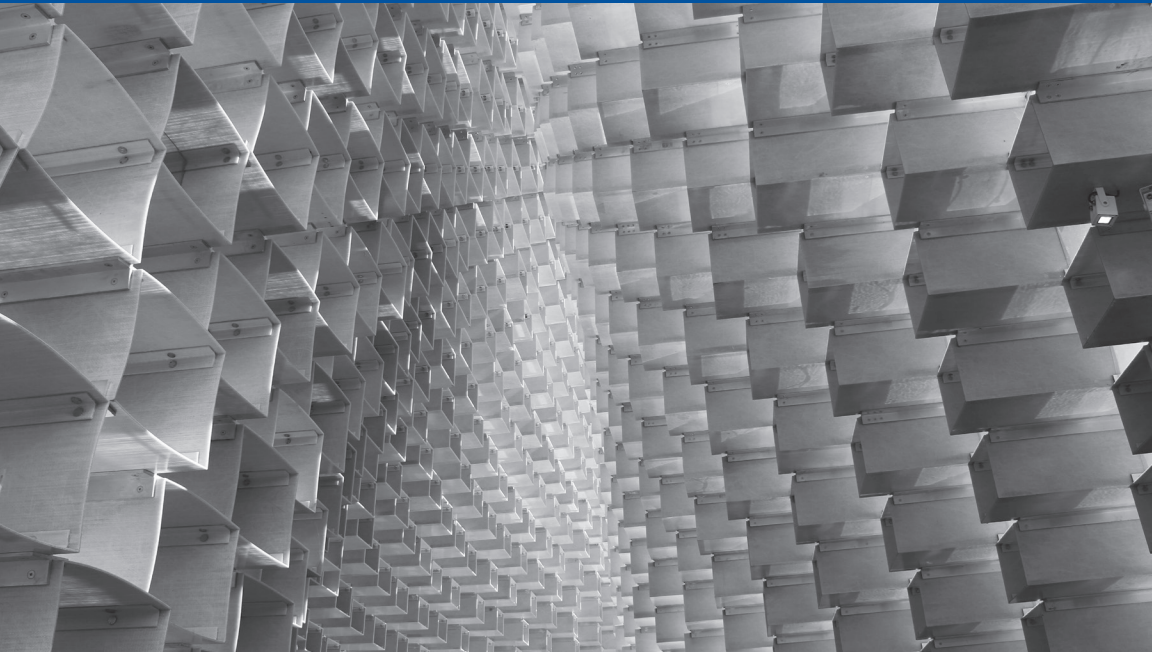


O'REILLY®

Compliments of


Kubernetes in the Enterprise

Deploying and Operating Production
Applications on Kubernetes in
Hybrid Cloud Environments



Michael Elder, Jake Kitchener
& Dr. Brad Topol

Build

Kubernetes makes it easy to bind your app to Watson, by relieving the pain around security, scale, and infrastructure management.

Get hands-on experience through tutorials and courses.

ibm.biz/oreillykubernetes

Smart



Kubernetes in the Enterprise

*Deploying and Operating Production
Applications on Kubernetes in
Hybrid Cloud Environments*

*Michael Elder, Jake Kitchener,
and Dr. Brad Topol*

Kubernetes in the Enterprise

by Michael Elder, Jake Kitchener, and Dr. Brad Topol

Copyright © 2018 O'Reilly Media. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://oreilly.com/safari>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Nicole Tache and Michele Cronin

Production Editor: Melanie Yarbrough

Copyeditor: Octal Publishing, LLC

Proofreader: Sonia Saruba

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Rebecca Demarest

October 2018: First Edition

Revision History for the First Edition

2018-09-28: First Release

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Kubernetes in the Enterprise*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-04324-9

[LSI]

To Wendy, for your love and encouragement. You will forever be “unforgettable in every way” to me. To Samantha, for your fearlessness and curiosity about all things in life. To David, for your inspirational smile and laughter. To my mother, Betty, for your amazing tenacity through all of life’s challenges while remaining optimistic about the future.

—Michael Elder

Great thanks go to my wife, Becky, for her love and support. To Oren goes my gratitude for his laughter and caring spirit. Thank you to my parents Nancy and Barry Kitchener: without their example I would not have the tenacity to take on the trials of life.

—Jake Kitchener

I dedicate this book to my wife, Janet; my daughter, Morgan; my son, Ryan; and my parents, Harold and Mady Topol. I could not have done this without your love and support during this process.

—Brad Topol

Table of Contents

Foreword.....	ix
Preface.....	xi
1. An Introduction to Containers and Kubernetes.....	1
The Rise of Containers	1
Kubernetes Arrives to Provide an Orchestration and Management Infrastructure for Containers	4
The Cloud Native Computing Foundation Tips the Scale for Kubernetes	6
CNCF Kubernetes Conformance Certification Keeps the Focus on User Needs	7
Summary	8
2. Fundamental Kubernetes Topics.....	9
Kubernetes Architecture	9
Let's Run Kubernetes: Deployment Options	12
Kubernetes Core Concepts	14
3. Advanced Kubernetes Topics.....	29
Kubernetes Service Object: Load Balancer Extraordinaire	29
DaemonSets	31
StatefulSets	33
Volumes and Persistent Volumes	36
ConfigMaps	40
Secrets	44
Image Registry	47

Helm	49
Next Steps	51
4. Introducing Our Production Application.	53
Our First Microservice	53
Namespaces	55
ServiceAccount	56
PodSecurityPolicy	57
Deploying a Containerized Db2 Database as a StatefulSet	57
Managing Our Portfolio Java-Based Microservice as a Deployment	74
Deploying the trader Microservice Web Frontend	79
Deploying a Containerized MQ Series Manager as a StatefulSet	81
Deploying Supporting Services for the portfolio Microservice	82
Putting It All together: Accessing Our Fully Configured Application	85
Summary	89
5. Continuous Delivery.	91
Image Build	92
Programmability of Kubernetes	94
General Flow of Changes	94
6. Enterprise Application Operations.	97
Log Collection and Analysis for Your Microservices	97
Health Management for Your Microservices	102
Summary	108
7. Cluster Operations and Hybrid Cloud.	109
Hybrid Cloud Overview	109
Access Control	110
Performance, Scheduling, and Autoscaling	116
Networking	123
Storage	131
Quotas	132
Audit and Compliance	135
Kubernetes Federation	136

8. Contributor Experience.....	137
Kubernetes Website	137
The Cloud Native Computing Foundation Website	138
IBM Developer Website	139
Kubernetes Contributor Experience SIG	140
Kubernetes Documentation SIG	141
Kubernetes IBM Cloud SIG	142
9. The Future of Kubernetes.....	143
Increased Migration of Legacy Enterprise Applications to Cloud-Native Applications	143
Increased Adoption of Kubernetes for High-Performance Computing	144
Kubernetes Will Become the de Facto Platform for Machine Learning and Deep Learning Applications	145
Kubernetes Will Be the Platform for Multicloud	145
Conclusions	145
A. Configuring Kubernetes as Used in This Book.....	147
B. Configuring Your Development Environment.....	151
C. Configuring Docker to Push or Pull from an Insecure Registry.....	153
D. Generating an API Key in Docker Cloud.....	155

Foreword

Welcome to *Kubernetes in the Enterprise*.

Great technologies come in many guises. Some start small. They can be created by just one person, quietly working alone to solve a specific problem in a personal way. Ruby on Rails and Node.js are two examples that exceeded their creator's wildest dreams. Other technologies make an immediate impact. The rarest of these win widespread support in just a few years—in a blink of an eye in our industry. Kubernetes and containers are such a technology. They represent a fundamental shift in the industry platform—as critical as HTTP and Linux.

For the first time since the 1990s an entire industry, from vendors to enterprises to individuals, is pushing one platform forward and we don't even know exactly what it means yet. The only thing we can expect is to be surprised. New businesses, practices, and tools will emerge—this is a wonderful time to build something new. Take your pick—connected cars, digital homes, healthtech, farmtech, drones, on-demand construction, blockchain—the list is long and growing.

People will use these technologies, and they will be built on the new cloud native tools appearing around Kubernetes. Containers will help you streamline your application footprint, transform it to cloud readiness, and adopt new architectures like microservices. Practices like GitOps will speed up your continuous delivery and observability.

This change is a tremendous opportunity for big businesses to transition to new digital platforms and markets.

Not for the first time, IBM is at the forefront of this change, in projects such as Istio, etcd, Service Catalog, Cloud Foundry, and of course, Kubernetes. I've personally worked with the authors to spearhead adoption of Kubernetes and the Cloud Native Computing Foundation that is its home. You are in the hands of experts here—a team who have been leaders in the open source community as well as put in the hard yards with real world deployments at scale.

In this book you will find that knowledge presented as a set of patterns and practices. Every business can apply these patterns to create a production-grade cloud-native platform with Kubernetes at the core. Reader, the applications are up to you—an exciting world is just around the corner.

— *Alexis Richardson*
CEO, Weaveworks
TOC Chair, Cloud Native
Computing Foundation

Preface

Kubernetes is a cloud infrastructure that provides for the deployment and orchestration of containerized applications. The Kubernetes project is supported by a very active open source community that continues to experience explosive growth. With support from all the major vendors and the myriad contributors of all sizes, Kubernetes has established itself as the de facto standard for cloud-native computing applications.

Although Kubernetes has the potential to dramatically improve the creation and deployment of cloud-native applications in the enterprise, getting started with it in enterprise environments can be difficult. This book is targeted toward developers and operators who are looking to use Kubernetes as their primary approach for creating, managing, deploying, and operating their container-based cloud-native computing applications.

The book is structured so that developers and operators who are new to Kubernetes can use it to gain a solid understanding of Kubernetes fundamental concepts. In addition, for experienced practitioners who already have a significant understanding of Kubernetes, this book provides several chapters focused on the creation of enterprise-quality Kubernetes applications in private, public, and hybrid cloud environments. It also brings developers and operators up to speed on key aspects of production-level cloud-native enterprise applications such as continuous delivery, log collection and analysis, security, scheduling, autoscaling, networking, storage, audit, and compliance. Additionally, this book provides an overview of several helpful resources and approaches that enable you to quickly become a contributor to Kubernetes.

Chapter 1 provides an overview of both containers and Kubernetes. It then discusses the Cloud Native Computing Foundation (CNCF) and the ecosystem growth that has resulted from its open governance model and conformance certification efforts. In **Chapter 2**, we provide an overview of Kubernetes architecture, describe several ways to run Kubernetes, and introduce many of its fundamental constructs including Pods, ReplicaSets, and Deployments. **Chapter 3** covers more advanced Kubernetes capabilities such as load balancing, volume support, and configuration primitives such as ConfigMaps and Secrets, StatefulSets, and DaemonSets. **Chapter 4** provides a description of our production application that serves as our enterprise Kubernetes workload. In **Chapter 5**, we present an overview of continuous delivery approaches that are popular for enterprise applications. **Chapter 6** focuses on the operation of enterprise applications, examining issues such as log collection and analysis and health management of your microservices. **Chapter 7** provides in-depth coverage of operating Kubernetes environments and addresses topics such as access control, autoscaling, networking, storage, and their implications on hybrid cloud environments. We offer a discussion of the Kubernetes developer experience in **Chapter 8**. Finally, in **Chapter 9**, we conclude with a discussion of areas for future growth in Kubernetes.

Acknowledgments

We would like to thank the entire Kubernetes community for its passion, dedication, and tremendous commitment to the Kubernetes project. Without the code developers, code reviewers, documentation authors, and operators contributing to the project over the years, Kubernetes would not have the rich feature set, strong adoption, and large ecosystem it has today.

We would also like to thank our Kubernetes colleagues, Zach Corleissen, Steve Perry, Joe Heck, Andrew Chen, Jennifer Randeau, William Dennis, Dan Kohn, Paris Pittman, Jorge Castro, Guang Ya Liu, Sahdev Zala, Srinivas Brahmaraoutu, Morgan Bauer, Doug Davis, Michael Brown, Chris Luciano, Misty Linville, Zach Arnold, and Jonathan Berkahn for the wonderful collaboration over the years.

We also extend our thanks to John Alcorn and Ryan Claussen, the original authors of the example Kubernetes application we use as an exemplar in the book. Also, we would like to thank Irina Delidj-kova for her review and wisdom for all things Db2.

A very special thanks to Angel Diaz, Todd Moore, Vince Brunssen, Alex Tarpinian, Dave Lindquist, Willie Tejada, Bob Lord, Jake Morlock, Peter Wassel, Dan Berg, Jason McGee, Arvind Krishna, and Steve Robinson for all of their support and encouragement during this endeavor.

— *Michael, Jake, and Brad*

An Introduction to Containers and Kubernetes

In this first chapter, we begin with a historical background of the origin of both containers and Kubernetes. We then describe the creation of the Cloud Native Computing Foundation and the role it has played in the explosive growth of Kubernetes and its ecosystem. We conclude this chapter with an overview of Kubernetes Conformance Certification initiatives, which are critical to ensuring Kubernetes interoperability, supporting portable workloads, and maintaining a cohesive open source ecosystem.

The Rise of Containers

In 2012, the foundation of most cloud environments was a virtualization infrastructure that provided users with the ability to instantiate multiple virtual machines (VMs). The VMs could attach volume storage and execute on cloud infrastructures that supported a variety of network virtualization options. These types of cloud environments could provision distributed applications such as web service stacks much more quickly than was previously possible. Before the availability of these types of cloud infrastructures, if an application developer wanted to build a web application, they typically waited weeks for the infrastructure team to install and configure web servers and database and provide network routing between the new machines. In contrast, these same application developers could uti-

lize the new cloud environments to self-provision the same application infrastructure in less than a day. Life was good.

Although the new VM-based cloud environments were a huge step in the right direction, they did have some notable inefficiencies. For example, VMs could take a long time to start, and taking a snapshot of the VM could take a significant amount of time as well. In addition, each VM typically required a large number of resources, and this limited the ability to fully exploit the utilization of the physical servers hosting the VMs.

At Pycon in March of 2013, Solomon Hykes **presented an approach for deploying web applications to a cloud** that did not rely on VMs. Instead, Solomon demonstrated how Linux containers could be used to create a self-contained unit of deployable software. This new unit of deployable software was aptly named a *container*. Instead of providing isolation at a VM level, isolation for the container unit of software was provided at the process level. The process running in the container was given its own isolated file system and was allocated network connectivity. Solomon announced that the software they created to run applications in containers was called *Docker*, and would be made available as an open source project.

For many cloud application developers that were accustomed to deploying VM-based applications, their initial experience with Docker containers was mind-blowing. When using VMs, deploying an application by instantiating a VM could easily take several minutes. In contrast, deploying a Docker container image took just a few seconds. This dramatic improvement in performance was because instantiating a Docker image is more akin to starting a new process on a Linux machine. This is a fairly lightweight operation, especially when compared to instantiating a whole new VM.

