the adapter sends data to Zipkin, which gets surfaced here in the Jaeger UI.

We have seen all-positive scenarios while looking at Grafana dashboards, where success rates were 100% and responses were all 200%. Let us look at a negative scenario through the eyes of a Jaeger. So, if there is something wrong with pet service and it starts throwing errors, you could observe that as well in traces. Refer to *Figure 13.19*, which shows traces with errors for calls from order service to pet service. These many errors in a short span of time clearly indicate that the error rate from pet service is high and it needs attention:

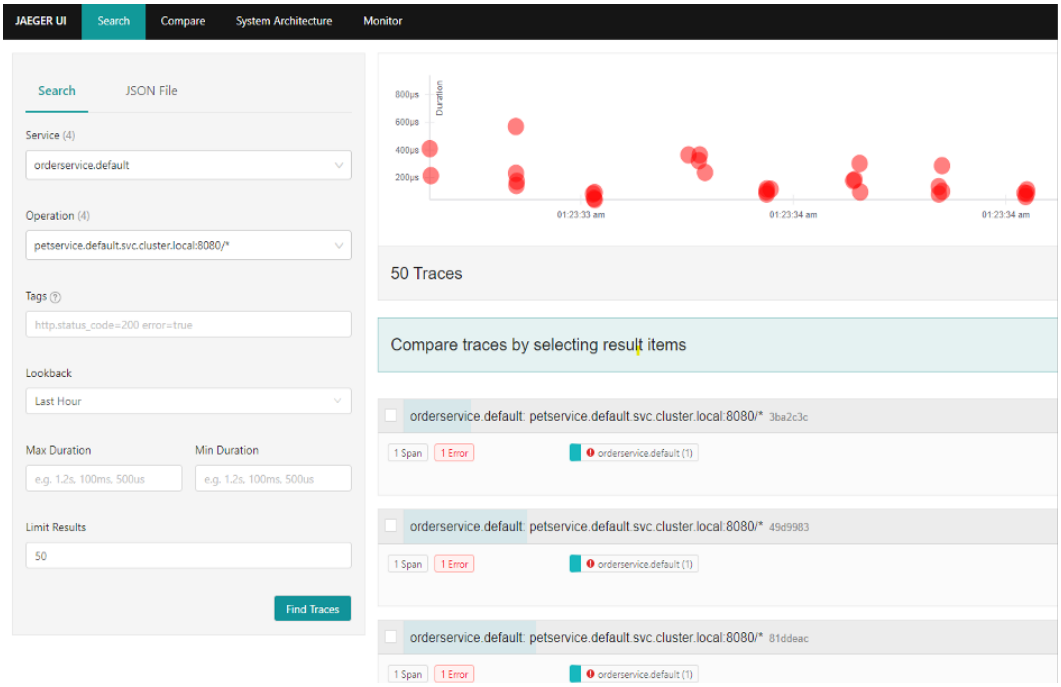


Figure 13.19: *Error traces in Jaeger*

You can inspect individual traces closely to find out which service gave what error response. Refer to *Figure 13.20*, which shows that the pet service responded with a 503 error code and the **`error`** tag is set to true for the same reason:

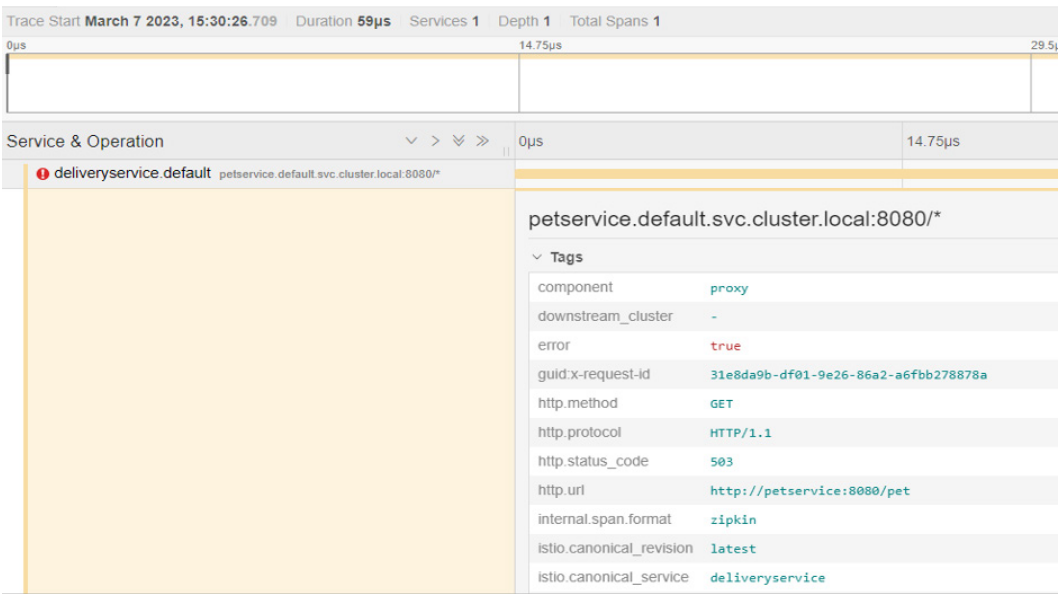


Figure 13.20: *Trace details in Jaeger*

Now, let us look at how Jaeger helps you track a request going from one service to another. *Figure 13.21* is an image of a span indicating that out of 3.17 ms taken by the delivery service to respond to a request, 2.23 ms were taken by pet service:

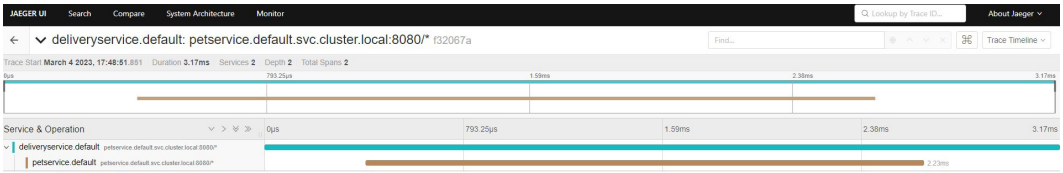


Figure 13.21: *Trace spans across 2 services in Jaeger*

Such spans help you identify the performance bottlenecks. Look at another example in *Figure 13.22*, where the delivery service calls order service and order service calls pet service. Here, you can see the time taken by individual service to perform their part of the operation:

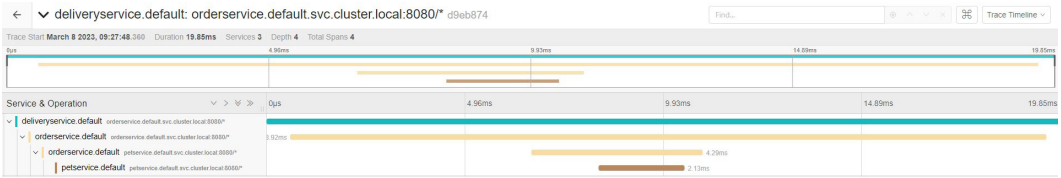


Figure 13.22*: Trace spans across 3 services in Jaeger*

This is especially useful when the latency is increasing rarely and you need to identify which component in a distributed system is adding to it. The actual meaning of the word Jaeger is a category of seabirds that are strong fliers, and they follow weaker birds from one place to another. If you did not know this earlier, look at the preceding images of traces and spans again, and see if you develop a new perspective.

Let us quickly touch upon the Zipkin UI, which you can launch using the **`istioctl dashboard zipkin`** command.

Zipkin is the tool that collects the traces, and Jaeger reads from it. Zipkin has its own UI, and you can search and inspect traces from Zipkin UI as well. Refer to *Figure 13.23*, which shows the number of traces for delivery service:

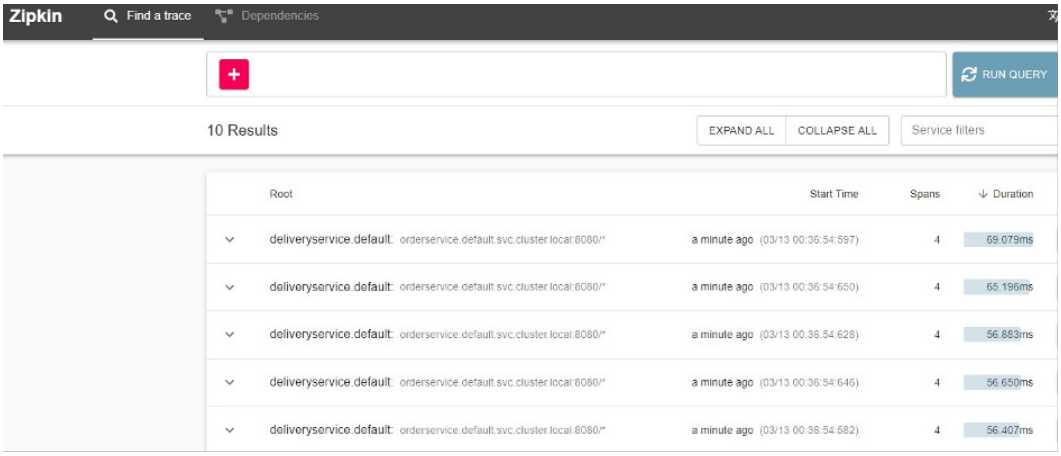


Figure 13.23: *Traces in Zipkin UI*

When you select an individual trace, you can get internal spans, as shown in *Figure 13.24*:

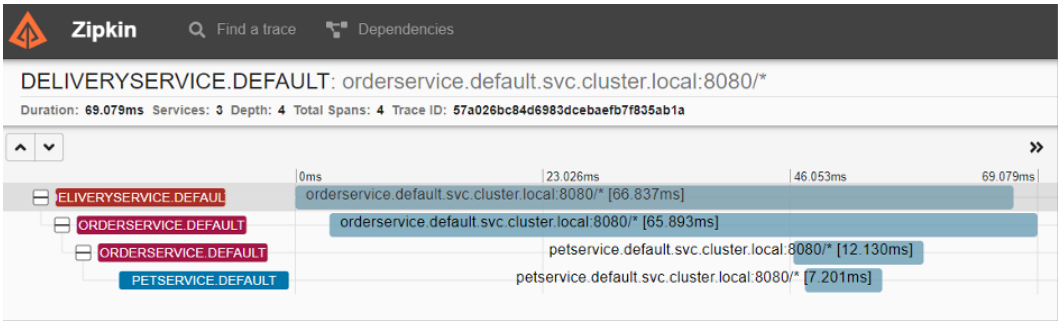


Figure 13.24: *Spans in Zipkin UI*

You can experiment with Zipkin UI and Jaeger UI to decide which one suits you best.

Conclusion

Istio provides plugins for industry standard observability tools. The data plane proxies installed with Istio provide telemetry that can be sent to different back-end tools, and then you can use different visualization tools to get a good view of the services within your cluster.

Istio ships with several adapters, and we looked at the Prometheus adapter for collecting metrics and the Zipkin adapter for collecting distributed traces.

Kiali is a very powerful tool to visualize your service mesh, and you can get different visualizations for workloads, services, and Istio objects in your cluster.

Grafana provides multiple dashboards that help you observe latencies, incoming-outgoing service requests, success rates and so on.

Jaeger provides the traceability required to debug latency issues and helps you track how the request is travelling from one service to another in the mesh.

Points to remember

- You can see the traffic percentage split between different versions of your service, the response times, traffic rates, throughput and traces of the calls to the services in your cluster using Kiali.
- You can see different Istio specific configs in Kiali UI, but editing them from Kiali UI is not advised. It is better to manage them using the version control system.
- You can check the endpoints Prometheus pulls the data from under the 'Targets' from the 'Status' tab in Prometheus UI.
- With the Istio demo deployment, you get Jaeger installation, but to have meaningful spans and traces, you need to pass standard headers from one service to another while handling the requests.
- Components in the service mesh need to add Open telemetry standard headers to the network calls, and Istio can write this telemetry data to Zipkin to power distributed tracing tools like Jaeger.

Questions

1. How can you access the logs of Envoy Proxies in the pods?
2. How does Jaeger connect multiple requests into a single trace?
3. Can you see Istio's custom resources in Kiali UI?
4. Can you validate Canary rollouts using Kiali UI?
5. What is the advantage of traceability tools like Jaeger?

Answers

1. You can access the logs of Envoy proxies using the **`kubectl logs`** command.
2. All the requests having the same values for **`b3`** headers will be connected to a single trace.
3. Yes, you can use the Istio Configs dashboard in Kiali UI to see Istio specific custom resources.
4. Yes, you can see the traffic flow between services along with the traffic break up in weights in Kiali UI.
5. The traces capture the duration of each service call, and you can see them in the Jaeger UI. This helps identify which services are adding to the latency.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Chapter 14

Securing Your Services Using Istio

Introduction

When we introduced the concept of Service Mesh, we touched upon the pros of breaking monolithic applications into microservices and some of the challenges this approach brings. We discussed Zero-Trust security, where the fundamental principle is to protect each component or service within the application from security attacks. This can be achieved by traffic that's not trusted coming from outside the cluster and the traffic within the cluster itself.

To follow the Zero-Trust security model, the different services in the cluster must communicate with each other using encrypted traffic. Along with encryption of traffic, the authentication and authorization should be in place for all the services. Istio provides a solution to these requirements. Istio helps secure the service endpoints and the communication between services. Istio proxies installed alongside the service containers act as policy enforcement points and play a critical role in security as well. Thus, a cluster having Istio installed and proxy-injection enabled for its services is secure by default. That is a pretty aggressive statement, and in this chapter, we will try to reach there as a conclusion.

Structure

In this chapter, we will discuss the following topics:

- Identity Management with Istio
- Authentication with Istio
- Authorization with Istio
- Security architecture of Istio

Objectives

Istio has enough capabilities to help you implement defense-in-depth. In this chapter, we will discuss how Istio aids the security architecture of your overall application, comprising the services in a cluster. You will be able to manage the certificates in Istio first, and then we will use Istio to provide mutual TLS between services. We will discuss how to set up different authentication policies with different scopes in your cluster, and the authentication policies that can be used for basic end-user authentication. Moving on, you will learn how to set up authorization policies to control users' access to your services. We will see a lot of yamls for authorization policy objects, including some fun stuff, such as deny-all and allow-all policies. The images related to authentication architecture, authorization architecture and the overall security architecture will give you a bird's-eye view of the overall system.

Identity Management with Istio

Protecting application data that is of critical value is an integral part of making an application secure. It is also important to ensure that the data does not fall into the hands of an unauthorized user. Thus, identity is the fundamental concept of any security infrastructure because you must identify the right users (read as authorized), and also the not-so-right users (read as unauthorized) for your application.

The application and the user must provide identification with each other to ensure that they are talking to the right person or tool. Similarly, in a service-to-service communication, two services must identify themselves to the other party and similarly, get proper identification of the other party too. The identification helps in auditing purposes to define who did what. Moreover, if the pricing model for your application is based on the period of usage, you can charge different users according to their usage.

Istio's identity model allows identification for individual users, services and a group of services. If the platform does not have a service identification, Istio can use other identities that help group workload instances.

Istio provides identities using X.509 certificates. Istio agents work with Istiod, the control plane of Istio, to automatically generate keys and certificates, and rotate them on expiry. X.509 is an international standard for communication that defines the format of public key certificates.