Container images also showed superior performance when a cloud application developer wanted to make changes to a VM image and snapshot a new version. This operation was typically a very time-consuming process because it required the entire VM disk file to be written out. With Docker containers, a multilayered filesystem is used instead. If changes are made in this situation, they are captured as changes to the filesystem and represented by a new filesystem layer. Because of this, a Docker container image could snapshot a new version by writing out only the changes to the filesystem as a new filesystem layer. In many cases, the amount of changes to the

filesystem for a new container image are quite small and thus the snapshot operation is extremely efficient. For many cloud application developers who started experimenting with containers, it quickly became obvious that this new approach had tremendous potential to improve the current state of the art for deploying applications in clouds.

There was still one issue holding back the adoption of container images: the perception that it was not possible to run enterprise middleware as container images. Advanced prototyping initiatives took place to investigate the difficulty of running these images. It was proven quickly that developers could successfully run enterprise middleware such as WebSphere Liberty, and Db2 Express as Docker container images. Sometimes, a few changes were necessary or perhaps a Linux kernel upgrade was required, but in general the Docker container image approach was proven to be suitable for running enterprise middleware.

The container approach for deploying web applications experienced significant growth in a short period, and it was **soon supported on a variety of cloud platforms**. Here is a summary of the key advantages of using the container-image approach over VM images for deploying software to cloud-based environments:

Container image startup is much faster than VM image startup

Starting a container image is essentially the equivalent of starting a new process. In contrast, starting a VM image involves first booting an operating system (OS) and related services and is much more time consuming,

Capturing a new container image snapshot is much faster than a VM snapshot operation

Containers utilize a layered filesystem and any changes to the filesystem are written as a new layer. With container images, capturing a new snapshot of the container image requires writing out only the new updates to the filesystem that the process running in the container has created. When performing a snapshot of a VM image instance, the entire VM disk file must be written out, and this is typically an extremely time-consuming process.

Container images are much smaller than VM images

A typical container image is portrayed in megabytes, whereas a VM image is most commonly portrayed in gigabytes.

Build once, run anywhere

Docker enabled developers to build container images on their laptops, test them, and then deploy to the cloud knowing that not only the same code would be running in the cloud, but the entire runtime would be a bit-for-bit copy. Oftentimes with virtualization and traditional Platform as a Service (PaaS), developers test on one runtime configuration on their local system but don't have control over the cloud runtime. This leads to reduced confidence and more test requirements.

Better resource utilization

Because container images are much smaller in size and are at the process level, they take up fewer resources than a VM. As a result, it is possible to put a larger number of containers on a physical server than is possible when placing VMs on a physical server.

In the next section, we provide a background on Kubernetes, which is a platform for the management and orchestration of container images.

Kubernetes Arrives to Provide an Orchestration and Management Infrastructure for Containers

As previously discussed, Docker was responsible for introducing developers to the concept of container-based applications. Docker provided very consumable tooling for container development and storage of containers in registries. However, Docker was not the only company with experience using container-based applications in cloud environments.

For more than a decade, Google had embraced the use of Linux containers as the foundation for applications deployed in its cloud.¹ Google had extensive experience orchestrating and managing containers at scale and had developed three generations of container management systems: **Borg**, **Omega**, and **Kubernetes**. Kubernetes was the latest generation of container management developed by

¹ Brendan Burns et al., “**Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade**”. *ACM Queue* 14 (2016): 70–93.

Google. It was a redesign based upon lessons learned from Borg and Omega, and was made available as an open source project. Kubernetes delivered several key features that dramatically improved the experience of developing and deploying a scalable container-based cloud application:

Declarative deployment model

Most cloud infrastructures that existed before Kubernetes was released provided a procedural approach based on a scripting language such as Ansible, Chef, Puppet, and so on for automating deployment activities. In contrast, Kubernetes used a declarative approach of describing what the desired state of the system should be. Kubernetes infrastructure was then responsible for starting new containers when necessary (e.g., when a container failed) to achieve the desired declared state. The declarative model was much clearer at communicating what deployment actions were desired, and this approach was a huge step forward compared to trying to read and interpret a script to determine what the desired deployment state should be.

Built-in replica and autoscaling support

In some cloud infrastructures that existed before Kubernetes, support for replicas of an application and providing autoscaling capabilities were not part of the core infrastructure and, in some cases, never successfully materialized. These capabilities were provided as core features in Kubernetes, which dramatically improved the robustness and consumability of its orchestration capabilities.

Improved networking model

Kubernetes mapped a single IP address to a *Pod*, which is Kubernetes' smallest unit of container aggregation and management. This approach aligned the network identity with the application identity and simplified running software on Kubernetes.²

² Brendan Burns et al., “Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade”. *ACM Queue* 14 (2016): 70–93.

Built-in health-checking support

Kubernetes provided container health checking and monitoring capabilities that reduced the complexity of identifying when failures occur.

Even with all the innovative capabilities available in Kubernetes, enterprise companies were still reticent to adopt a technology that is an open source project supported by a single vendor, especially when other alternatives for container orchestration such as Docker Swarm were available. Enterprise companies would have been much more willing to adopt Kubernetes if it were instead a multiple-vendor and meritocracy-based open source project backed by a solid governance policy and a level playing field for contributing. In 2015, the Cloud Native Computing Foundation was formed to address these issues.

The Cloud Native Computing Foundation Tips the Scale for Kubernetes

In 2015, the Linux Foundation initiated the creation of the Cloud Native Computing Foundation (CNCNF).³ The CNCNF's mission is to create and drive the adoption of a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self-healing multitenant nodes.⁴ In support of this new foundation, Google donated Kubernetes to the CNCNF to serve as its seed technology. With Kubernetes serving as the core of its ecosystem, the CNCNF has grown to more than 250 member companies, including Google Cloud, IBM Cloud, Amazon Web Services (AWS), Docker, Microsoft Azure, Red Hat, VMware, Intel, Huawei, Cisco, Alibaba Cloud, and many more.⁵ In addition, the CNCNF ecosystem has grown to hosting 17 open source projects, including Prometheus, Envoy, gRPC, and many others. Finally, the CNCNF also nurtures several early stage projects and has eight projects accepted into its Sandbox program for emerging technologies.

3 Vaughan-Nicholls, Steven J. (2015-07-21). "Cloud Native Computing Foundation seeks to forge cloud and container unity", ZDNet.

4 Check out the "Cloud Native Computing Foundation ("CNCNF") Charter" on the Cloud Native Computing Foundation website.

5 See the [list of members](#) on the Cloud Native Computing Foundation website.

With the weight of the vendor-neutral CNCF foundation behind it, Kubernetes has grown to have **more than 2,300 contributors** from a wide range of industries. In addition to hosting several cloud-native projects, the CNCF provides training, a Technical Oversight Board, a Governing Board, a community infrastructure lab, and several certification programs. In the next section, we describe CNCF's highly successful Kubernetes Conformance Certification, which is focused on improving Kubernetes interoperability and workload portability.

CNCF Kubernetes Conformance Certification Keeps the Focus on User Needs

A key selling point for any open source project is that different vendor distributions of the open source project are interoperable. Customers are very concerned about vendor lock-in: being able to easily change the vendor that provides a customer their open source infrastructure is crucial. In the context of Kubernetes, it needs to be easy for the customer to move its Kubernetes workloads from one vendor's Kubernetes platform to a different vendor's Kubernetes platform. In a similar fashion, a customer might have a workload that normally runs on an on-premises Kubernetes private cloud, but during holiday seasons, the workload might merit obtaining additional resources on a public Kubernetes cloud as well. For all these reasons, it is absolutely critical that Kubernetes platforms from different vendors be interoperable and that workloads are easily portable to different Kubernetes environments.

Fortunately, the CNCF identified this critical requirement early on in the Kubernetes life cycle before any serious forks in the Kubernetes distributions had occurred. The CNCF formed the **Kubernetes Conformance Certification Workgroup**. The mission of the Conformance Certification Workgroup is to provide a software conformance program and test suite that any Kubernetes implementation can use to demonstrate that it is conformant and interoperable.

As of this writing, 60 vendor distributions had successfully passed the Kubernetes Conformance Certification Tests. The Kubernetes Conformance Workgroup **continues to make outstanding progress**, focusing on topics such as increased conformance test coverage, automated conformance reference test documentation generation,

and was even a major highlight of the [KubeCon Austin 2017 Key-note presentation](#).

Summary

This chapter discussed a variety of factors that have contributed to Kubernetes becoming the de facto standard for the orchestration and management of cloud-native computing applications. Its declarative model, built-in support for autoscaling, improved networking model, health-check support, and the backing of the CNCF have resulted in a vibrant and growing ecosystem for Kubernetes with adoption across cloud applications and high-performance computing domains. In [Chapter 2](#), we begin our deeper exploration into the architecture and capabilities of Kubernetes.

Fundamental Kubernetes Topics

In this chapter, we provide an introduction to the basic foundations of Kubernetes. We begin with an overview of the Kubernetes architecture and its deployment models. Next, we describe a few options for running Kubernetes and describe a variety of deployment environments. We then describe and provide examples of several fundamental Kubernetes concepts including Pods, labels, annotations, ReplicaSets, and Deployments.

Kubernetes Architecture

Kubernetes architecture at a high level is relatively straightforward. It is composed of a *master node and a set of worker nodes*. The nodes can be either physical servers or virtual machines (VMs). Users of the Kubernetes environment interact with the master node using either a command-line interface (kubectl), an application programming interface (API), or a graphical user interface (GUI). The master node is responsible for scheduling work across the worker nodes. In Kubernetes, the unit of work that is scheduled is called a *Pod*, and a Pod can hold one or more container. The primary components that exist on the master node are the *kube-apiserver*, *kube-scheduler*, *etcd*, and the *kube-controller-manager*:

kube-apiserver

The kube-apiserver makes available the Kubernetes API that is used to operate the Kubernetes environment.

kube-scheduler

The kube-scheduler component is responsible for selecting the nodes on which Pods should be created.

kube-controller-manager

Kubernetes provides several high-level abstractions for supporting replicas of Pods, managing nodes, and so on. Each of these is implemented with a controller component, which we describe later in this chapter. The kube-controller-manager is responsible for managing and running controller components.

etcd

The etcd component is a distributed key-value store and is the primary communication substrate used by master and worker nodes. This component stores and replicates the critical information state of your Kubernetes environment. Kubernetes outstanding performance and scalability characteristics are dependent on etcd being a highly efficient communication mechanism.

The worker nodes are responsible for running the Pods that are scheduled on them. The primary Kubernetes components that exist on worker nodes are the *kubelet*, *kube-proxy*, and the *container runtime*:

kubelet

The kubelet is responsible for making sure that the containers in each Pod are created and stay up and running. The kubelet will restart containers upon recognizing that they have terminated unexpectedly.

kube-proxy

One of Kubernetes key strengths is the networking support it provides for containers. The kube-proxy component provides networking support in the form of connection forwarding, load balancing, and the mapping of a single IP address to a Pod.

Container runtime

The container runtime component is responsible for actually running the containers that exist in each Pod. Kubernetes **supports several container runtime environment options** including Docker, rkt, and containerd.

Figure 2-1 shows a graphical representation of the Kubernetes architecture encompassing a master node and two worker nodes.

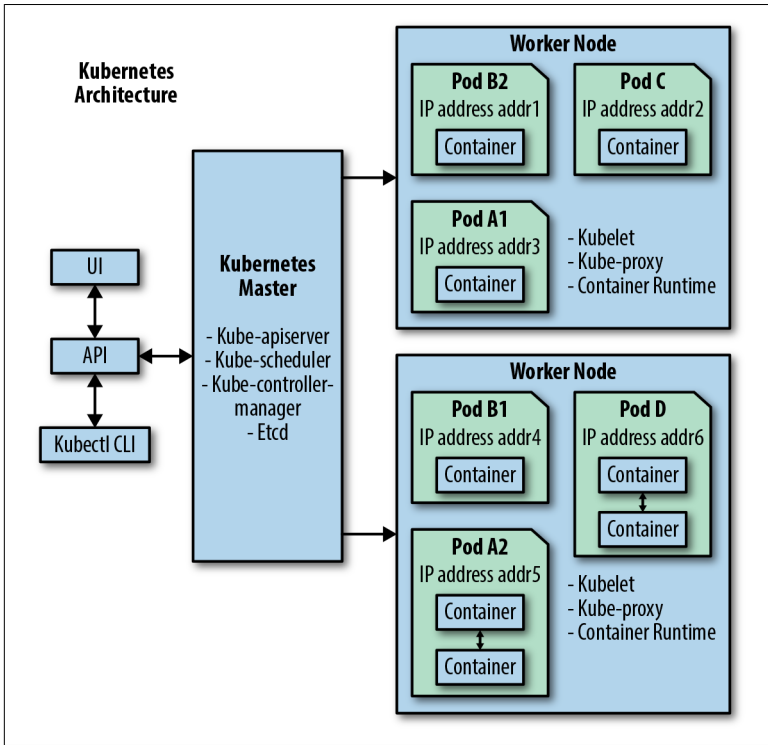


Figure 2-1. Graphical representation of the Kubernetes architecture

As shown in Figure 2-1, users interact with the Kubernetes master node using either a GUI or by command-line interface (kubectl CLI). Both of these use the Kubernetes exposed API to interact with the Kubernetes master node. The Kubernetes master node schedules Pods to run on different worker nodes. Each Pod contains one or more containers, and each Pod is assigned its own IP address. In many real-world applications, Kubernetes deploys multiple replica copies of the same Pod to improve scalability and ensure high availability. Pods A1 and A2 are Pod replicas that differ only in the IP address they are allocated. In a similar fashion Pods B1 and B2 are also replica copies of the same Pod. The containers located in the same Pod are permitted to communicate with one another using standard interprocess communication (IPC) mechanisms.

In the next section, we expand our understanding of the Kubernetes architecture by learning about several ways to run Kubernetes.

Let's Run Kubernetes: Deployment Options

Kubernetes has reached such an incredible level of popularity that there are now numerous public cloud and on-premises cloud Kubernetes deployments available. The **list of deployment options** is too large to include here. In the following subsection, we summarize a few Kubernetes options that are representative of the types of deployments currently available. We will discuss the Katacoda Kubernetes Playground, Minikube, IBM Cloud Private, and the IBM Cloud Kubernetes Service.

Katacoda Kubernetes Playground

The **Katacoda Kubernetes Playground** provides online access to a two-node Kubernetes environment. The environment provides two terminal windows that allow you to interact with this small Kubernetes cluster. The cluster is available for only 10 minutes—then you need to refresh the web page, and the entire environment disappears. The 10-minute playground session is long enough to try all of the Kubernetes examples that are presented in the next section of this chapter. Just remember that the environment lasts only 10 minutes, so avoid taking a long coffee break when using it.

Minikube

Minikube is a tool that enables you to run a single-node Kubernetes cluster within a VM locally on your laptop. Minikube is well suited for trying many of the basic Kubernetes examples that are presented in the next section of this chapter, and you can also use it as a development environment. In addition, Minikube supports a variety of VMs and container runtimes.