Identity verification in TLS

Let us get a quick overview of identity verification portion of **Transport Layer Security** (**TLS**).

To use TLS in a web application, you need two keys: one public and the other private. The public key is often called **certificate**. The public key is shared with everyone else via a certificate. To begin the communication, the client asks the server for identification. The server responds with its certificate to the client, and the client then sends a secret number encrypted using the public key in the certificate to the server and challenges the server to decrypt it. The server can decrypt the number only if it has the corresponding private key. This proves that the server has the right private key and thus, the server is the owner of this certificate, because the client never shared its private key anywhere. This is the basic of how identity verification is done in TLS.

Certificate generation process in Istio

Let us understand the certificate generation process first, and then we will see how the certificate is used by the services in the cluster.

Istiod accepts certificate signing requests, and the Istio agent sends requests to Istiod after generating the private key. The certificate authority component in Istiod validates the credentials in the certificate signing request and the certificate is generated only upon successful validation. The Istio agent stores the signed

certificate and the private key with itself. *Figure 14.1* features certificate generation with Istio:

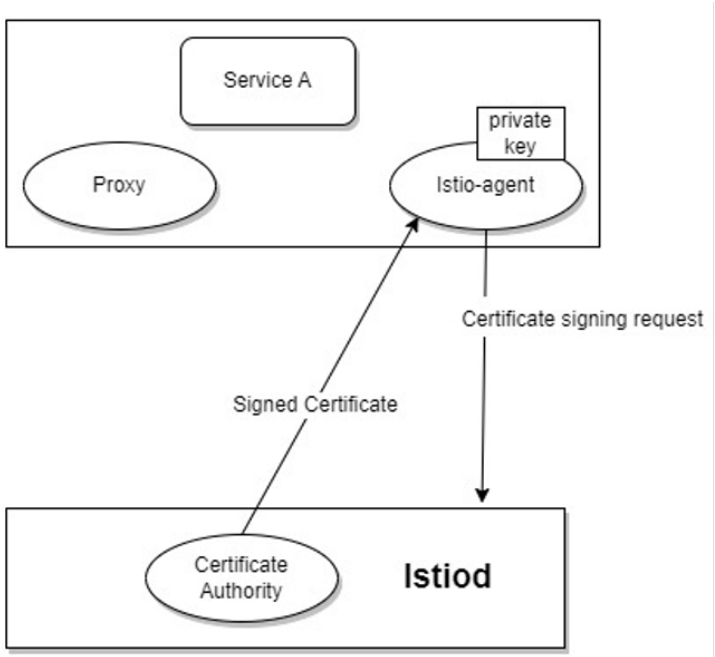


Figure 14.1: *Certificate generation with Istio*

When the service is started, the data plane that is Envoy proxy in the container, asks for a certificate to the Istio agent. The Envoy secret discovery service is used for this communication to share the certificate securely. Istio agent shares the private key and the certificate received from the Istiod. *Figure 14.2* features the Proxy using the certificate:

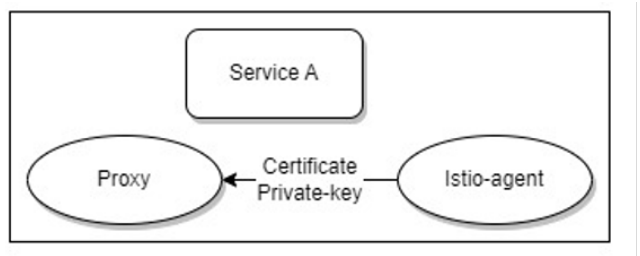


Figure 14.2: *Proxy using the certificate*

An Istio agent keeps track of the expiration of the certificates. The certificates and keys are rotated periodically, and then the preceding process of sharing certificate and key is repeated.

Thus Istiod, **`istio-agent`** and **`envoy proxy`** work together to provide identities to the workloads in the cluster. While these certificates help encrypt communication between services within the cluster, HTTPS should always be used to for communicating with the services outside the cluster to ensure that security is not compromised.

Authentication with Istio

Istio provides two ways for authentication: peer authentication and request authentication.

Peer authentication is used for service-to-service authentication to verify the identity of the client making the connection. For transport-level authentication of the communication between clusters, mutual TLS can be enabled without making any changes in the service code.

Peer authentication is used to secure service-to-service communication, and it provides the services within the cluster with an identity. Peer authentication also provides a key management system that is used during the key and certificate generation scenario that we discussed before. The same key management system is used for the distribution and rotation of the keys and certificates.

Request authentication, on the other hand, is used for end-user authentication, that is, to identify the credentials, which are mostly auth tokens passed along the request. JSON Web Tokens (JWT) passed as headers in HTTP requests is quite common. Istio enables authentication using JWTs and also has support for OpenID Connect providers like Auth0, Keycloak, Google Auth, and so on, which provide authentication.

The authentication policies are stored in the Istio config store by Istio. The control plane Istiod keeps the policies up to date for each proxy, along with the keys as required. One of the good things is that Istio supports permissive mode for authentication, which eases the integration of the existing systems with the new systems. The permissive mode accepts plain text traffic along with mutual TLS traffic. This also gives administrators control over the security stature of the overall application.

Mutual TLS authentication

Mutual TLS or mTLS takes the transport layer security one step ahead, where the server accepts the requests only from trusted clients and not just everyone. This means the server also verifies the identity of the client and rejects a request if it is from an untrusted source.

The Envoy proxies installed alongside the application in the Istio enabled cluster act as policy enforcement points because the service-to-service communication is routed through these Envoy proxies.

Thus, let us say the delivery service gets a request and needs to call order service to fetch some data to be able to respond to the incoming request. Following are the steps that are taken to ensure that communication is secure:

1. Istio redirects the traffic from delivery service to the sidecar Envoy proxy.
2. The Envoy of delivery service then starts the mutual TLS handshake with the Envoy proxy of the order service. As a part of handshake, Delivery service envoy checks the name of the service account presented in the certificate received from order service.
3. After the handshake is done and identities are established, the request from delivery service is forwarded to the order service envoy.
4. The order service envoy does the authorization check against the request, and if the check is successful, the request is passed to order service container via local TCP connection.

Refer to *Figure 14.3*:

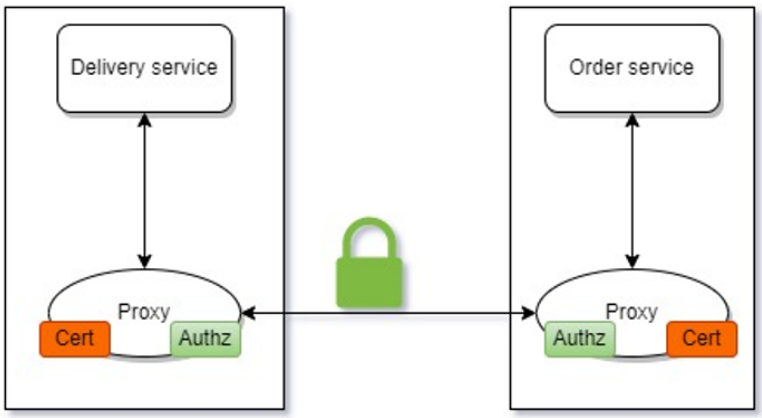


Figure 14.3: *Mutual TLS communication between 2 services*

Istio configures **`TLSv1_2`** as the minimum TLS version for both client and server. Official Istio documentation shall be referred to check the supported cipher suites.

Secure naming

There is one important concept called secure naming that needs to be discussed to know more about mTLS authentication. Service names are discovered in a cluster via

discovery service or DNS, whereas the Server identities are encoded in certificates. The mapping between service names and the corresponding server identities is called secure naming.

In the preceding example that we saw about communication between delivery service and order service, let us say that mapping of identity **`delivery-admin`** to delivery service is done. This means **`delivery-admin`** is authorized to run delivery service. Similarly, a different identity, such as **`orders-team`**, is authorized to run orders service. The control plane keeps communicating with the **`apiserver`** and generates the secure naming mappings. The control plane is also responsible for distributing the mappings securely to the Policy Enforcement Points or data planes, which then use the secure naming mappings during the mTLS communication.

Let us consider a scenario where a malicious user obtains the identity, that is, the private key and the certificate, of **`orders-team`**. Let us say this user also hacks to route the traffic intended for delivery service to its own malicious server, as shown in *Figure 14.4*:

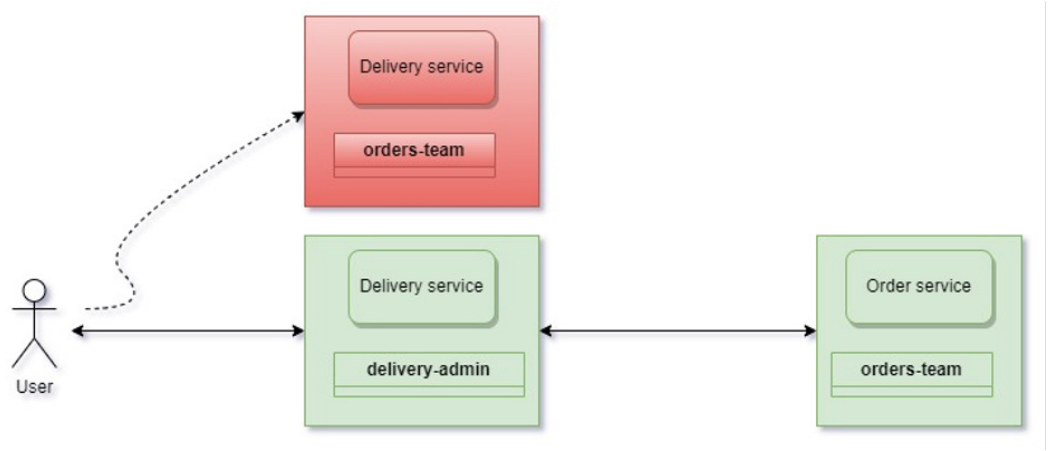


Figure 14.4: *Malicious user scenario*

Now, when a client calls the delivery service, it would be redirected to the malicious server called delivery service. However, in response, the caller should get orders-team identity in the certificate of delivery service. The client should rightfully check whether the orders-team identity is allowed to run the delivery service. Since ONLY the delivery-admin is allowed to run the delivery service, the client should fail the authentication check, and further communication should be stopped. This is how secure naming helps avoid security attacks due to spoofing.

Peer authentication with a sample application

Let us understand service-to-service authentication with a practical example of the sample application that we used in the previous chapters. The sample application

consists of three services, and for this chapter, a change is made where order service is deployed in a different namespace called **orders**, as shown in *Figure 14.5*:

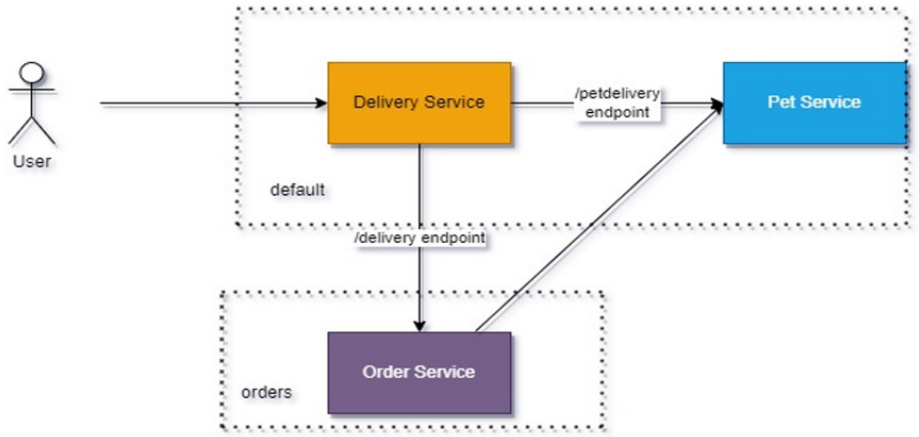


Figure 14.5: *Sample application with different namespaces*

As shown in the preceding image, the **/delivery** endpoint of the delivery service calls the order service, while the **/petdelivery** endpoint calls the pet service. Note that both are endpoints of the delivery service that call other services internally.

Initially, **istio-injection** is not enabled for orders **namespace**, so order service container will not have the Envoy proxy alongside it.

```
apiVersion: v1
kind: Namespace
metadata:
  name: orders
  labels:
    istio-injection: disabled
```

The **PeerAuthentication** is a custom Istio object that can be used to decide which sort of mutual TLS would be used for the communication between services.

Create a **PeerAuthentication** object at root namespace, which is mostly **istio-system**, and set the mode as STRICT, which should apply at the cluster. Following is a yaml to create the **PeerAuthentication** object with the default name in **namespace istio-system**:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
```

```
  namespace: istio-system
spec:
  mtls:
    mode: STRICT
```

The **`spec.mtls.mode`** value is set to STRICT, which means communication without TLS would not be allowed within the cluster.

Now try delivery service's **`/petdelivery`** endpoint. It should work successfully. However, delivery service's **`/delivery`** endpoint should fail as it tries to communicate with order service, which is in a different namespace where **`istio-injection`** is not enabled. So, the communication to order service pod is blocked by the envoy proxy in the delivery service pod. However, pet service being in the same namespace has envoy proxy with it, so communication with pet service is successful. The broken communication between services scenario is illustrated in *Figure 14.6*:

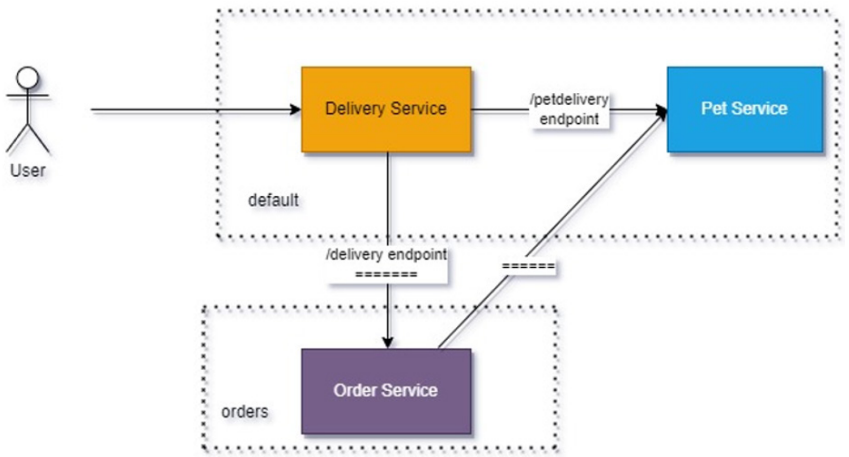


Figure 14.6: *Broken communication between services*

Now, to solve this problem, modify the **`peerauthentication`** object using the **`kubectl edit`** command and change the value of **`spec.mtls.mode`** from **`STRICT`** to **`PERMISSIVE`**. Save the object:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: istio-system
spec:
  mtls:
    mode: PERMISSIVE
```

Once the mode is changed to permissive, communication with order service will be allowed, and delivery service can communicate successfully to order service, as shown in *Figure 14.5*. Note that no change in the actual service code was required to enable and disable the TLS restriction for communication between the services.