IBM Cloud Private

IBM Cloud Private is a Kubernetes-based private cloud platform for running cloud-native or existing applications. IBM Cloud Private provides an integrated environment that enables you to design, develop, deploy, and manage on-premises containerized cloud applications on your own infrastructure, either in a datacenter or on

public cloud infrastructure that you source from a cloud vendor. IBM Cloud Private is a software form factor of Kubernetes that focuses on keeping a pure open source distribution complemented with the capabilities you would typically have to build around it, including the operational logging, health metrics, audit practices, identity and access management, management console, and ongoing updates for each component. IBM Cloud Private also provides a rich catalog of IBM and open source middleware to enable you to quickly deploy complete stacks for data, caching, messaging, and microservices development. The **Community Edition** is available at no charge and quickly enables you to stand up an enterprise-ready Kubernetes platform.

See **Appendix A** for instructions on configuring an IBM Cloud Private cluster.

IBM Cloud Kubernetes Service

The **IBM Cloud Kubernetes Service** is a managed Kubernetes offering that delivers powerful tools, an intuitive user experience, and built-in security for rapid delivery of container applications that you can bind to cloud services related to IBM Watson, Internet of Things (IoT), DevOps, and data analytics. The IBM Cloud Kubernetes Service provides intelligent scheduling, self-healing, horizontal scaling, service discovery and load balancing, automated rollouts and rollbacks, and secret and configuration management. The Kubernetes service also has advanced capabilities around simplified cluster management, container security and isolation policies, the ability to design your own cluster, and integrated operational tools for consistency in deployment.

See **Appendix A** for instructions on configuring an IBM Cloud Kubernetes Service cluster.

Running the Samples Using kubectl

After covering some core concepts in Kubernetes, the next sections provide several examples in the form of YAML files. For all of the aforementioned environments, you can run the samples provided by using the standard Kubernetes command-line tool known as *kubectl*. They also describe how you can install kubectl. After you have your Kubernetes environment up and running and kubectl installed, you can run all of the following YAML file samples in the next sections

by first saving the YAML to a file (e.g., *kubesample1.yaml*) and then by running the following `kubectl` command:

```
$ kubectl apply -f kubesample1.yaml
```

The `kubectl` command provides a large number of options beyond just creating an environment based on a YAML file.

Kubernetes Core Concepts

Kubernetes has several concepts that are specific to its model for the orchestration and management of containers. These include *Pods*, *labels*, *annotations*, *ReplicaSets*, and *Deployments*.

What's a Pod?

Because Kubernetes provides support for the management and orchestration of containers, you would assume that the smallest deployable unit supported by Kubernetes would be a container. However, the designers of Kubernetes learned from experience¹ that it was more optimal to have the smallest deployable unit be something that could hold multiple containers. In Kubernetes, this smallest deployable unit is called a Pod. A Pod can hold one or more application containers. The application containers that are in the same Pod have the following benefits:

- They share an IP address and port space
- They share the same hostname
- They can communicate with each other using native interprocess communication (IPC)

In contrast, application containers that run in separate Pods are guaranteed to have different IP addresses and have different hostnames. Essentially, containers in different Pods should be viewed as running on different servers even if they ended up on the same node.

Kubernetes provides a robust list of features that make Pods easy to use:

¹ Brendan Burns et al. (2016). "Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade". *ACM Queue* 14: 70–93.

Easy-to-use Pod management API

Kubernetes provides the `kubectl` command-line interface, which supports a variety of operations on Pods. The list of operations includes the creating, viewing, deleting, updating, interacting, and scaling of Pods.

File copy support

Kubernetes makes it very easy to copy files back and forth between your local host machine and your Pods running in the cluster.

Connectivity from your local machine to your Pod

In many cases, you will want to have network connectivity from your local host machine to your Pods running in the cluster. Kubernetes provides port forwarding whereby a network port on your local host machine is connected via a secure tunnel to a port of your Pod that is running in the cluster.

Volume storage support

Kubernetes Pods support the attachment of remote network storage volumes to enable the containers in Pods to access persistent storage that remains long after the lifetime of the Pods and the containers that initially utilized it.

Probe-based health-check support

Kubernetes provides health checks in the form of probes to ensure the main processes of your containers are still running. In addition, Kubernetes also provides liveness checks that ensure the containers are actually functioning and capable of doing real work. With this health check support, Kubernetes can recognize when your containers have crashed or become non-functional and restart them on your behalf.

How Do I Describe What's in My Pod?

Pods and all other resources managed by Kubernetes are described by using a YAML file. The following is a simple YAML file that describes a rudimentary Pod resource:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
```

```
- name: nginx
  image: nginx:1.7.9
  ports:
  - containerPort: 80
```

This YAML file contains the following fields and sections:

apiVersion

This field is used to declare which version of the Kubernetes API schema is being used. Kubernetes continues to experience a rapid growth in features and functionality. It manages the complexity that results from its growth in capabilities by supporting multiple versions of its API. By setting the `apiVersion` field, you can control the API version that your resource uses.

kind

You use the `kind` field to identify the type of resource the YAML file is describing. In the preceding example, the YAML file declares that it is describing a Pod object.

metadata

The `metadata` section contains information about the resource that the YAML is defining. In the preceding example, the metadata contains a `name` field that declares the name of this Pod. The `metadata` section can contain other types of identifying information such as labels and annotations. We describe these in the next section.

spec

The `spec` section provides a specification for what is the desired state for this resource. As shown in the example, the desired state for this Pod is to have a container with a name of `nginx` that is built from the Docker image that is identified as `nginx:1.7.9`. The container shares the IP address of the Pod it is contained in and the `containerPort` field is used to allocate this container a network port (in this case, 80) that it can use to send and receive network traffic.

To run the previous example, save the file as *pod.yaml*. You can now run it by doing the following:

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

After running this command, you should see the following output:

```
pod "nginx" created
```


To confirm that your Pod is actually running, use the `kubectl get pods` command to verify:

```
$ kubectl get pods
```

After running this command, you should see output similar to the following:

```
NAME    READY   STATUS    RESTARTS   AGE
nginx   1/1     Running   0           21s
```

If you need to debug your running container, you can create an interactive shell that runs within the container by using the following command:

```
$ kubectl exec -it nginx -- bash
```

This command instructs Kubernetes to run an interactive shell for the container that runs in the Pod named `nginx`. Because this Pod has only one container, Kubernetes knows which container you want to connect to without you specifying the container name as well. Typically, accessing the container interactively to modify it at runtime is considered a bad practice. However, interactive shells can be useful as you are learning or debugging applications before deploying to production. After you run the preceding command, you can interact with the container's runtime environment, as shown here:

```
root@nginx:/# ls
bin boot dev etc home lib lib64 media mnt opt proc
root run sbin
selinux srv sys tmp usr var
root@nginx:/# exit
```

If your Pod had multiple containers within it, you would need to include the container name as well in your `kubectl exec` command. To do this, you would use the `-c` option and include the container name in addition to the Pod name. Here is an example:

```
$ kubectl exec -it nginx -c nginx -- bash
root@nginx:/# exit
exit
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod nginx
```

You should see the following confirmation that the Pod has been deleted:

```
pod "nginx" deleted
```

When using Kubernetes you can expect to have large numbers of Pods running in a cluster. In the next section, we describe how labels and annotations are used to help you keep track of and identify your Pods.

Labels and Annotations

Kubernetes supports the ability to add key–value data pairs to its Pods and also to the other Kubernetes resources such as ReplicaSets and Deployments, which we describe later in this chapter. There are two forms of these key–value pairs, *labels* and *annotations*. Labels are added to Pods to give extra attribute fields that other resources can then use to identify and select the desired Pods in which they are interested. Annotations are used to add extra attribute information to Pods as well. However, unlike labels, annotations are not used in query operations to identify Pods. Instead, annotations provide extra information that can be helpful to users of the Pods or automation tools. The following example takes the previous YAML file describing your Pod and adds labels and annotations:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    kubernetes.io/change-cause: "Update nginx to 1.7.9"
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

In this example, we have added a label with `app` as the key and `webserver` as the value. Other Kubernetes constructs can then do searches that match on this value to find this Pod. If there is a group of Pods with this label, they can all be found. This simple and elegant approach of identifying Pods is used heavily by several higher-level Kubernetes abstractions that are described later in this chapter.

Similarly, the previous example also demonstrates that we have added an annotation. In this case, the annotation has `kuber`

netes.io/change-cause as the key and Update nginx to 1.7.9 as its value. The purpose of this annotation is to provide information to users or tools; it is not meant to be used as a way to query and identify desired Kubernetes resources.

In the next section, we introduce ReplicaSets, which is one of Kubernetes higher-level abstractions that uses labels to identify a group of Pods to manage.

ReplicaSets

Kubernetes provides a high-level abstraction called a *ReplicaSet* that is used to manage a group of Pod replicas across a cluster. The key advantage of a ReplicaSet is that you get to declare the number of Pod replicas that you desire to run concurrently. Kubernetes will monitor your Pods and will always strive to ensure that the number of copies running is the number you selected. If some of your Pods terminate unexpectedly, Kubernetes will instantiate new versions of them to take their place. For cloud application operators accustomed to being contacted in the middle of the night to restart a crashed application, having Kubernetes instead automatically handle this situation on its own is a much better alternative.

To create a ReplicaSet, you provide a specification that is similar to the Pod specification shown in [“How Do I Describe What’s in My Pod?” on page 15](#). The ReplicaSet adds new information to the specification to declare the number of Pod replicas that should be running and also provide matching information that identifies which Pods the ReplicaSet is managing. Here is an example YAML specification for a ReplicaSet:

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    kubernetes.io/change-cause: "Update nginx to 1.7.9"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
```

```
  labels:
    app: webserver
spec:
  containers:
  - name: nginx
    image: nginx:1.7.9
    ports:
    - containerPort: 80
```

In this specification, the `kind` field is set to `ReplicaSet`, denoting this specification is for a `ReplicaSet` resource. Similar to the previous `Pod` example, the `ReplicaSet` YAML has a `metadata` section for `name`, `labels`, and `annotation` fields. The real differences between `ReplicaSets` and `Pods` occurs in the `spec` section, which has a `replicas` field that is used to denote the number of `Pod` replicas that should run concurrently. In this example, we are declaring that Kubernetes should strive to always have three copies of the replicated `Pods` running. It is worth noting that if you change this value and use the `kubectl apply` command to update the `ReplicaSet` specification, Kubernetes will then either increase or decrease the number of `Pods` to satisfy the new `replicas` value you requested.

The `spec` section has a `selector` field that is used to provide the labels that this `ReplicaSet` will use to identify its `Pod` replicas. As shown in this example, the selector for this `ReplicaSet` states this `ReplicaSet` is managing `Pod` replicas that have a label with `app` as the key and `webserver` as its associated value.

The `template` section is the next section of this specification. It provides a template that describes what the `Pod` replicas that are managed by the `ReplicaSet` will look like. Note that the `template` section must be able to describe everything that a standalone `Pod` YAML could describe. Because of this, the `template` section itself contains a `metadata` section and a `spec` section.

The `metadata` section, similar to previous examples, contains labels. In the preceding example, the `metadata` section declares a label with `app` as the key and `webserver` as its associated value. Not surprisingly, this is the exact label that the `ReplicaSet` selector field is using to identify the `Pod` replicas it manages.

Additionally, the `template` section contains its own `spec` section. This `spec` section describes the containers that comprise the `Pod` replicas the `ReplicaSet` will manage, and in the example, you can see that fields such as `name`, `images`, and `ports` that are found in `Pod`

YAMLs are also repeated here. As result of this structure, ReplicaSet can thus have multiple `spec` sections, and these sections are nested inside one another which can look complex and intimidating. However, after you understand that a ReplicaSet needs to specify not only itself but also the Pod replicas it manages, the nested `spec` structure is less bewildering.

To run the previous example, save the example as the file `replicaset.yaml`. You can now run the example by doing the following:

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

After running this command, you should see the following output:

```
replicaset.apps "nginx" created
```

To confirm that your Pod replicas are actually running, use the `kubectl get pods` command to verify:

```
$ kubectl get pods
```

After running this command, you should see output similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
nginx-fvtzq	1/1	Running	0	23s
nginx-jfxdn	1/1	Running	0	23s
nginx-v7kqq	1/1	Running	0	23s

To demonstrate the capabilities of the ReplicaSet, let's purposely delete one of its pods:

```
$ kubectl delete pod nginx-v7kqq
pod "nginx-v7kqq" deleted
```

If we run `kubectl get pods` quickly enough, we see that the pod we deleted is being terminated. The ReplicaSet realizes that it lost one of its Pods. Because its YAML specification declares that its desired state is three Pod replicas, the ReplicaSet starts a new instance of the `nginx` container. Here's the output of this command:

```
$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-fvtzq	1/1	Running	0	1m
nginx-jfxdn	1/1	Running	0	1m
nginx-kfgxk	1/1	Running	0	5s
nginx-v7kqq	0/1	Terminating	0	1m

After a short amount of time, if you run `kubectl get pods` again, you'll notice just the two original Pod replicas and the newly created substitute Pod replica are present:

```
$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
nginx-fvtzq   1/1     Running   0           1m
nginx-jfxdn   1/1     Running   0           1m
nginx-kfgxk   1/1     Running   0           23s
```

To delete the ReplicaSet that you just created, run the following command:

```
$ kubectl delete replicaset nginx
```

You should see the following confirmation that the ReplicaSet has been deleted:

```
replicaset.extensions "nginx" deleted
```

Although ReplicaSets provide very powerful Pod replica capabilities, they provide no support to help you manage the release of new versions of your Pod ReplicaSets. ReplicaSets would be more powerful if they supported the ability to roll out new versions of the Pod replicas and provide flexible control on how quickly the Pod replicas were replaced with new versions. Fortunately, Kubernetes provides another high-level abstraction, called Deployments, that provides this type of functionality. The next section describes the capabilities provided by Deployments.

Deployments

Deployments are a high-level Kubernetes abstraction that not only allow you to control the number of Pod replicas that are instantiated, but also provide support for rolling out new versions of the Pods. Deployments rely upon the previously described ReplicaSet resource to manage Pod replicas and then add Pod version management support on top of this capability. Deployments also enable newly rolled out versions of Pods to be rolled back to previous versions if there is something wrong with the new version of the Pods. Furthermore, Deployments support two options for upgrading Pods, Recreate and RollingUpdate:

Recreate

The Recreate Pod upgrade option is very straightforward. In this approach the Deployment resource modifies its associated ReplicaSet to point to the new version of the Pod. It then proceeds to terminate all of the Pods. The ReplicaSet then notices that all of the Pods have been terminated and thus spawns new Pods to ensure that the number of desired replicas are up and

running. The Recreate approach will typically result in your Pod application not being accessible for a period of time and thus it is not recommended for applications that need to always be available.

RollingUpdate

The Kubernetes Deployment resource also provides a RollingUpdate option. With this option, your Pods are replaced with the newer version incrementally over time. This approach results in there being a mixture of both the old version of the Pod and the new version of the Pod running simultaneously and thus avoids having your Pod application unavailable during this maintenance period.

The following is an example YAML specification for a Deployment that uses the RollingUpdate option:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    deployment.kubernetes.io/revision: "1"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
      type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

This Deployment example encompasses many of the characteristics that we have seen in ReplicaSets and Pods. In its metadata are labels

and annotations. For the Deployment, an annotation with `deployment.kubernetes.io/revision` as the key and `1` as its value provides information that this is the first revision of the contents in this Deployment. Similar to ReplicaSets, the Deployment declares the number of replicas it provides and uses a `matchLabels` field to declare what labels it uses to identify the Pods it manages. Also similar to ReplicaSets, the Deployment has both a `spec` section for the Deployment and a nested `spec` section within a `template` that is used to describe the containers that comprise the Pod replicas managed by this Deployment.

The fields that are new and specific to a Deployment resource are the `strategy` field and its subfields of `type` and `rollingUpdate`. The `type` field is used to declare the Deployment strategy being utilized; currently, you can set this to `Recreate` or `RollingUpdate`.

If you choose the `RollingUpdate` option, you need to set the subfields of `maxSurge` and `maxUnavailable` as well. You use the options as follows:

`maxSurge`

The `maxSurge` `RollingUpdate` option enables extra resources to be allocated during a rollout. You can set the value of this option to a number or a percentage. As a simple example, assume a Deployment is supporting three replicas and `maxSurge` is set to `2`. In this scenario, there will be a total of five replicas available during the `RollingUpdate`.

At the peak of the deployment, there will be three replicas with the old version of the Pods running and two with the new version of the Pods running. At this point, one of the old version Pod replicas will need to be terminated and another replica of the new Pod version can then be created. At this point, there would be a total of five replicas, three of which have the new revision, and two have the old version of the Pods. Finally, having reached a point of having the correct number of Pod replicas available with the new version, the two Pods with the old version can now be terminated.

`maxUnavailable`

You use this `RollingUpdate` option to declare the number of the Deployment replica Pods that can be unavailable during the update. You can set this to either a number or a percentage.

The following YAML example shows a Deployment that has been updated to initiate a rollout. Note that a new annotation label with a key of `kubernetes.op/change-cause` has been added with a value that denotes an update to the version of `nginx` running in the container has occurred. Also note that the name of the image used by the container in the `spec` section has changed to `nginx:1.13.10`. This declaration is what actually drives the Pod replicas managed by the Deployment to now have a new version of the container images when the upgrade occurs. Let's take a look at the code:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: nginx
  labels:
    app: webserver
  annotations:
    kubernetes.io/change-cause: "Update nginx to 1.13.10"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: webserver
  strategy:
    rollingUpdate:
      maxSurge: 1
      maxUnavailable: 1
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
      - name: nginx
        image: nginx:1.13.10
        ports:
        - containerPort: 80
```

To demonstrate the capabilities of Deployments, let's run the two previous examples. Save the first Deployment example as `deploymentset.yaml` and the second example as `deploymentset2.yaml`. You can now run the first deployment example by doing the following:

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

After running this command, you should see the following output:

```
deployment.extensions "nginx" created
```

To confirm that your Pod replicas managed by the Deployment are actually running, use the `kubectl get pods` command to verify, as shown here:

```
$ kubectl get pods
```

After running this command, you should see output similar to the following:

NAME	READY	STATUS	RESTARTS	AGE
nginx-7bbd56b666-5x7fl	0/1	ContainerCreating	0	10s
nginx-7bbd56b666-cm7fn	0/1	ContainerCreating	0	10s
nginx-7bbd56b666-ddtt7	0/1	ContainerCreating	0	10s

With Deployments, there is a new command called `kubectl get deployments` that provides the status on the Deployments as they update their images. You run this command as follows:

```
$ kubectl get deployments
```

After running this command, you should see output similar to the following:

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx	3	3	3	3	2m

Now to make things interesting, let's update the image in the Deployment by applying the second Deployment example that you saved in `deploymentset2.yaml`. Note that you could have just updated your original YAML that you saved in `deploymentset.yaml` instead of using two separate files. Begin the update by doing the following:

```
$ kubectl apply -f deploymentset2.yaml
```

After running this command, you should see the following output:

```
deployment.extensions "nginx" configured
```

Now, when you rerun the `kubectl get deployments` command, you see a much more interesting result:

```
$ kubectl get deployments
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
nginx	3	4	2	2	4m

As shown in this output, the Deployment currently has four Pod replicas running. Two of the Pod replicas are up to date, which means they are now running the updated nginx image, two of the Pod replicas are available, and currently there are four Pod replicas in total. After some amount of time, when the rolling image update is complete, we reach the desired state of having three updated Pod

replicas available. You can confirm this by rerunning the `kubectl get deployments` command and viewing that the output now matches your desired state:

```
$ kubectl get deployments
NAME          DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
nginx         3         3         3             3           4m
```

To delete the Deployment that you just created, run the following command:

```
$ kubectl delete deployment nginx
```

You should see the following confirmation that the Deployment has been deleted:

```
deployment.extensions "nginx" deleted
```

Deployments also provide commands for pausing rollouts, resuming rollouts, and for rolling back the update of an image. The commands are quite helpful if you have some concerns about the new image being rolled out that merits investigation, or if you determine that the updated image being rolled out is problematic and needs to be rolled back to a previous version.

NOTE

For more information on how to use these deployment capabilities, go to [Kubernetes docs](#).

Although Deployments provide support for managing Pod replicas and their versioning life cycle, they do not provide a load balancer for distributing requests across the Pod replicas. In [Chapter 3](#), we introduce the Kubernetes Service Object, which provides this capability and also discuss several other advanced Kubernetes topics.

Advanced Kubernetes Topics

In this chapter, we provide an overview of some of Kubernetes more advanced topics. We begin with a description of Kubernetes' Service Object, which is its built-in facility for load balancing across Pod replicas. Next, we describe a few specialized options for managing Pod groups, including DaemonSets and StatefulSets. We then describe and provide examples of several advanced Kubernetes concepts, including Volumes, PersistentVolumes, ConfigMaps, Secrets, and image registry support. We conclude this chapter with a section on Helm, Kubernetes' package manager, which enables you to create a package containing the multiple templates that make up your application.

Kubernetes Service Object: Load Balancer Extraordinaire

Kubernetes provides a Service Object as its mechanism for providing load balancing across Pod replicas. Upon first inspection, having this feature built in to Kubernetes might appear to be overkill because there are lots of open source load balancers already available. However, the high-level Pod replica management features Kubernetes provides necessitates it providing a custom load balancer for Pod replicas. As discussed in previous chapters, Kubernetes provides several high-level abstractions that are capable of starting new Pod replicas when needed. When this occurs, these Pod replicas might end up moving to different servers. Most load balancers that are available were not built to handle this high level of dyna-

mism. Kubernetes thus provides a load balancer in the form of a Service Object. The Service Object has the following critical features that are tailored to supporting Pod replicas:¹

Virtual IP allocation and load balancing support

When a Service Object is created for a group of Pod replicas, a virtual IP address is created that is used to load balance across all the Pod replicas. The virtual IP address, also referred to as a cluster IP address, is a stable value and suitable for use by Domain Name System (DNS) services.

Port mapping support

Service Objects support the ability to map from a port on the cluster IP address to a port being used by Pod replicas. For example, the Service Object might want to expose the Pod's application as running on port 80 even though the Pod replicas are listening on a more generic port.

Built-in readiness-check support

Kubernetes Service Objects provide built-in readiness-check support. With this capability, the load-balancing capabilities provided by the Service Object are smart enough to avoid routing requests to Pod replicas that are not ready to receive requests.

Creating a Service Object for a Deployment of Pod replicas is very straightforward. You can accomplish this by using the `kubectl expose` command. Assuming that you have provisioned the `nginx` Deployment example from [Chapter 2](#), you can expose it as a service by doing the following:

```
$ kubectl expose deployment nginx --port=80 --target-port=8000
```

In this example, the `nginx` Deployment has been exposed as a service. The service is running on port 80 and will forward to the Pod replicas, which are listening on port 8000. To identify what the cluster IP address is for this newly exposed service, use the `kubectl get services` command:

```
$ kubectl get services
```

¹ *Kubernetes: Up and Running* by Kelsey Hightower, Brendan Burns, and Joe Beda (O'Reilly). Copyright 2017 Kelsey Hightower, Brendan Burns, and Joe Beda, 928-1-491-93567-5.

In the next section, we describe another high-level construct provided by Kubernetes, the `DaemonSet`, which handles the specialized use case of having to run a single Pod replica on each Node in the cluster.

DaemonSets

One common specialized use case when running Kubernetes applications is the need to have a single Pod replica running on each Node. The most common scenario for this is when a single monitoring or logging container application should be run on each Node. Having two of these containers running on the same Node would be redundant, and all Nodes need to have the monitoring/logging container running on it. For this use case, Kubernetes provides a *DaemonSet* resource that ensures a single copy of the Pod replica will run on each Node. The following example illustrates how to run a single Pod replica on each Node using a `DaemonSet`:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx
  labels:
    app: webserver
spec:
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
      containers:
        - name: nginx
          image: nginx:1.7.9
          ports:
            - containerPort: 80
```

As shown in the example, a `DaemonSet` YAML looks very similar to a `ReplicaSet`. It uses a `selector` to provide the labels this `DaemonSet` will use to identify its Pod replicas. In the example, the `selector` for this `DaemonSet` states that it is managing Pod replicas that have a label with `app` as the key and `webserver` as its associated value. The `DaemonSet` also has a `template` section and a nested `spec` section that serve the same purpose as they do in `ReplicaSets`.

The notable differences of a DaemonSet from a ReplicaSet are also illustrated in the previous example. The `kind` field is set to `DaemonSet`, and there is no need to declare the number of replicas desired. The number of replicas will always be set to be a value that matches the placing of one Pod replica on each Node.

Customizing DaemonSets

In some situations, a DaemonSet is desired, but with the complicating factor that we don't want a Pod replica running on all the Nodes. For this scenario, you can customize a DaemonSet to identify the Nodes deserving of running the Pod replica by using a `nodeSelector` construct. In this approach, a `nodeSelector` is used to define a label that the DaemonSet will look for on all the Nodes in the cluster. The Nodes that have this label will be the ones that will have a DaemonSet-managed Pod replica instantiated on them. As an illustration of this capability, we first label a Node (*node1* in this example) on which we want to run the Pod replica by using the `kubectl label nodes` command:

```
$ kubectl label nodes node1 needsdaemon=true
```

With the `needsdaemon` label added to the Node, you can modify and customize this DaemonSet to include the appropriate label and node Selector:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: nginx
  labels:
    app: webserver
spec:
  selector:
    matchLabels:
      app: webserver
  template:
    metadata:
      labels:
        app: webserver
    spec:
      nodeSelector:
        needsdaemon: "true"
      containers:
        - name: nginx
          image: nginx:1.7.9
```



```
ports:
  - containerPort: 80
```

As shown in this example, the `nodeSelector` contains the label `needsdaemon` with a value of `"true"`. The quotes added around the `true` value are required to ensure that this value is interpreted as a string and not a Boolean. With this value added to the `DaemonSet` YAML, you have now customized it such that the container replicas will run on only the properly labeled Nodes.

In the next section, we switch gears a little and focus on `StatefulSets`, which is how Kubernetes provides support for the integration of stateful services.

StatefulSets

Many of the high-level constructs Kubernetes provides, such as `ReplicaSets`, provide support for managing a set of Pod replicas that are identical and hence interchangeable. When integrating stateful, replicated services into Kubernetes, such as persistent databases, standard identical Pod replicas are not sufficient to meet the requirements of this class of applications. Instead, these applications need Pods that have a unique ID so that the correct storage volume can always be reattached to the correct Pod. To support this class of applications, Kubernetes provides the **StatefulSet resource**. `StatefulSets` have the following characteristics:²

Stable hostname with unique index

Each Pod associated with the `StatefulSet` is allocated a persistent hostname with a unique, monotonically increasing index appended to the hostname.

Orderly deployment of Pod replicas

Each Pod associated with the `StatefulSet` is created sequentially in order from lowest index to highest index.

Orderly deletion of Pod replicas

Each Pod associated with the `StatefulSet` is deleted sequentially in order from highest index to lowest index.

² Hightower, Kelsey, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running*. Sebastopol: O'Reilly Media, 2017.

Orderly deployment and scaling of Pod replicas

Each Pod associated with the StatefulSet is scaled sequentially in order from lowest index to highest index, and before a scaling operation can be applied to a Pod all its predecessors must be running.

Orderly automated rolling upgrades of Pod replicas

Each Pod associated with the StatefulSet is updated by deleting and recreating each Pod sequentially in order from highest index to lowest index.

Headless service associated with the StatefulSet

A Service Object that has no cluster virtual IP address assigned to it is associated with the StatefulSet to manage the DNS entries for each Pod. This Service does not load balance across the Pods in the StatefulSet, because each Pod is unique and client requests need to be always directed to the same Pod.

The YAML descriptions for StatefulSets look very similar to those used for ReplicaSets. The following is an example StatefulSet that also includes a YAML description for its headless service:

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
    app: nginx
spec:
  ports:
    - port: 80
      name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: nginx
  labels:
    app: webserver
spec:
  serviceName: "nginx"
  replicas: 3
  selector:
    matchLabels:
      app: webserver
```

```

template:
  metadata:
    labels:
      app: webserver
  spec:
    containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
          - containerPort: 80

```

At the top of this example is the YAML specification for the headless service:

- The `kind` field is set to `Service`, denoting that the top portion of the specification is for a `Service` resource.
- Note that this service is declared to be headless by setting the `clusterIP` field to `None`.
- In the next section of the specification, the `StatefulSet` is defined. In this portion, the `kind` field is set to `StatefulSet`. It is also worth noting is that `StatefulSet` must identify the `Service` that manages it.
- You accomplish this by setting the `serviceName` field to the name declared in the `Service`.
- In this example, the name of the `Service` is `nginx`, and that is the value placed in the `serviceName` field.

The remaining portions of the `StatefulSet` are the same types of values that are set in `ReplicaSets`. A key portion of the `StatefulSet` that is not shown in the preceding example is the creation of persistent volume storage for each `Pod` in the `StatefulSet`. Kubernetes provides a `volumeClaimTemplate` to manage the mapping of a volume with each `Pod`, and the volume is automatically mounted when the `Pod` is rescheduled. More detail on Kubernetes volume support is provided in [“Volumes and Persistent Volumes” on page 36](#).