Istio applies the policies for the workloads in the cluster in the following order of priority:

1. **`workload-specific`**
2. **`namespace-wide`**
3. **`mesh-wide`**

This means if we create a policy at namespace default, it should allow non-mTLS traffic in the namespace as well. You can verify this by taking the following two steps:

1. Modify the **`PeerAuthentication`** object at the **`istio-system`** namespace level to be STRICT. Thus, communication between delivery service and order service will be blocked again, as shown in *Figure 14.6*.
2. Then, create a **`PeerAuthentication`** object at default namespace level with permissive mode. Check the delivery service's **`/delivery`** endpoint again; it should work just as shown in *Figure 14.5*:

```
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
  namespace: default
spec:
  mtls:
    mode: PERMISSIVE
```

Let us discuss the benefits of permissive mode. The permissive mode of Istio mutual TLS allows the services in the cluster to accept encrypted and plain text traffic at the same time. This permissibility greatly improves the onboarding experience for mutual TLS. The newly added services to the cluster are tracked by the Istio, and once proxy is injected alongside them, they start sending mutual TLS traffic. For the services without sidecars, communication using plain text continues.

For a cluster that contains many services without istio proxies, it is not straightforward to migrate all the services to enable mutual TLS traffic. The operator may not have enough rights to install istio proxy in all namespaces or the installation may not

be possible for some services due to certain constraints. Moreover, after installing the Istio proxies on services, if mutual TLS is enabled by default, then existing communications will break. In situations like these, permissive mode helps seamless transition of non-istio services to istio enabled services. Because of permissive mode, the plain text traffic also continues to be accepted. The operator can gradually install and configure Istio sidecars to send mutual TLS traffic for all the services in the cluster. Once all services are tuned to mutual TLS traffic, the mode can be changed to **`STRICT`**, where only mutual TLS traffic will be accepted.

Look at *Figure 14.7*, which is a simplified diagram of Istio's authentication architecture, before we discuss authorization with Istio. Official documentation of Istio should be referred to for the latest diagram of authentication architecture.

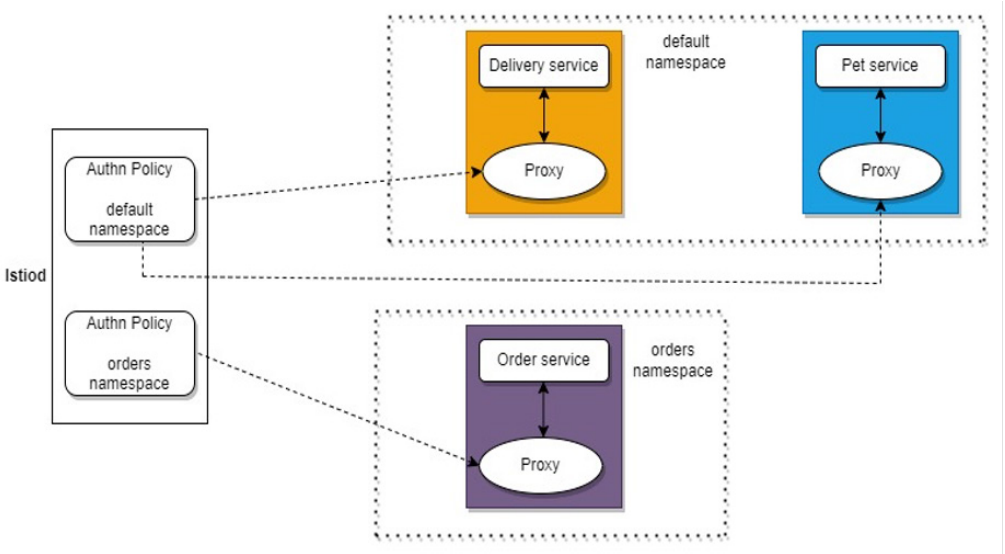


Figure 14.7: *Authentication Architecture with Istio*

Authorization with Istio

Let us look at a very small yaml specification of the **`AuthorizationPolicy`** object of Istio to begin the discussion on authorization policies:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz-allow-nothing
spec:
  action: ALLOW
```

This is an allow-nothing policy named **orders-authz**. This is because although the **spec.action** attribute says **ALLOW**; it is interpreted as allow-nothing because no rules are specified underneath for allowing. This may seem strange in the beginning, but it is a good security practice where the policy denies everything instead of allowing everything. This can be used as a base security policy to build less restrictive policies for your cluster.

Following is another version of the preceding policy, which is known as the deny-all policy:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: orders-authz-deny-all
spec:
  action: DENY
  rules:
  - {}
```

Here, the rules field is an empty array that will result in all rules being matched, and hence, denied as an outcome of policy enforcement. By looking at the two preceding policies, it should be clear that we can create an allow-all policy, but let us keep that for later.

Service authorization

Let us see another yaml where we want to allow traffic to a service only from services within the specified namespace. Refer to *Figure 14.5*, which shows that orders service is deployed in the **orders** namespace, and the other two services are deployed in a **default** namespace.

For the following exercise, **istio-injection** should be enabled in the orders namespace as well, and **istio proxy** should be installed alongside order service application container. Let us say you put an authorization policy like the following:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
 namespace: orders
spec:
```

```
  action: ALLOW
  rules:
    - from:
      - source:
          namespaces: ["orders"]
```

This will break the communication between delivery service and order service, and between order service and pet service, because the policy states that requests from only the **orders** namespace should be allowed. You can verify this by looking at the log for the **istio-proxy** container inside the orders service pod. Use the **kubectl** command as follows:

```
kubectl logs <ordersvc-deployment-pod-id> -c istio-proxy
```

You shall see error 403 in the log, indicating that access is unauthorized or forbidden. This is the Istio response indicating that user is authenticated or identified correctly but not authorized to access the service. Following is a snippet of the log:

```
"GET /order HTTP/1.1" 403 - rbac_access_denied_matched_policy[none]
- "-" 0 19 0 - "-" "python-requests/2.28.2" "orderservice.orders.svc.
cluster.local:8080" "-" inbound|8080|| - 10.1.2.74:8080 10.1.2.70:34380
outbound_.8080_._.orderservice.orders.svc.cluster.local –
```

This indicates that **istio-proxy** intercepts the incoming traffic to orders service and executes the authorization policy to allow or deny the request.

Similar logs are reflected in the logs for the **istio-proxy** container in delivery service:

```
kubectl logs <deliverysvc-deployment-pod-id> -c istio-proxy
```

This command shows something like the following:

```
"GET /order HTTP/1.1" 403 - via_upstream - "orderservice.orders.svc.
cluster.local:8080" "10.1.2.74:8080" outbound|8080||orderservice.orders.
svc.cluster.local 10.1.2.70:34380 10.109.22.103:8080 10.1.2.70:51454 –
default
```

Now, to fix the preceding problem where communication between delivery service and order service is broken, let us put another policy in place:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
```

```
 namespace: orders
spec:
  action: ALLOW
  rules:
    - from:
      - source:
          namespaces: ["default"]
      to:
      - operation:
          methods: ["GET"]
```

This policy is less restrictive as it allows requests from the **`default`** namespace to GET methods of the services in the **`orders`** namespace. While this policy allows specific operations, it is applicable to all the services in the **`orders`** namespace. You can use **`spec.selector`**, as shown in the following yaml, to allow or deny the requests to a specific service in the namespace:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
 namespace: orders
spec:
  selector:
    matchLabels:
      app: orderservice
  action: ALLOW
  rules:
    - from:
      - source:
          namespaces: ["default"]
      to:
      - operation:
          methods: ["GET"]
```

This policy applies only to services where label app has the **orderservice** value as specified under the **selector**. There is also an interesting **when** clause that can be used to specify **custom** conditions. Following is a yaml where **source.namespace** is used as a key under the **when** clause:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
 namespace: orders
spec:
  action: ALLOW
  rules:
    - when:
      - key: source.namespace
        values: ["default"]
```

The rule is passed when the namespace is **default**.

The Istio official documentation should be checked for more interesting keys like **source.ip**, **request.headers**, and **source.principal**. We will also discuss some of the custom keys related to **request.auth** in the next section on end user authentication.

It should be noted that AuthorizationPolicy object can be used for services using TCP as well. For example, if you want to restrict access to a MongoDB service running on a port to only specific services, you can create an Authorization policy for that too.

AuthorizationPolicy sample yaml:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: mongodb-access-policy
  namespace: default
spec:
 selector:
   matchLabels:
     app: mongodb
 action: ALLOW
```

```
 rules:
 - from:
   - source:
       principals: ["cluster.local/ns/default/sa/pet-service"]
   to:
   - operation:
       ports: ["27017"]
```

This policy allows connection to the MongoDB service on port **27017** from pet service's service account only.

Let us take it to where it all started about authorization policies. We looked at an allow-nothing policy at the beginning of our discussion. Following is an allow-all policy where all rules match and all requests are allowed:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: orders-authz-allow-all
spec:
  action: ALLOW
  rules:
  - {}
```

This is just to indicate one of the many possibilities about the policies that you can create and play around with, but of course, such anti-zero-trust policies are not advisable to be used.

End user authorization

If the source section is left empty in the authorization policy, that is equivalent to making the service publicly available, which is if the service is made available outside the Kubernetes cluster. To make sure only authenticated users access the service or workload, specify **source.principals** as *, as shown in the following yaml:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
```

```
 namespace: orders
spec:
  action: ALLOW
  rules:
    - from:
      - source:
          principals: ["*"]
```

To allow the access to only specific service accounts, you can use the authorization policy as follows, where requests from default service account in default namespace are only allowed:

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
 name: orders-authz
 namespace: orders
spec:
  action: ALLOW
  rules:
    - from:
      - source:
          principals: ["cluster.local/ns/default/sa/default"]
```

While the examples listed here are using one clause at a time like from, to, when and so on, it is quite a common practice to combine multiple clauses as per requirement.

It should be noted that the **`principals`**, **`notPrincipals`**, **`namespaces`**, and **`notNamespaces`** fields in the authorization policy depend on mutual TLS being enabled. Similarly, using **`source.principal`**, **`source.namespace`** under custom conditions like **`when`** requires mutual TLS to be enabled. This is because mutual TLS is what is used to securely pass information between different services in the cluster.

`RequestAuthentication` is another custom resource that aids security configurations when Istio is present in the cluster. It is used to define the request authentication methods supported by a service in the cluster. The configured authentication rules are used by the data plane to reject or accept an incoming request. **`RequestAuthentication`** works in tandem with **`AuthorizationPolicy`** to restrict access to authenticated users. Following is an example where the policy is

applied at the **istio-system** namespace requiring all the incoming requests to have a **JSON Web Token** (**JWT**) from a specified issuer:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: request-authn-default
  namespace: istio-system
spec:
  jwtRules:
  - issuer: "auth-issuer"
    jwksUri: https://xyz.com/.well-known/jwks.json
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt-default
  namespace: istio-system
spec:
  rules:
  - from:
    - source:
        requestPrincipals: ["*"]
```

You can use **spec.selector** fields like in the following yaml to apply the **RequestAuthentication** and **AuthorizatioPolicy** objects to a specific service in the cluster:

```
apiVersion: security.istio.io/v1beta1
kind: RequestAuthentication
metadata:
  name: request-authn-orders
  namespace: orders
spec:
  selector:
    matchLabels:
```

```
      app: orderservice
  jwtRules:
  - issuer: "auth-issuer"
    jwksUri: https://xyz.com/.well-known/jwks.json
---
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: require-jwt-default
  namespace: orders
spec:
  rules:
  - from:
    - source:
        requestPrincipals: ["*"]
```

The following snippet is an example of authorization policy where the issuer of the JWT is checked to have a specific value:

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: delivery-authz
  namespace: default
spec:
  selector:
    matchLabels:
      app: deliveryservice
  rules:
  - to:
    - operation:
        methods: ["GET"]
    when:
    - key: request.auth.claims[iss]
      values: ["mytrustedissuer@example.issuer.com"]
```

Issuer is a claim in the JWT indicating which entity has generated the token. The authorization policy puts a restriction to accept requests having JWTs only from trusted or known issuers. Thus, even for authenticated users, the request would fail if the token is not coming from trusted issuers.

Refer to *Figure 14.8* for a simplified diagram of authorization architecture supported with Istio:

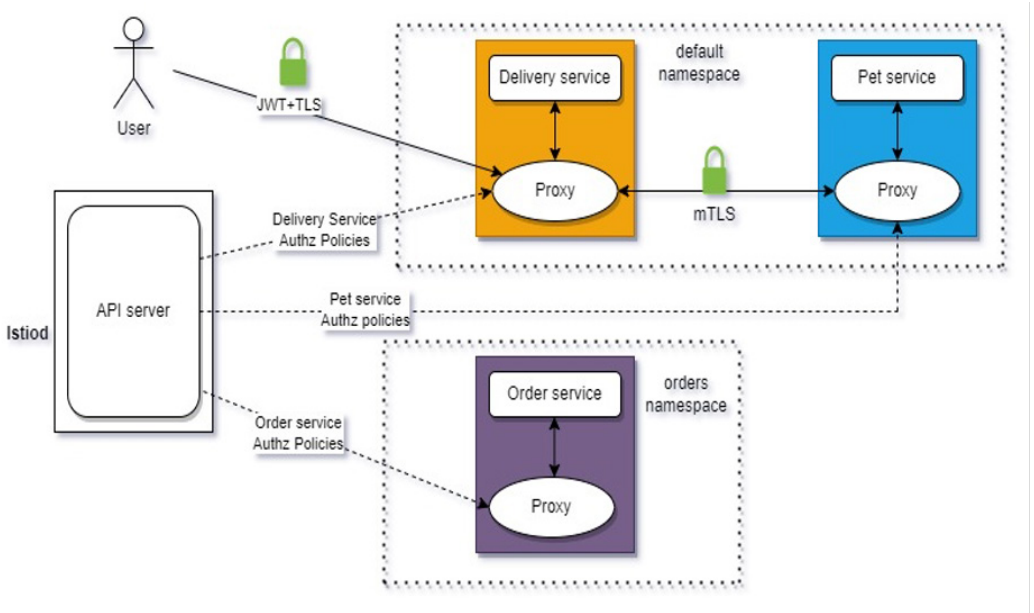


Figure 14.8: *Authorization architecture with Istio*

Security architecture of Istio

Let us bring it all together and try to look at the bigger picture containing all components enabling security using Istio. *Figure 14.9* is an oversimplified depiction of the security architecture of Istio, and the official documentation of Istio shall also be referred to get another view of the things:

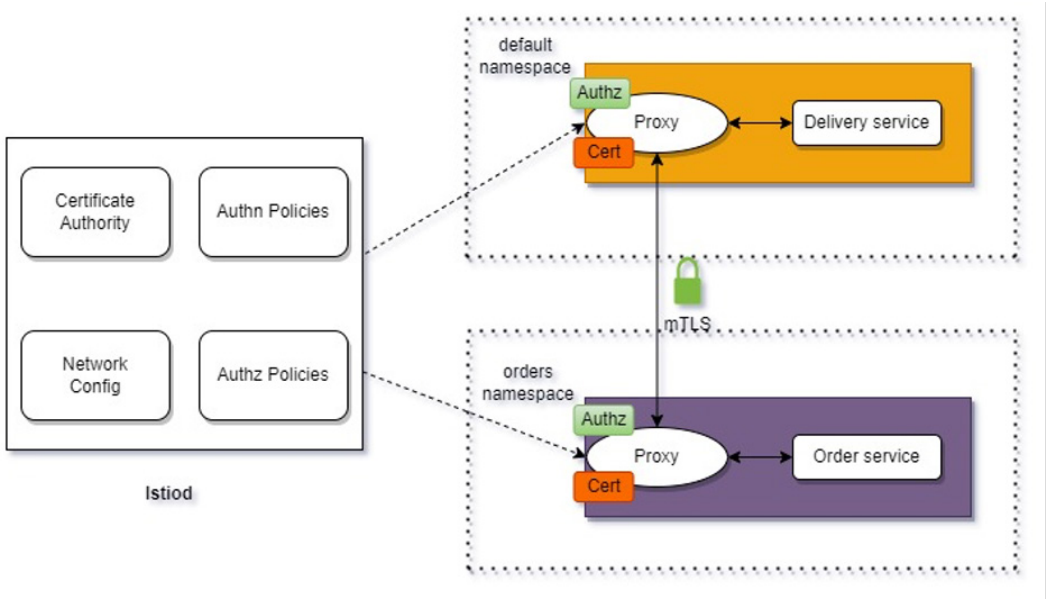


Figure 14.9: *Security Architecture with Istio*

Here, Istiod is the control plane which distributes the authorization and authentication policies to the proxies. Istiod also contains the certificate authority for key and certificate management. The proxies inside the application pods act as policy enforcement points and control the traffic. The policies are created and managed in control plane using an API server, and the Envoy proxies implement these policies.

Conclusion

The security architecture of Istio helps enable Zero-Trust security for the services in your application. Things like automated rotation of certificates and keys are taken care of by the control plane of Istio, freeing up operation teams to do other useful tasks. With an end-user JSON Web token, you can specify the rules to allow access to certain operations based on the contents of that token, which could be a well-known property like the issuer or the subject or any custom claim that the token has.

`PeerAuthentication`, **`RequestAuthentication`** and **`AuthorizationPolicy`** are custom objects created in Kubernetes cluster to enable security policies in the cluster. The proxies installed with the application containers act as policy enforcement points to implement these policies.

Points to remember

- Authentication, authorization, and encrypted traffic are the three pillars of security in your application, and Istio provides support for all of them.
- HTTPS uses a TLS certificate in the server to encrypt traffic and confirm the identity of the caller, while mutual TLS also uses a client certificate for authentication.
- **`PeerAuthentication`** is used for service-to-service authentication, while **`RequestAuthentication`** is for authenticating end users.
- **`PeerAuthentication`** is the custom object used to define the policies related to TLS in the sidecar for service-to-service communication.
- Operators should use permissive mode of mutual TLS to gradually migrate workloads or services in a non-istio cluster to istio enabled cluster.
- **`RequestAuthentication`** is the object used to define the rules for jwt issuer and the key server.
- You can control access to services in Istio using the **`AuthorizationPolicy`** object, where you can define the allowed principles and the allowed operations for them.
- You can start by creating a deny-all or an allow-nothing policy in your cluster and then add restrictions on top of it.

Questions

1. What is the order in which authentication policies are applied?
2. What is secure naming, and what role does it play in mTLS authentication?
3. How do you migrate an existing app onto Istio and enforce mutual TLS?
4. How can you put restrictions based on issuer for Json Web Tokens (JWTs)?

Answers

1. From narrow scope to wide scope that is port-level policies are applied before service- level policies. Service-level policies are applied before namespace-level policies, and namespace-level policies are applied before mesh-level policies.
2. Secure naming helps map services to the identities that are authorized to run those services. This mapping helps avoid spoofing attacks where even

though traffic is diverted to a malicious server, clients can avoid damage by having proper authorization checks in place.

3. It will be a staged effort where all clients of the services are required to be upgraded to support mutual TLS first, and then the services will support only mutual TLS.

4. Use a custom condition in the **`AuthorizationPolicy`** object to check the value for **`request.auth.claims[iss]`**. Apply this **`AuthorizationPolicy`** object in the namespace for the service using selector attributes.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com



Index

A

advanced details 161
  cluster IP environment variables 164
  endpoints 161-163
  manual service discovery 163
advanced Ingress 64
  multiple Ingress, managing 64
  multiple Ingress, running 64
  namespaces 64
  path rewriting 64, 65
  TLS, serving 65, 66
annotations 119
  sample situations 119
Apache Zookeeper 151
API gateways 68
  cross-cutting concerns 69
  need for 68
  protocols, translating 69
  requests, routing 69
authentication, with Istio
  mutual TLS authentication 321, 322
  peer authentication 321
  peer authentication, with sample application 323-327
  request authentication 321
  secure naming 322, 323
authorization, with Istio 327, 328
  end user authorization 332-336
  service authorization 328-332
auto scaling 231, 232

B

Burn Rate threshold 202

C

circuit breaking 288
client-side discovery pattern 148, 149
cluster autoscaling 242-244
cluster IPs 164, 165
ConfigMap 99-102
  command-line arguments, setting 106, 107
  consuming 104, 105
  consuming, in environment variables 105, 106
  consuming, via volume plugin 107, 108
  creating 102-104
ConfigMaps and Secrets
  creating 114
  listing 113, 114
  managing 113
  updating 114, 115
Consul 151
Container Network Interface (CNI) 51
  configuring 54, 55
Container Runtime Environment (CRE) 4
Container Runtime Interface (CRI) 51
Container states
  running 32
  terminated 32, 33
  waiting 32
Container Storage Interface (CSI) 141, 142
Contour
  installing 59
CronJob 95
custom metric scaling 247-249

D

DaemonSets 87
  creating 87-89
  deleting 91
  restricting, to specific nodes 89, 90
  updating 90, 91
data persistence 38, 39
  data, on remote disks 41, 42
  data volumes, using with PODs 39-41
Denial of Service (DoS) 171
deployments 81
  creating 82, 83
  deleting 87
  managing 83
  status, monitoring 86
  strategies 86
  updating 83-86
destination hashing (DH) 54
disaster recovery 140, 141
distributed tracing 308-313
Docker 1, 2, 3
  commands 4, 5
  container runtime 8
  images, storing in registries 7
  multistage images, building 6, 7
  process 4, 5
Docker 101 3

E

equality-based selector 117
error budget 198
etcd 151

G

Grafana
  installing 208-211
Grafana dashboard
  creating, on metrics 213, 214

H

health checks 42
  liveness probe 43
  readiness probe 43, 44
  startup probe 42
HoriziontalPodScalar 233
Horizontal Pod Autoscaling (HPA) 234
  limitations 239
horizontal pod scaling 233-237
  metric threshold 237-239

I

Identity management, with Istio 318, 319
  certification generation process 319, 320
  identity verification, in TLS 319
Incident 222
Ingress
  specifications 55-59
Ingress controller 55-59
  alternate implementations 66-68
Ingress usage 62
  hostnames, utilizing 62, 63
  paths, utilizing 63
IP Virtual Server (IPVS) 158
Istio 260
  characteristics 260
  dashboards, monitoring with Grafana 305-308
  installing 263, 264
  installing, with istioctl command 264-266
  metrics, querying with Prometheus 303-305
  security architecture 336, 337
Istio architecture 261
  control plane 263
  data plane 262
istioctl operator init command 264
Istiod 263
Istio setup
  customizing 268, 269

J

Jaeger
  installing 218

K

K8S 9
Kiali 297
  dashboard 299
Kubeproxy 164, 165
  configuring 53, 54
Kubernetes 1, 8, 9
  architecture 10, 11
  declarative configurations 16, 17
  features 9
  installing 17, 18
  installing, in Docker 19
  installing locally, with Minikube 18, 19
  principles of immutability 16
  self-healing systems 17
  storage, configuring with 123
Kubernetes client 19, 20

status of Kubernetes Master Daemons, checking 20
version, checking 20
worker nodes, listing 21-23
Kubernetes Jobs 92
creating 92, 93
CronJobs 95, 96
finished jobs, cleaning automatically 94, 95
patterns 94
Pod and container failures 94
Kubernetes Master 11, 12
components 12, 13
Kubernetes observability 197
alerts, creating 201-204
challenges 205, 206
error budgets, tracking 200, 201
metrics, for SLIs 199
pillars 204, 205
probes and uptime checks 202
SLO, setting 200
Kubernetes security challenges 171
Kubernetes Worker 14, 15
best practices 15
Kubernetes Workload Resources 75