NOTE

The preceding `StatefulSet` example uses the `nginx` web server in an effort to keep the example simple to understand. Typically, you would not use a `StatefulSet` for a stateless application like `nginx`. Instead, you would more commonly use `StatefulSets` for stateful applications such as `redis`, `etcd`, and `sql`.

To run the previous example, save it in a file named *statefulset.yaml*. You can now run the example by doing the following:

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

To see the Pods that are created with the index added to the names, run the following command:

```
$ kubectl get pods
```

You will then see output that looks like the following:

NAME	READY	STATUS	RESTARTS	AGE
nginx-0	1/1	Running	0	16s
nginx-1	1/1	Running	0	3s
nginx-2	1/1	Running	0	2s

In the next section, we switch gears a little and focus on how Kubernetes enables containers within a Pod to share filesystem directories using the concept of Volumes.

Volumes and Persistent Volumes

Pods can have multiple containers that need to share filesystem resources. Kubernetes supports the notion of *Volumes* as its mechanism for enabling containers in a Pod to support the sharing of a directory in the filesystem. In general, there are two types of Volumes, *basic* and *persistent*:

Basic Volumes

Basic Volumes, which are typically referred to as just Volumes, are pretty straightforward. They enable containers in the same Pod to share a directory in the filesystem. The content that they share is often stored on the filesystem of the Node that is hosting the Pod. The file content stored in the Volume will survive a container being destroyed and re-created, but as soon as the Pod itself is destroyed the file content in the Volume is deleted permanently.

Persistent Volumes

Kubernetes supports several types of PersistentVolumes. PersistentVolumes are able to save the shared file content such that it can survive Pod restarts. A variety of PersistentVolume types exist and they are usually implemented by using some underlying network-based storage mechanism such as NFS, FlexVo-

lume, iSCSI, VsphereVolume, AzureFile, GCEPersistentDisk, AWSElasticBlockStore, and several others.

The following YAML specification illustrates a simple Pod example that mounts a basic volume for the containers in the Pod to share as a common filesystem directory:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  volumes:
    - name: sharedspace
      hostPath:
        path: "/var/sharedPodDir"
  containers:
    - name: nginx
      image: nginx:1.7.9
      volumeMounts:
        - mountPath: "/containerMount"
          name: "sharedspace"
  ports:
    - containerPort: 80
```

As shown in this example, the Pod declares that a Volume is available using the `volumes` field in the YAML. In this example, the volume is named `sharedspace`. The `name` subfield of `volumes` is critical because this name is used by the containers to declare that they want to mount this Volume. This Volume example also declares a `hostPath` with a `path` value of `/var/sharedPodDir`. The `hostPath` declares the path on the host Node that is the location of this shared directory. With this specification, a directory called `sharedPodDir` will be created in `/var` on the Node hosting the Pod.

The containers in this Pod that want to mount this Volume have an easier declaration process. They add a `volumeMounts` field that contains a `name` and a `mountPath` as subfields. The `name` field must match the name that was used when the Volume was declared in the Pod's `volumes` section. The `mountPath` declares the path in the container's filesystem that will be the location of this shared directory when accessed by the container application. It is worth noting that it is perfectly acceptable for the container to have a different `mountPath` than the `hostPath`. Furthermore, each container in the Pod can have a different `mountPath` value to the same shared directory. In the

next section, we provide an overview of PersistentVolumes, which provide a more durable version of Volumes.

Persistent Volumes

Most **Twelve-Factor Apps** discourage the use of disk-based persistent storage because it's difficult to maintain concurrency at scale without specific guarantees for distributed read and write semantics. Kubernetes extends beyond just 12-factor apps, as we have seen with DaemonSets and StatefulSets, which do have a legitimate need for persistent storage.

StatefulSets such as databases, caches, and database services, or DaemonSets such as monitoring agents require their Pods to have access to storage that is more permanent and survives across the lifetimes of multiple Pods. For this purpose, Kubernetes provides PersistentVolume storage support. There are many implementations of PersistentVolume support for Kubernetes, and they typically are built on top of some form of remote network storage capability.

Support for PersistentVolumes in most Kubernetes clusters begins with the cluster administrator creating **PersistentVolumes** and making these available in the cluster. The Pods in the cluster acquire access to the PersistentVolume through a resource abstraction called a **PersistentVolumeClaim** that is also represented as a YAML document:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: my-pv-claim
  labels:
    app: nginx
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

In the preceding example, the **PersistentVolumeClaim** is a request for five gigabytes of storage that can be attached as read/write by only a single Node. When this request is created, Kubernetes takes responsibility for mapping the request to an actual **PersistentVolume** that is available.

After this has been accomplished, a Pod can access the PersistentVolume storage through the PersistentVolumeClaim, as shown in the following Pod example:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  volumes:
    - name: persistentsharedspace
      persistentVolumeClaim:
        claimName: my-pv-claim
  containers:
    - name: nginx
      image: nginx:1.7.9
      volumeMounts:
        - mountPath: "/containerMount"
          name: persistentsharedspace
  ports:
    - containerPort: 80
```

As this example shows, the `volumes` section of the YAML declares a `persistentVolumeClaim` with a `claimName` of `my-pv-claim`. The name used to reference this PersistentVolume in the Pod is `persistentsharedspace`. The containers in the Pod can now reference this PersistentVolume using a `volumeMounts` section just like they did for mounting basic volumes. This example also shows that the containers add a `volumeMounts` field that contains a `name` and a `mountPath` as subfields. The `name` field contains the value of `persistentsharedspace`, which matches the `name` value provided in the `name` subfield of the `volumes` section. As was the case when containers were attaching basic volumes, the `mountPath` declares the path in the container's filesystem that will be the location of this shared directory when accessed by the container application.

Matching PersistentVolumeClaims (PVCs) with PersistentVolumes (PVs) can be a frustrating task sometimes. PVCs are matched based on a number of factors, including:

Requested capacity

A PVC will match a PV only if the capacity requested in `spec.resources.requests.storage` matches the capacity declared in the PV.

Storage class

A PVC will match a PV only if the `StorageClass` is consistent; or, for the case in which a PVC has omitted the `StorageClass`, a PV that matches all other attributes and uses the default `StorageClass` is available.

Labels

A PVC will match a PV only if the labels assigned to the PVC align with the labels declared on the PV. Typically, labels reflect the associations between microservices and PVCs/PVs (such as `app=portfolio`).

Access mode

A PVC will match a PV only if the access mode (`ReadWriteOnce` [`RWO`], `ReadOnlyMany` [`ROX`], or `ReadWriteMany` [`RWX`]) is consistent between the claim and the volume.

In the next section, we introduce the concept of `ConfigMaps`, which is the Kubernetes mechanism for passing configuration information into containers.

ConfigMaps

In many cases, there needs to be a way to pass configuration information into your container-based applications. Kubernetes provides *ConfigMaps* as its mechanism to pass information into the containers it manages. `ConfigMaps` store configuration information as key-value pairs. The following is a sample `ConfigMap`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: custom-config
  namespace: default
data:
  customkey1: foo
  customkey2: bar
```

In this example, there are two new configuration keys: `customkey1` and `customkey2`. These keys hold the values `foo` and `bar`, respectively. To add this `ConfigMap` to your Kubernetes environment, save the example YAML as *configmap.yaml* and run the following `kubectl` command:

```
$ kubectl create -f configmap.yaml
```


This snippet shows how you create a ConfigMap named `custom-config`, and the source of the ConfigMap is the file `configmap.yaml`. After the ConfigMap is created, there are two general ways for using it. One is the creation of a file for each key-value, and the other is setting environment variables.

ConfigMap Keys as Files

With the ConfigMap-keys-as-files approach, a file is created with the name of the key as its filename, and that file will contain the value of the key. You use `volumeMounts` to mount these files, and the ConfigMap is accessed through a Volume. The following example illustrates this ConfigMap approach:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmapexample-volume
spec:
  containers:
    - name: configmapexample-volume
      image: busybox
      command: [ "/bin/sh", "-c", "ls /etc/bt_config ; cat
/etc/bt_config/customkey1 ; echo" ]
      volumeMounts:
        - name: config-volume
          mountPath: /etc/bt_config
          imagePullPolicy: IfNotPresent
  volumes:
    - name: config-volume
      configMap:
        name: custom-config
```

As shown here, a volume named `config-volume` is created by referencing a ConfigMap named `custom-config`. Note that this is the name of the ConfigMap created at the beginning of this section. A `volumeMount` is then used to mount the `config-volume` with a `mountPath` of `/etc/bt_config`. With this mounting defined, the keys from the ConfigMap are stored as files in `/etc/bt_config`. To demonstrate that the previous example does in fact create these files, a `command` option has been added to the example that overrides the default command provided by the container image. The new `command` option prints the contents of the `/etc/bt_config` directory to show the key files were indeed created. The command also prints the contents of the `/etc/bt_config/customkey1` file to prove that it really contains the value `foo`.

To run this example, save the file as *configmapPod-volume.yaml*. Make sure that you have already created the ConfigMap from *configmap.yaml* as described at the beginning of this section. When you complete these steps, you can now run the example by doing the following:

```
$ kubectl apply -f configmapPod-volume.yaml
```

To view the output from the container, use the `kubectl logs <pod Name>` command, as follows:

```
$ kubectl logs configmapexample-volume
```

You will then see the following output confirming that the keys were created:

```
customkey1
customkey2
foo
$
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod configmapexample-volume
```

ConfigMap Keys as Environment Variables

Using the ConfigMap-keys-as-environment-variables approach, the keys are stored as environment variables and their values are stored as the value of the environment variable. An `env` section is added to the container portion of the Pod YAML. The following example illustrates the environment variable–based approach:

```
apiVersion: v1
kind: Pod
metadata:
  name: configmapexample
spec:
  containers:
  - name: configmapexample
    image: busybox
    command: [ "/bin/sh", "-c", "echo customkey1: $(KEY1)
      customkey2: $(KEY2) " ]
    env:
      - name: KEY1
        valueFrom:
          configMapKeyRef:
            name: custom-config
            key: customkey1
      - name: KEY2
```

```
valueFrom:
  configMapKeyRef:
    name: custom-config
    key: customkey2
```

You can see that an `env` section has been added to the `containers` section. The `env` section declares two keys: `KEY1` and `KEY2`. Each of these will be environment variables that are created. The `KEY1` environment variable is assigned the value stored in `customkey1` of the `custom-config` `ConfigMap` using the `valueFrom` and `configMapKeyRef` constructs. In a similar fashion, the `KEY2` environment variable receives the value from `customkey2` in the `custom-config` `ConfigMap`. The environment variables `KEY1` and `KEY2` can be used in the command that is run by the container image. To illustrate this, the previous example prints the values for `KEY1` and `KEY2` when the container runs the command line provided.

To run the example, save the file as `configmapPod-env.yaml`. Make sure that you have already created the `ConfigMap` from `configmap.yaml`, as described at the beginning of this section. When you've completed these steps, you can now run the example yourself by doing the following:

```
$ kubectl apply -f configmapPod-env.yaml
```

To view the output from the container, use the `kubectl logs <pod Name>` command, as follows:

```
$ kubectl logs configmapexample
```

You will then see the following output confirming that the keys were created:

```
customkey1: foo customkey2: bar
$
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod configmapexample
```

As you have seen in this example, `ConfigMaps` are extremely valuable for injecting configuration information into containers. But what if you need to inject some form of sensitive information into a container, such as a user ID and password? A `ConfigMap` would not be appropriate for storing this type of information, because it needs to remain hidden. For this type of data, Kubernetes provides a `Secrets` object, which we look at in the next section.

Secrets

Kubernetes provides the *Secrets* construct for dynamic injection of sensitive information into containers. As a best practice, sensitive information such as user identification, passwords, and security tokens should not be bundled directly into container images, because this provides a greater opportunity for this sensitive information to be compromised. Instead, you should dynamically inject sensitive information into containers in a Pod by using the Secrets construct.

The Secrets construct works in a fashion that is very analogous to ConfigMaps. Similar to ConfigMaps, you first create a Secret. After you've created the Secret, Pods provide mechanisms to inject the sensitive information that is stored in the Secret into a running container process. As an example, let's assume that you have a username and password that is needed by your container, and that the username is `admin` and the password is `letbradin`. To create a Secret representing this information, you store both pieces of information as text files and run the `kubectl create secret` command, as follows:

```
$ echo -n "admin" > username.txt
$ echo -n "letbradin" > password.txt
$ kubectl create secret generic webserver-credentials
--from-file=./username.txt --
from-file=./password.txt
```

After you've created the Secret, there are two general ways for using it: creating a file for each key value, and setting environment variables.

Secret Keys as Files

With the Secret-keys-as-files approach, a file is created with the name of the key as its filename, and the file will contain the value of the Secret represented by the key. You use `volumeMounts` to mount these files, and the sensitive data contained in the Secret is accessed through a Volume. The following example illustrates this Secret approach:

```
apiVersion: v1
kind: Pod
metadata:
  name: secretexample-volume
```

```

spec:
  containers:
    - name: secretexample-volume
      image: busybox
      command: [ "/bin/sh", "-c", "ls /etc/bt_config ; cat
/etc/bt_config/web_password ; echo" ]
      volumeMounts:
        - name: secret-volume
          mountPath: /etc/bt_config
          imagePullPolicy: IfNotPresent
  volumes:
    - name: secret-volume
      secret:
        secretName: webserver-credentials
        items:
          - key: password.txt
            path: web_password
          - key: username.txt
            path: web_username

```

Here, a Volume named `secret-volume` is created by referencing a Secret named `webserver-credentials`. Note that this is the name of the Secret created at the beginning of this section. You then use a `volumeMount` to mount the `secret-volume` with a `mountPath` of `/etc/bt_config`. With this mounting defined, the keys from the Secret are stored as files in `/etc/bt_config`. To demonstrate that the preceding example does in fact create these files, a `command` option has been added that overrides the default command provided by the container image. The new `command` option prints the contents of the `/etc/bt_config` directory to show that the key files were indeed created. The command also prints the contents of the `/etc/bt_config/web_password` file to prove that it really contains the value `letbradin`.

To run this example, save the file as `secretPod-volume.yaml`. Make sure that you have already created the `webserver-credentials` Secret, as described at the beginning of this section. When you've done these steps, you can run the example by doing the following:

```
$ kubectl apply -f secretPod-volume.yaml
```

To view the output from the container, use the `kubectl logs <pod Name>` command, as follows:

```
$ kubectl logs secretexample-volume
```

You should then see the following output confirming that the keys were created:

```
web_password
web_username
"letbradin"
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod secretexample-volume
```

Secret Keys as Environment Variables

With the Secret-keys-as-environment variables approach, the keys are stored as environment variables, and the sensitive values they represent are stored as the values of the environment variables. An `env` section is added to the container portion of the Pod YAML. The following example illustrates the environment variable-based approach for accessing Secret data:

```
apiVersion: v1
kind: Pod
metadata:
  name: secretexample
spec:
  containers:
  - name: secretexample
    image: busybox
    command: [ "/bin/sh", "-c", "echo user name: $(USER)
password: $(PASS) " ]
    env:
      - name: USER
        valueFrom:
          secretKeyRef:
            name: webserver-credentials
            key: username.txt
      - name: PASS
        valueFrom:
          secretKeyRef:
            name: webserver-credentials
            key: password.txt
```

Note that an `env` section has been added to the `containers` section. The `env` section declares two keys: `USER` and `PASS`. Each of these will be environment variables that are created. The `USER` environment variable is assigned the value stored in `username.txt` of the `webserver-credentials` Secret using the `valueFrom` and `secretKeyRef` constructs. In a similar fashion, the `PASS` environment variable receives the value from `password.txt` in the `webserver-credentials` Secret. As shown in the preceding example, you can use the environment variables `USER` and `PASS` in the command that

is run by the container image. To illustrate this, the above example prints the values for `USER` and `PASS` when the container runs the command line provided in this example.

To run the example, save the file as `secretPod-env.yaml`. Make sure that you have already created the `webserver-credentials` Secret, as described at the beginning of this section. When you've completed these steps, you can run the example by doing the following:

```
$ kubectl apply -f secretPod-env.yaml
```

To view the output from the container, use the `kubectl logs <pod Name>` command, as follows:

```
$ kubectl logs secretexample
```

You should then see the following output confirming that the keys were created:

```
user name: "admin" password: "letbradin"
$
```

To delete the Pod that you just created, run the following command:

```
$ kubectl delete pod secretexample
```

NOTE

We Aren't Done Yet: Securing Your Secrets!

Here's a very critical safety tip that we must mention: the default in Kubernetes is that secrets are encoded but not encrypted. To truly secure your secrets, we strongly recommend that you **enable encryption of your secrets**. This topic is beyond the scope of this book.

Image Registry

We've focused all of our attention thus far on the declarative resources for Kubernetes—but where are containers in all of this?

You've already seen images referenced by the Pod:

```
...
    image: nginx:1.7.9
...
```

But what is this statement? Let's break it down. Every image reference follows the form `repository:tag`. Let's take a look at each of these:

repository

The *repository* reflects the logical name of an image. Typical examples include *nginx*, *ubuntu*, or *alpine*. These repositories are shorthand for *docker.io/nginx*, *docker.io/ubuntu*, and *docker.io/ubuntu*. The Docker runtime knows to request the bytes for these layers from the **Docker Registry**. You can also store images in your own registry.

tag

This is a label that denotes a particular version. Tags can follow any pattern that works for your delivery process. We recommend adopting *semver*, Git commit hashes, or a combination of the two. You will also find the convention to use *latest* to denote the most recent version available.

An actual image layer can have multiple *tags* and can be even be referenced by multiple *repository:tag* combinations at the same time.

As you create a Continuous Integration/Continuous Delivery (CI/CD) pipeline, you will be building images for all of the changes to your application source code. Each image then is tagged and pushed into an image registry, which your Kubernetes cluster can access. Some options for private image registries include the following:

JFrog Artifactory

Artifactory is an artifact management tool. Artifactory supports repositories for things like Java libraries (**.jar*, **.war*, **.ear*) or node modules along with your container images. Artifactory offers both an open source and commercial version. With it, you can extend your usage from build artifacts to Docker images. You can deploy Artifactory via Helm charts in your Kubernetes cluster as well, such as into your own IBM Cloud Private Kubernetes cluster.

IBM Cloud Container Registry

An IBM-managed private container registry.

IBM Cloud Private built-in cluster registry

A built-in container registry available by default in your on-premises Kubernetes cluster and tied to the same Role-Based Access Control (RBAC) used for Namespaces.

Docker Hub

Default source registry used by the Docker runtime if no explicit registry host is provided.

For Kubernetes to access an image registry, you must create an *Image Pull Secret*. Image Pull Secrets store the credentials to access a particular image registry. A Pod might specify its Image Pull Secret to enable the container runtime to pull the image to the host where the Pod is running. Alternatively, you can store the Image Pull Secret such that any Pod deployed within a given Namespace has access to the same image registry. [Chapter 4](#) presents an example of this and also provides an overview of Namespaces and how to use them.

Helm

We've talked about many kinds of Kubernetes resources that often work in concert to deliver a complete application. When you want to change values dynamically as part of a packaging or build step, there is no built-in way to override parameter values on the command line without editing the file. *Helm* is the Kubernetes package manager that enables you to create a package containing multiple templates for all of the resources that make up your application. Common Kubernetes resources that you would expect to find included in a Helm package are ConfigMaps, Deployments, PersistentVolumeClaims, Services, and many others.

A collection of templates is called a *Helm chart*. Helm provides its own command-line interface (`helm`) to deploy a chart and provide options for parameter values, either via command line or a single file named *values.yaml* by convention.

IBM Cloud Private includes a [rich catalog of Helm charts](#) (see [Figure 3-1](#)), which is available in the community edition, as well as commercially supported versions.

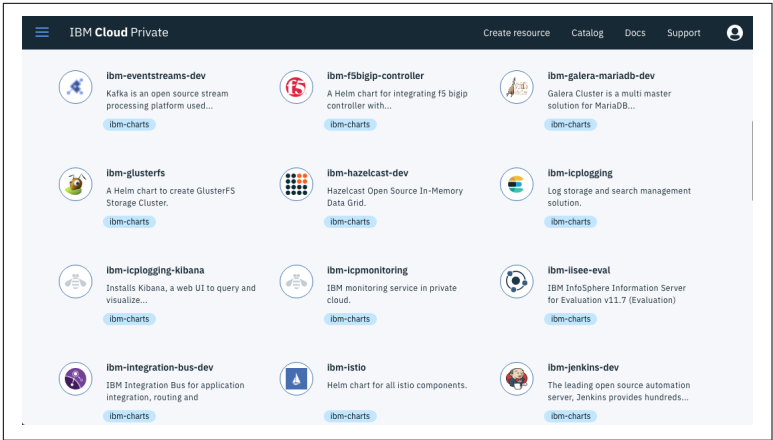


Figure 3-1. IBM Cloud Private provides a rich catalog of content, delivered as Helm charts. Users can consume community charts or add their own.

For users of IBM Cloud Kubernetes Service, you will find **Helm content easily browsable** via the **Solutions > Helm** section of the **User Interface**, as depicted in Figure 3-2.

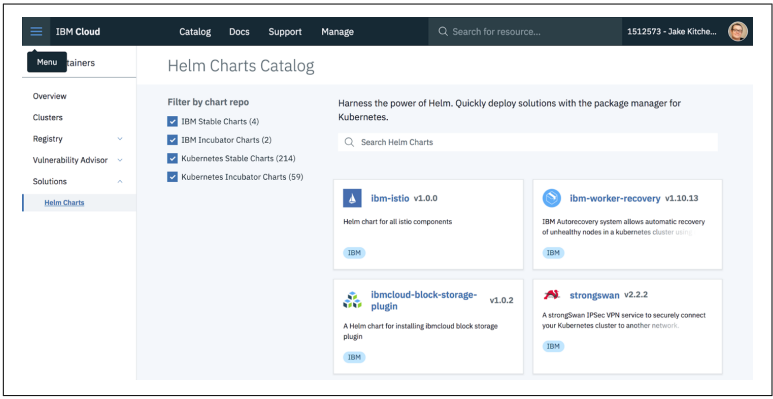


Figure 3-2. IBM Cloud Kubernetes Service exposes Helm charts in the catalog as well as providing a consistent packaging and deployment capability for private and public clusters.

You can also take advantage of a broad range of community changes from **KubeApps**. **Figure 3-3** shows a snapshot of the KubeApps catalog.

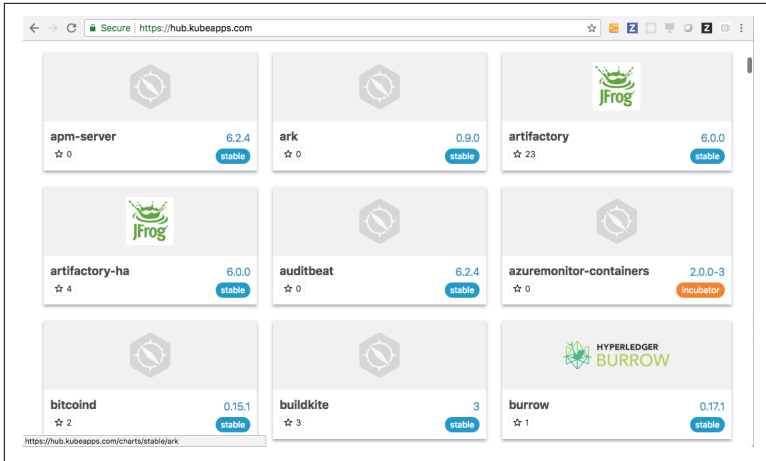


Figure 3-3. The *kubeapps.com* catalog provides a community-driven way to create and distribute Helm charts for any Kubernetes application.

Chapter 4 provides examples of Helm charts for supporting middleware. You can also create Helm charts for your own applications, but that is beyond the scope of this book.

Next Steps

At this point we have covered a considerable amount of the key constructs and high-level abstractions that Kubernetes provides. In Chapter 4, we explore the creation of an enterprise-level production application.

Introducing Our Production Application

In this chapter, we provide several examples that walk you through running an enterprise-quality application on Kubernetes. We begin with a discussion about microservices in Kubernetes. We then introduce several important concepts, explaining each one as we go.

We invite you to prepare your own cluster to run these examples yourself. We explain a few options for running your own Kubernetes cluster and configuring your command-line interface (CLI) in [Appendix A](#).

Our First Microservice

Lao Tzu is credited with the saying, “A journey of a thousand miles begins with one step.” So it is with microservices. Let’s dig in on our first microservice by building an example end to end. We will then expand to a complete application.

We recommend two foundational learning resources for building great microservices: [Twelve-Factor Apps](#), and Stability and Availability Patterns described in the book [Release It!](#).

Each factor in [The Twelve-Factor manifesto](#) isolates one aspect of building scalable, manageable microservices. Characteristics such as *declarative format* and *separation of concerns* are emphasized.

One of the Twelve Factors that enables scalable web services is the externalization of configuration via the process environment. Many aspects of Kubernetes can be directly mapped back to Twelve-Factor principles. For example, as you've already seen in [Chapter 3](#), Kubernetes defines ConfigMaps and Secrets to provide configuration to your app. Configuration data can take the form of process environment variables, configuration files, or even Transport Layer Security (TLS) certificates and keys.

A microservice is only as good as its dependencies; or, more precisely, *as good as how it anticipates and responds to failures within its dependencies*. If you are new to building microservices, we strongly recommend reading *Release It!*, which introduces design patterns such as [Circuit Breaker](#), [Bulkhead](#), Fail Fast, and Timeouts. Each of these patterns focuses on addressing distributed systems behaviors like chain reactions, cascading failures, and Service-Level Agreement (SLA) inversion. Each of these failure patterns is the result of nested dependencies of microservices upon microservices upon other microservices. The corresponding availability and stability patterns (Circuit Breaker, Bulkhead, Fail Fast, etc.) are pragmatic, language-agnostic approaches to responding gracefully with service degradation rather than complete system outage.

We will now use an application built to demonstrate many aspects of containerized applications, known as StockTrader. You can browse the source code for this application at this book's [GitHub page](#).

Our portfolio application connects to several other services, including a database, messaging service, and other microservices. We take advantage of various resources from Kubernetes to implement this application. We've already talked about the declarative model of Kubernetes. Let's talk through the moving parts and then dig into the details of how we apply a declarative model for our Stock Trader application.

We are going to build our portfolio end to end, starting with ready-to-go source code, building an image, and deploying the container with Kubernetes. We'll configure its dependencies and reuse images for its dependent services which are already published and available on DockerHub.

All of our containers are deployed as Pods. How these Pods are managed by Kubernetes will be determined by their orchestration controller. For instance, a Deployment creates a ReplicaSet, which

expects stateless Pods; any failed Pod will be immediately rescheduled until the desired number of replicas are observed as Ready by Kubernetes. A StatefulSet creates Pods in an orderly fashion. Each Pod is expected to hold a partial or complete replica of data; so new Pods are scheduled only as existing Pods become Ready. Deployments are used for any Twelve-Factor microservice. As you've guessed, StatefulSets are used for data, messaging, and caching services. Of course, our StatefulSet Pods will also mount storage via PersistentVolumes.

Microservices consume dependencies in Kubernetes via Services, which offer a Domain Name System (DNS)-resolvable name that will be serviced by one or more Endpoints. Alternative service registration and discovery frameworks such as [Netflix OSS Eureka](#) or [Consul](#) can be used as well, but let's focus on what comes out of the box.

Namespaces

A Namespace in Kubernetes allows you to define isolation for microservices. You can create a Namespace through YAML or the command line. The following is a simple YAML file that describes the stock-trader Namespace resource used by our Stock Trader application:

```
apiVersion: v1
kind: Namespace
metadata:
  name: stock-trader
```

To create the stock-trader Namespace, save the preceding example as *namespace.yaml* and then apply the document using `kubectl apply`:

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

Alternatively, you can use `kubectl create` to create the Namespace and thus skip the use of the YAML declaration:

```
$ kubectl create namespace stock-trader
```

You can always select the namespace when running commands using the flag `--namespace=[ns]` or `-n=[ns]`. Alternatively, you can update the default namespace used for all commands by updating your context:

```
$ kubectl config set-context $(kubectl config current-context) \  
  --namespace=stock-trader
```

Kubernetes groups many of the concepts discussed earlier into Namespaces. Namespaces allow you to isolate applications on your cluster. With this isolation, you can control the following:

Role-Based Access Control (RBAC)

Defines what users can view, create, update, or delete within the Namespace. We get into more details about how RBAC works in Kubernetes in just a bit.

Network policies

Defines isolation to tightly manage incoming network traffic (network ingress) and outgoing network traffic (network egress) communication between Pods.

Quota management

Controls the resources allowed to be consumed by Pods within the Namespace. Controlling quota allows you to ensure that some teams do not overwhelm the capacity available within the cluster.

Workload isolation by node

Namespaces can work with **Admission Controllers** to limit Pods to running on specific nodes in your cluster.

Workload readiness or provenance

Namespaces can work with Admission Controllers to allow only certain images (by whitelisting, image signatures, etc.) from running in the context of the Namespace.

ServiceAccount

When you interact with your cluster, you often represent yourself as a user identity. In the world of Kubernetes, you build intelligence into the system to help it interact with its world. Many times, Pods might use the Kubernetes API to interact with other parts of the system or to spawn work like Jobs. When we deploy a Pod, it might interact with PersistentVolumes, the host filesystem, the host networking, or be sensitive to what operating system (OS) user it is given access to use for filesystem access. In most cases, you want to restrict the default permissions for a given Pod from doing anything more than the absolute basics. Basically, the less surface area that a

Pod is given access to in the cluster, the host OS, the networking layer, and your storage layer, the fewer attack vectors there will be that can be exploited.