L

labels
applying 115
modifying 115, 116
labels selectors 117
equality-based selector 117
role, in Kubernetes architecture 118
set-based selector 118
least connection (lc) 54
logging 214
with Fluentd 215-217
lookback duration 201

M

metrics
exploring, with Prometheus and Grafana 206
metrics, for SLIs
availability 199
correctness 199
data freshness 199
error rate 199
latency 199
selecting 199
microservices scaling 228, 229
challenges 230, 231
principles 229, 230
MongoDB
installing on Kubernetes, StatefulSets used 138-140
monitoring tools
selecting 224
mutual TLS authentication 321, 322

N

Network Address Translation (NAT) 52
Networking 101 50-53
network security 69
best practices 71, 72
securing, via network policies 69, 70
securing, via third-party tool 70, 71
never queue (nq) 54
Nginx Controller
installing 60, 61

O

observability, with Istio 293
  sample application and proxy logs 295, 296
Open Telemetry Library (OTel) 217

P

pillars, Kubernetes observability
  logs 205
  metrics 204
  tracing 205
  visualization 205
Pods 27, 29
  accessing 34
  accessing, via port forwarding 34, 35
  commands, running with exec 35
  creating 30
  CRUD operations 30
  deleting 33, 34
  Failed phase 32
  listing 31
  logs, accessing 36
  Pending phase 32
  running 30, 31
  Running phase 32
  Succeeded phase 32
  Unknown phase 32
POD security 44, 45
  admissions 46
  standards 45
Prometheus 206
  custom metrics, pushing to 211-213
  installing 207-211

R

registration patterns 151
  self-registration pattern 152
  third-party registration 152
ReplicaSets 77
  creating 78, 79
  deleting 81
  designing 77, 78
  inspecting 79
  scaling 79, 80
RequestAuthentication 333
resources
  managing 36
  requests 36-38
role-based access control (RBAC) 173
  cluster roles, aggregating 178, 179
  identity 173
  managing 177, 178
  role bindings 174-177
  user groups, for bindings 179
round-robin (rr) 54

S

scaling, in Kubernetes 232
  best practices 249
  cluster autoscaling 242-244
  custom metric scaling 247-249
  horizontal pod scaling 233- 237
  standard metric scaling 244-247
  vertical pod scaling 239-242
Secrets 109
  consuming 111
  consuming, environment variables 112

creating 109, 110
mounted as volume, consuming 111, 112
private docker registries 112, 113
secure naming 322
security architecture, Istio 336, 337
self-registration pattern 152
server-side discovery pattern 150
service discovery 146-148
client-side discovery pattern 148, 149
server-side discovery pattern 150
service discovery, in Kubernetes 153
DNS 160
readiness checks 160, 161
service objects 159, 160
via etcd 153-156
via Kuberproxy and DNS 157, 158
service level agreement (SLA) 8, 198
service level indicators (SLI) 198
service level objectives (SLO) 197, 198
Service Mesh 257-259
benefits 254, 255
control plane performance and resource consumption 267, 268
cost 267
data plane performance and resource consumption 267
failure, recovering from 256, 257
load balancing, of traffic 256
metrics, collecting 256
service discovery 256
traffic, monitoring between services 256
visualizing, with Kiali 297-302
service registry 151
examples 151
set-based selector 118
shortest expected delay (sed) 54
Site Reliability Engineering (SRE)
drill 223
incident management 222, 223
Playbook maintenance 223
responsibilities 221, 222
team 197
source hashing (sh) 54
SRE drills 201
SRE process
best practices 220, 221
defining 220
standard metric scaling 244-247
StatefulSets 133
headless service 137
properties 133-137
volume claim templates 137
storage provisioning, Kubernetes 124
persistent volume claims 125-130
persistent volumes 125
storage class 130-132
StorageClass, for dynamic provisioning 132, 133
volumes 124, 125
strategies, cluster quality validation 23
cost-efficiency, as measure of quality 23
data transfer costs and network costs 24, 25
persistent volumes 24
request and restrict specifications for pod CPU and memory resources 24

right nodes 24
security, as measure of quality 25

T

telemetry flow 294, 295
third-party registration 152
threat
insider threat 172
malicious actor infrastructure 171, 172
supply chain threat 171
tracing 214
with Open Telemetery, Jaeger used 217-219
traffic management, with Istio 273
circuit breaking 289, 290
faults, injecting for testing 284-286
Ingress and Egress traffic, controlling 279, 280
retries 286-288
timeouts 286-288
traffic shifting, between versions 280-284
via gateways 274-276
virtual service and destination rules 276-278

V

Vertical Pod Autoscalar (VPA) 239, 240
vertical pod scaling 239-242
visualization tools
selecting 224
volumeBindingMode 132

Z

Zero Trust Architecture (ZTA) 180
recommendations, for application security practices 187
recommendations, for auditing and threat detection 187
recommendations, for authentication and authorization 186
recommendations, for Kubernetes network security 185, 186
recommendations, for Kubernetes Pod security 182-185
requirements 180-182
Zero Trust, in Kubernetes 188
identity-based service to service access 188
Kubernetes encryption 189, 190
observability, with audits and logging 190, 191
secret and certificate management 189, 190
service-to-service communication 188, 189

Hands-On Kubernetes, Service Mesh and Zero-Trust

DESCRIPTION

Building and managing secure applications is a crucial aspect of modern software development, especially in distributed environments. Kubernetes and Istio, when combined, provide a powerful platform for achieving application security and managing it effectively. If you want to build and manage secure applications with ease, then this book is an ideal resource for you.

The book extensively covers the building blocks of the Kubernetes orchestration engine, providing in-depth insights into key Kubernetes objects that can be effectively used for deploying containerized applications. It then focuses on all major Kubernetes constructs, offering guidance on their appropriate utilization in different scenarios, while emphasizing the significance of a Zero Trust architecture. Additionally, the book addresses important aspects such as service discovery, optimized logging, and monitoring, which play a critical role in managing distributed applications. It also incorporates essential concepts from Site Reliability Engineering and enabling engineering teams, to proactively meeting Service Level Agreements and attaining operational readiness. In the final section, the book takes a deep dive into Service Mesh, with a special focus on harnessing the strength of Istio as a powerful tool.

By the end of the book, you will have the knowledge and skills to effectively build, deploy, and manage secure applications using Kubernetes and Istio.

KEY FEATURES

- Delve into security practices that guarantee resilience and secure deployments.
- Discover strategies for managing Kubernetes clusters, enhancing performance, and achieving high availability and scalability.
- Acquire a conceptual understanding of the challenges faced in production environments and explore industry-standard solutions for efficient resolution.

WHAT YOU WILL LEARN

- Learn how to successfully deploy applications on Kubernetes.
- Gain insights into the principles of Zero Trust architecture and its implementation within the Kubernetes orchestration platform.
- Get familiar with the concepts of service discovery and efficient scaling in Kubernetes, empowering you to optimize your application deployments.
- Learn about monitoring and logging within applications, and explore the essential aspects of observability to ensure the reliability of your systems.
- Acquire expertise in service mesh, particularly Istio, to efficiently handle traffic, enhance application reliability, and fortify security measures.

WHO THIS BOOK IS FOR

This book caters to a wide range of readers, including developers utilizing Kubernetes, DevOps teams, senior software engineers, cloud-native teams, and cloud developers with a foundational knowledge of containers and software



BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-867-5