For a Pod to interact with the system, it is assigned a ServiceAccount. Think of this like a functional identity. ServiceAccounts are subjects that can authenticate with the system via tokens and that are authorized for certain behaviors.

We introduce this topic now because we create a ServiceAccount shortly as part of our Namespace.

PodSecurityPolicy

When Pods are deployed into a Namespace, the ServiceAccount affords them various privileges. A PodSecurityPolicy controls what privileges are allowed. We will demonstrate how a PodSecurityPolicy grants a container special OS permissions, including restricting specific Linux kernel capabilities with the potential for very fine-grained controls based on your organization's security standards.

As we deploy our database, we will create a PodSecurityPolicy to be used by a particular ServiceAccount to ensure that our database has the required permissions to run in a container.

For users of IBM Cloud Kubernetes Service, you will find that PodSecurityPolicy is enabled by default. There are `privileged-psp-user` and `restricted-psp-user` PodSecurityPolicy options, as well as corresponding RBAC included in all clusters.¹

Deploying a Containerized Db2 Database as a StatefulSet

To save our portfolio data, let's deploy a database. IBM Db2 is a battle-tested, scalable, multiplatform database designed for mission-critical workloads. We will deploy Db2 as a container. Our Db2 service will run as a StatefulSet, with a mounted PersistentVolume to store its data. Any time the Pod or worker node fails, the Kubernetes scheduler will automatically reschedule the Pod onto another

¹ [“Configuring pod security policies”](#), IBM Cloud documentation.

healthy node. To ensure that our underlying storage is also highly available, we use a distributed file system called GlusterFS.

NOTE

For more information about GlusterFS and persistence for containers, see [Chapter 7](#).

In this section, we demonstrate how to isolate the Db2 database container from other containers in its own Namespace. We discuss alternative deployment paths for Db2 on Kubernetes in a [devWorks recipe](#). We then define the permissions that are granted when it runs. Next, we define a Service to make the database available to Pods in our application Namespace.

The examples referenced in this section are available on GitHub. You can clone these examples locally using the following command:

```
$ git clone https://github.com/kubernetes-in-the-enterprise/
portfolio-database.git
```

Db2 is packaged as a Helm chart. Think of a Helm chart as a fast way to create several parts of a Kubernetes application in one fell swoop (for more information on Helm charts, see [Chapter 3](#)). Here's what we are about to do:

1. Create a Namespace to protect our database containers.
2. Configure a Secret to store credentials to access a remote image registry (Docker Store) to pull the free Db2 image into our cluster.
3. Configure the PodSecurityPolicy for our Namespace to allow certain permissions for our database container.
4. Configure the ServiceAccount for the Namespace to include the image pull Secret and use the PodSecurityPolicy.
5. Configure the Helm Repository for our client to deploy the chart into our cluster.

Creating the Namespace for the database

Creating a Namespace allows us to specify custom security policies for your database. We may also choose to later restrict the incoming or outgoing network communication for Pods deployed in the Namespace. Furthermore, we might want to define a quota for the

Namespace to ensure that our database is given preferential treatment by the cluster for available capacity. Let's look at the code to do this:

```
$ kubectl create namespace stock-trader-data
$ kubectl config set-context $(kubectl config current-context)\
  --namespace=stock-trader-data
```

Creating a Custom PodSecurityPolicy for the Database ServiceAccount

Our Db2 container requires a few additional capabilities that are typically restricted by default. Let's now create a PodSecurityPolicy that we will make available to the ServiceAccount in our stock-trader-data Namespace.

You can think of a PodSecurityPolicy as the “house rules” for a container. When the Pod is launched, it is matched to a PodSecurityPolicy, which governs all containers that are a part of the Pod. As shown in the code example that follows, we create the following policy that prevents privileged execution but allows privileged escalation, allows access to various kinds of Volumes, and applies restrictions to what users and groups may be used as the identity for our container. These rules help reduce the surface area for attacks that could be used by a malicious container to gain access to the host.

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: db2-privileges
spec:
  privileged: false
  allowPrivilegeEscalation: true
  allowedCapabilities:
  - 'SYS_RESOURCE'
  - 'IPC_OWNER'
  - 'SYS_NICE'
  hostIPC: true
  hostNetwork: false
  volumes:
  - 'configMap'
  - 'emptyDir'
  - 'projected'
  - 'secret'
  - 'downwardAPI'
  - 'persistentVolumeClaim'
```

```

runAsUser:
  rule: 'RunAsAny'
seLinux:
  rule: 'RunAsAny'
seLinux:
  rule: 'RunAsAny'
supplementalGroups:
  rule: 'MustRunAs'
  ranges:
    - min: 1
      max: 65535
fsGroup:
  rule: 'MustRunAs'
  ranges:
    - min: 1
      max: 65535

```

In this policy, we also grant specific capabilities that we will allow specifically for Db2. If we allowed the container to run as privileged, it would have access to all capabilities offered by the Linux kernel. Here, we reduce the list to only that which is required.

To understand all of the available capabilities to the Linux kernel, consult the manual page for capabilities on your own OS or at the [Linux man page](#).

We apply the security policy with our favorite command, `kubectl apply`. Unlike most of the resources we've discussed so far, the `PodSecurityPolicy` is cluster scoped. As a result, there is no need to specify a Namespace in the document or the command:

```
$ kubectl apply -f db2-pod-security-policy.yaml
podsecuritypolicy.policy "db2-privileges" created
```

```
$ kubectl get psp
NAME          DATA          CAPS          \
SELINUX      RUNASUSER
FSGROUP      SUPGROUP      READONLYROOTFS  VOLUMES
db2-privileges  false        SYS_RESOURCE,IPC_OWNER,SYS_NICE  \
RunAsAny      RunAsAny
MustRunAs     MustRunAs     false
configMap,emptyDir,projected,secret,downwardAPI,\
persistentVolumeClaim
default       false          \
RunAsAny
MustRunAsNonRoot  MustRunAs  MustRunAs  false
configMap,emptyDir,projected,secret,downwardAPI,\
persistentVolumeClaim
privileged     true         *          \
```

```
RunAsAny RunAsAny
RunAsAny RunAsAny false *
```

Creating an Image Pull Secret to Access the Db2 Container Image

In most environments, you will use a private image registry, which makes all of your images available. To enable Kubernetes to pull your images, you create a special kind of Secret known as an *Image Pull Secret*. These types of Secrets follow a particular structure and define your credentials to access your image registry. When the Pod is launched on a host, the Image Pull Secret provides the container runtime with the necessary access to pull the image from the registry before the container is started.

Db2 is made available to developers via the Docker Store. You can visit the [Docker Cloud to subscribe to Db2](#) (at no charge).

After you've subscribed to this free image, you can create the necessary Image Pull Secret to enable your cluster to access the image. You can either use your password or generate an API key.

NOTE

For instructions on how to create an API key, see [Appendix D](#).

```
$ kubectl create secret docker-registry dockerhub \
  --docker-username=<userid> \
  --docker-password=<API key or user password> \
  --docker-email=<email> \
  --namespace=stock-trader-data
```

Creating the Namespace also creates the ServiceAccount. We now associate the Image Pull Secret with the ServiceAccount using the patch command on kubectl. The patch command accepts a snippet of JSON and applies it to the target object.

The following command updates the default ServiceAccount in our stock-trader-data Namespace to use Image Pull Secret dockerhub:

```
$ kubectl patch --namespace=stock-trader-data serviceaccount \
  default -p '{"imagePullSecrets": [{"name": "dockerhub"}]}'

$ kubectl describe serviceaccount default
Name:          default
Namespace:    stock-trader-data
```

```
Labels:           <none>
Annotations:      <none>
Image pull secrets: dockerhub
Mountable secrets: default-token-7qvqq
Tokens:           default-token-7qvqq
Events:           <none>
```

Configuring the ServiceAccount to use the PodSecurityPolicy and the Image Pull Secret

Be sure that you are still using the `stock-trader-data` Namespace:

```
$ kubectl config set-context $(kubectl config current-context) \
  --namespace=stock-trader-data
```

All Namespaces come with a default ServiceAccount. Recall that a ServiceAccount provides a functional identity that is used by Pods to interact with the rest of the cluster. Just like a user, ServiceAccounts are assigned to roles that allow them to interact with the Kubernetes API.

Let's create a ClusterRole that allows access to our PodSecurityPolicy. The ClusterRole is part of Kubernetes RBAC. In general, roles define a set of verbs, resources types, and resources. When a user identity or ServiceAccount is assigned to a role, it will be able to execute the verbs (API invocations) against the resource types or specific resources. ClusterRoles are described in more detail in [Chapter 7](#).

We also create the ClusterRoleBinding, which associates the ClusterRole with our specific ServiceAccount in the `stock-trader-data` Namespace. Notice that we have used the “`---`” separator at the top of the file and between them to denote multiple objects in a single file:

```
---
kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: db2-privileges-cluster-role
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resourceNames: ['db2-privileges']
---
kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
```

```
metadata:
  name: db2-privileges-cluster-role-binding
roleRef:
  kind: ClusterRole
  name: db2-privileges-cluster-role
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: default
  namespace: stock-trader-data
```

We can apply both objects in a single command:

```
$ kubectl apply -f db2-pod-security-policy-cluster-role.yaml

clusterrole.rbac.authorization.k8s.io "db2-privileges-cluster\
-role" configured
clusterrolebinding.rbac.authorization.k8s.io "db2-privileges-\
cluster-role-binding"
created
```

The default `ServiceAccount` in the `stock-trader-data` Namespace is now authorized to use the `PodSecurityPolicy` `db2-privileges` created in the preceding example.

Deploying Our Database

A Helm Repository collects a set of charts to be available for deployment. For this example, we are using the public IBM Helm Repository for charts. When you configure Helm, you need to add the definition of this repository, as shown here:

```
$ export HELM_HOME=~/.helm
$ helm repo add ibm-charts \
  https://raw.githubusercontent.com/IBM/charts/master/repo/
  stable/
$ helm repo update
```

Each Helm chart exposes a set of parameters. For `Db2`, we just need to provide the expected size of our persistent store, the name of our storage class, and options for the `Db2` instance and database names.

Most of these values will be the same for any Kubernetes cluster that you use. However, the value for the `storageClassName` will vary based on your environment. Here, we are using an IBM Cloud Private cluster with `GlusterFS` deployed, with a storage class named `glusterfs`. On IBM Cloud Kubernetes Service, consider using the `ibmc-block-gold` `StorageClass`.

With the Db2 Helm chart, you can choose to configure the database in one of two modes:

- StatefulSet with a single Pod, with mounted persistent storage. Under this configuration, if the Pod becomes unhealthy or the node fails, a new Pod will be scheduled onto a healthy node, and the original PersistentVolume will be remounted into the container. No data should be lost that was already committed to disk.
- StatefulSet with two Pods, configured for High Availability/ Disaster Recovery (HADR), with mounted persistent storage. Under this configuration, both Pods maintain an active replica of the database. Placement rules will ensure that the Pods are scheduled and deployed on separate hosts. To support the election of a primary database replica, a second StatefulSet is automatically configured for `etcd`, a distributed key-value store. If either Pod experiences a failure, the remaining Pod will detect that the primary replica has become unhealthy and use the warm backup of the database to provide continuity of service for incoming requests.

Let's begin with the single-Pod StatefulSet. Here, we configure the Helm chart with the database instance name, password, and create a default database named STRADER:

```
$ kubectl config set-context $(kubectl config current-context) \
  --namespace=stock-trader-data>

$ export HELM_HOME=~/.helm>

# Optionally, fetch the chart prior to installing it
$ helm fetch ibm-charts/ibm-db2oltp-dev

$ helm install --name stocktrader-db2 ibm-charts/ibm-db2oltp-dev \
  --tls \
  --set db2inst.instname=db2inst1 \
  --set db2inst.password=ThisIsMyPassword \
  --set options.databaseName=STRADER \
  --set persistence.useDynamicProvisioning=true \
  --set dataVolume.size=20Gi \
  --set dataVolume.storageClassName=glusterfs
```


NOTE

To run Db2 in the HADR configuration, you might need to make additional updates to your worker nodes. As a highly optimized database, special considerations to the Linux kernel parameters are made for best performance. You might need to validate or update these settings according to the documentation.²

We can configure the database with HADR support by setting the flag `hadr.enabled=true`. Setting this flag causes the Helm chart to configure additional resources including a StatefulSet for etcd (a distributed key-value store) and run additional configuration as the container starts up:

```
$ kubectl config set-context $(kubectl config current-context) \
  --namespace=stock-trader-data

$ export HELM_HOME=~/.helm

# Optionally, fetch the chart prior to installing it
$ helm fetch ibm-charts/ibm-db2oltp-dev

$ helm install --name stocktrader-db2 ibm-charts/ibm-db2oltp\
-dev
  --tls \
  --set db2inst.instname=db2inst1 \
  --set db2inst.password=ThisIsMyPassword \
  --set options.databaseName=STRADER \
  --set persistence.useDynamicProvisioning=true \
  --set dataVolume.size=2Gi \
  --set hadr.enabled=true \
  --set hadr.useDynamicProvisioning=true \
  --set dataVolume.storageClassName=glusterfs \
  --set hadrVolume.storageClassName=glusterfs \
  --set etcdVolume.storageClassName=glusterfs

NAME:      stocktrader-db2
LAST DEPLOYED: [...]
NAMESPACE: stock-trader-data
STATUS:    DEPLOYED

RESOURCES:
==> v1/Service
NAME                                TYPE          CLUSTER-IP \
EXTERNAL-IP  PORT(S)
AGE
```

² IBM Db2 11.1 Kernel Parameter Minimums for Interprocess Communication.

```

stocktrade-ibm-db2oltp-dev-db2 NodePort 10.0.0.95 <none>
50000:31187/TCP,55000:31392/TCP 2s
stocktrade-ibm-db2oltp-dev ClusterIP None <none>
50000/TCP,55000/TCP,60006/TCP,60007/TCP 2s
stocktrade-ibm-db2oltp-dev-etcd ClusterIP None <none>
2380/TCP,2379/TCP 2s

```

```
==> v1beta2/StatefulSet
```

```

NAME DESIRED CURRENT AGE
stocktrade-ibm-db2oltp-dev 2 2 2s
stocktrade-ibm-db2oltp-dev-etcd 3 3 2s

```

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

```

NAME READY STATUS RESTARTS AGE
stocktrade-ibm-db2oltp-dev-0 0/1 Pending 0 1s
stocktrade-ibm-db2oltp-dev-1 0/1 Pending 0 1s
stocktrade-ibm-db2oltp-dev-etcd-0 0/1 Pending 0 1s
stocktrade-ibm-db2oltp-dev-etcd-1 0/1 Pending 0 1s
stocktrade-ibm-db2oltp-dev-etcd-2 0/1 Pending 0 1s

```

```
==> v1/Secret
```

```

NAME TYPE DATA AGE
stocktrade-ibm-db2oltp-dev Opaque 1 2s

```

```
==> v1/PersistentVolumeClaim
```

```

NAME STATUS VOLUME CAPACITY ACCESS \
MODES STORAGECLASS AGE
stocktrade-hadr-stor Pending glusterfs 2s

```

NOTES:

```

1. Get the database URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace stock-trader- \
  data -o jsonpath="{.spec.ports[0].nodePort}" \
  services stocktrade-ibm-db2oltp-dev)

```

```

  export NODE_IP=$(kubectl get nodes --namespace \
  stock-trader-data -o \
  jsonpath="{.items[0].status.addresses[0].address}")

```

```
  echo jdbc:db2://$NODE_IP:$NODE_PORT/sample
```

Regardless of which configuration you choose, you should find messages in the logs of the container indicating that our database, STRADER, was created:

```

...
(*) User chose to create STRADER database
(*) Creating database STRADER ...
DB20000I The CREATE DATABASE command completed successfully.
DB20000I The ACTIVATE DATABASE command completed successfully.

```

```
08/14/2018 01:54:14    0    0    SQL1026N  The database manager \
is already active.
SQL1026N  The database manager is already active.
### Enabling LOGARCHMETH1

Database Connection Information

Database server          = DB2/LINUX8664 11.1.3.3
SQL authorization ID    = DB2INST1
Local database alias    = STRADER

DB20000I  The UPDATE DATABASE CONFIGURATION command completed \
successfully.
...
```

We have now deployed our database container. In the next section, we describe how to enable applications running in other Namespaces to discover and connect to our database service.

Connecting to Our Database from Other Namespaces

For our portfolio microservice to connect to the database, we are relying on Kubernetes' built-in service registration and discovery based on DNS. Our Db2 Helm chart created a Kubernetes Service resource. All Kubernetes Services can be resolved via DNS by the name of the Service.

In this example, we named our Db2 Helm Release `stocktrader-db2`. Our Service name in the `stock-trader-data` Namespace is `stocktrade-ibm-db2oltp-dev`. Our portfolio microservice will be deployed in the `stock-trader` Namespace, so we will also define a second Service in the `stock-trader` Namespace that references the Service in the `stock-trader-data` Namespace. [Figure 4-1](#) illustrates these Services and their respective Namespaces.

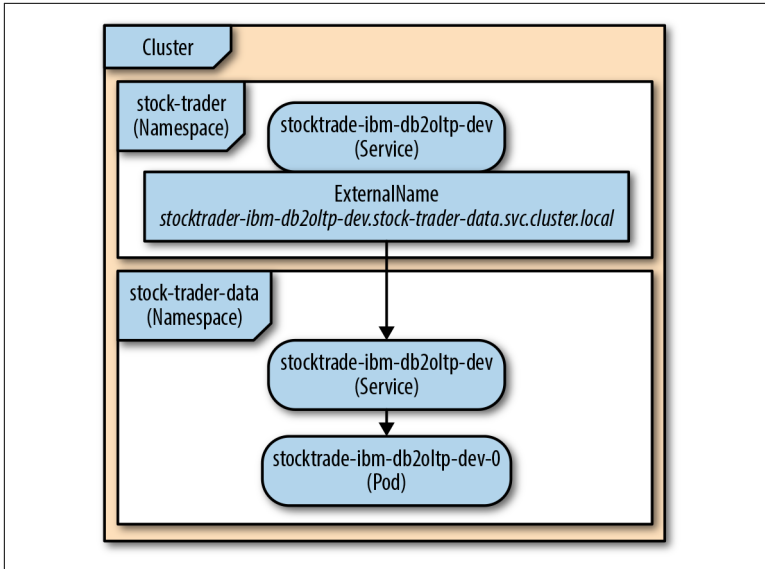


Figure 4-1. The database from *stock-trader-data* is exposed within *stock-trader-data* but aliased from *stock-trader* with an *ExternalName* Service.

In essence, we are creating an alias near our application (in the *stock-trader* Namespace) which we redirect to the Db2 Service in its own Namespace (*stock-trader-data*). As shown in [Figure 4-1](#), the original Service created by the deployment of the Db2 Helm chart was created within the Namespace. Let's now create a second Service, within the application's Namespace, which acts as an alias to reference the original. This useful technique lets you isolate microservices or collections of microservices in their Namespace but still allow access from consumers in other Namespaces. Interestingly, this approach still keeps all network access to these backend microservices isolated from external access.

```

kind: Service
apiVersion: v1
metadata:
  name: stocktrade-ibm-db2oltp-dev
  namespace: stock-trader
  labels:
    app: stocktrade-ibm-db2oltp-dev
spec:
  type: ExternalName
  externalName: stocktrade-ibm-db2oltp-dev.stock-trader-data \
  
```

```
.svc.cluster.local
ports:
- port: 50000
```

Create the Service in the stock-trader Namespace:

```
$ kubectl --namespace=stock-trader apply -f db2-service.yaml
```

You can now view both services by filtering by the label app=stocktrade-ibm-db2oltp-dev:

```
$ kubectl get svc -l app=stocktrade-ibm-db2oltp-dev \
--all-namespaces
NAMESPACE          NAME                                TYPE \
CLUSTER-IP         EXTERNAL-IP                         \
PORT(S)           AGE
stock-trader-data  stocktrade-ibm-db2oltp-dev         \
ClusterIP None
<none>
50000/TCP,55000/TCP,60006/TCP,60007/TCP  1h

stock-trader-data  stocktrade-ibm-db2oltp-dev-db2  NodePort \
10.0.0.208 <none>
50000:30567/TCP,55000:30057/TCP          1h

stock-trader       stocktrade-ibm-db2oltp-dev       ExternalName \
<none> stocktrade-ibm-db2oltp-dev.stock-
trader-data.svc.cluster.local 50000/TCP          6s
```

Populating Our Database with Application Schema

Now that our database is running, let's populate it with some application tables.

We will use a Kubernetes Job to configure the database. You can use Kubernetes Jobs for batch processing for all kinds of compute workloads, and they also serve as a useful tool for maintenance activities within a cluster. We use a Job, instead of `kubectl exec`, to run commands because it allows us to save our changes as source code and track updates along with the rest of the application.

Our Job resource reuses the same image as the actual database container. A ConfigMap is attached to a Volume to mount several scripts into the Job for execution. When the Job starts, the command executes these scripts and updates the database.

You can browse this file in the [GitHub repository for this book](#).

```

apiVersion: batch/v1
kind: Job
metadata:
  name: create-database-schema
spec:
  template:
    spec:
      containers:
        - name: create-database-schema
          image: store/ibmcorp/db2_developer_c:11.1.3.3a-x86_64
          command: [ "/bin/sh", "-c", "/scripts/db2-setup.sh" ]
          volumeMounts:
            - name: db2-createschema
              mountPath: /scripts
          securityContext:
            capabilities:
              add: ["SYS_RESOURCE", "IPC_OWNER", "SYS_NICE"]
          env:
            - name: LICENSE
              value: "accept"
            - name: DB2INSTANCE
              value: db2inst1
            - name: DB2INST1_PASSWORD
              valueFrom:
                secretKeyRef:
                  name: stocktrade-ibm-db2oltp-dev
                  key: password
            - name: DB2_SERVICE_NAME
              value: stocktrade-ibm-db2oltp-dev
            - name: DBNAME
              value: strader
          restartPolicy: Never
          volumes:
            - name: db2-createschema
              configMap:
                name: db2-createschema
                defaultMode: 0744
      backoffLimit: 1
    ---
apiVersion: v1
data:
  db2-setup.sh: |
    #!/bin/sh
    export SETUPDIR=/var/db2_setup
    source ${SETUPDIR?}/include/db2_constants
    source ${SETUPDIR?}/include/db2_common_functions

    if ! getent passwd ${DB2INSTANCE?} > /dev/null 2>&1; then
      echo "(*) Previous setup has not been detected. \
      Creating the users... "
      create_users

```

```

fi
if ! create_instance; then
    exit 1
fi
start_db2
cp /scripts/db2-createschema.sh /database/db2- \
createschema.sh
chmod +x /database/db2-createschema.sh
su - $DB2INSTANCE-c "/database/db2-createschema.sh \"$DB2_
SERVICE_NAME\"
\"$DB2INSTANCE\" \"$DB2INST1_PASSWORD\" \"$DBNAME\""
db2-createschema.sh: |
    #!/bin/sh
    DB2_SERVICE_NAME=$1
    DB2INSTANCE=$2
    DB2INST1_PASSWORD=$3
    DBNAME=$4
    echo "Configure schema for database \"$DBNAME\" on host
    \"$DB2_SERVICE_NAME\"."
    db2 "catalog tcpip node TRADERDB remote $DB2_SERVICE_NAME
    server 50000"
    db2 "catalog db $DBNAME as $DBNAME at node TRADERDB"
    db2 terminate
    db2 "activate database $DBNAME"
    db2 "connect to $DBNAME user $DB2INSTANCE using $DB2INST1_
    PASSWORD"
    sleep 2
    db2 -tvvf /scripts/stock-trader.sql
    echo "Database $DBNAME has been configured."
stock-trader.sql: |
    CREATE TABLE Portfolio(owner VARCHAR(32) NOT NULL, total
    DOUBLE, loyalty
    VARCHAR(16), balance DOUBLE, commissions DOUBLE, free INTEGER,
    sentiment
    VARCHAR(16), PRIMARY KEY(owner));
    CREATE TABLE Stock(owner VARCHAR(32) NOT NULL, symbol
    VARCHAR(8) NOT NULL,
    shares INTEGER, price DOUBLE, total DOUBLE, dateQuoted DATE,
    commission DOUBLE,
    FOREIGN KEY (owner) REFERENCES Portfolio(owner) ON DELETE
    CASCADE, PRIMARY
    KEY(owner, symbol));
kind: ConfigMap
metadata:
  name: db2-createschema

```

In the Job resource, we have declared several important elements:

image

We use the same image running the StatefulSet in the role of a client instead of a server. The Job is assumed to run in the same

Namespace as the database and therefore must have access to the same dockerhub Image Pull Secret that we created earlier.

command

We want to override the default entry point of the image. Instead of creating a new database server, we want to use the same binaries and connect to our database running as a StatefulSet. The command references files from a volumeMount.

volumeMount

We load the scripts into the container filesystem by way of a volumeMount, which references a ConfigMap.

env

Our logic reuses some information where it can, such as the password. Additional parameter values must match your environment, including the instance name and database name.

volumes

We reference a ConfigMap as a Volume in this example. All keys which are known to the ConfigMap will appear as files to the container, using the name of the key to define the name of the file. We also set the defaultMode, to ensure that our initial script will be executable.

configMap

The configMap defines our various files to be used in the initialization. If you are creating a ConfigMap from a file, you can simplify the operation by way of the kubectl create command: `kubectl create configmap db2-createschema --from-file db2-createschema.sh`.

db2-setup.sh

The initial entry script which performs some initial configuration of the container and prepares the next script to run as the db2inst1 user.

db2-createschema.sh

This connects to the remote database, catalogs it for remote connections, and triggers the execution of SQL against the database.

`stock-trader.sql`

The SQL commands that define the tables used by our *portfolio* microservice

Like all resources, you can run this Job by using the `kubectl apply` command:

```
$ kubectl apply -f stock-trader-job.yaml
```

Alternatively, you could have enhanced our *portfolio* service to be smart enough to detect when its intended schema doesn't exist and create it automatically.

Our database service is now ready to serve requests from the *portfolio* microservice. To summarize our progress so far, we've focused on several important concepts in this workflow.

- We created a Namespace to isolate our database containers from the rest of the cluster.
- We created a specialized PodSecurityPolicy to allow our database to take advantage of a small set of Linux kernel capabilities.
- We created a ServiceAccount to represent the identity of Pods deployed in this Namespace. Because we have only a single ServiceAccount (default) in the Namespace (`stock-trader-data`), all Pods will be assigned to the same ServiceAccount.
- We used ClusterRole and ClusterRoleBindings to ensure that the ServiceAccount was allowed to use our PodSecurityPolicy.
- We created an Image Pull Secret and associated it with the ServiceAccount to use for any Pods which were deployed in the Namespace and assigned to the ServiceAccount.

After preparing the Namespace and supporting security objects, we deployed our Db2 database container from a Helm chart. The Helm chart provided a collection of resources needed to run the container, along with parameterized options that allowed us to customize the settings from the command line.

Finally, we injected the application schema into the database using a Job and supporting ConfigMaps to make it ready for our *portfolio* microservice.

Next, we deploy our *portfolio* microservice, which uses the database that we just deployed.

Managing Our Portfolio Java-Based Microservice as a Deployment

To deploy the *portfolio* microservice, we begin by cloning the portfolio app from GitHub:

```
$ git clone https://github.com/kubernetes-in-the-enterprise/
portfolio.git
```

Let's take a look at our Deployment manifest. As described earlier, a Deployment captures our application and its configuration into one neat package.

The Deployment references our image, the configuration parameters needed by the image to run correctly, and the ports that are exposed. Unlike Db2, there are no mounted volumes for this Pod, because it is completely stateless:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: portfolio
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: portfolio
        solution: stock-trader
    spec:
      containers:
        - name: portfolio
          image: mycluster.icp:8500/stock-trader/portfolio:latest
          env:
            - name: JDBC_HOST
              valueFrom:
                secretKeyRef:
                  name: db2
                  key: host
            - name: JDBC_PORT
              valueFrom:
                secretKeyRef:
                  name: db2
                  key: port
            - name: JDBC_DB
              valueFrom:
                secretKeyRef:
                  name: db2
                  key: db
```

```

- name: JDBC_ID
  valueFrom:
    secretKeyRef:
      name: db2
      key: id
- name: JDBC_PASSWORD
  valueFrom:
    secretKeyRef:
      name: db2
      key: pwd
# ... a whole bunch of other parameters
- name: JWT_AUDIENCE
  valueFrom:
    secretKeyRef:
      name: jwt
      key: audience
- name: JWT_ISSUER
  valueFrom:
    secretKeyRef:
      name: jwt
      key: issuer
ports:
  - containerPort: 9080
  - containerPort: 9443
imagePullPolicy: Always

```

To successfully deploy the *portfolio* microservice, we need to accomplish the following steps:

1. Create the required Secrets. Many Secrets that are used (MQ, ODM, Watson) are optional, which means the container can start without them.
2. Build the image and push it to an image registry and create the Image Pull Secret, if necessary, to deploy the *portfolio* microservice into the cluster.
3. Create the Deployment via `kubectl`.

Creating the Required Secrets

Most of our parameters are conveyed by Secrets, which provide information about dependencies. The last options, `JWT_AUDIENCE` and `JWT_ISSUER`, provide agreed-upon Secrets between the *portfolio* and *stock-quote* microservices. Let's go ahead and create the `jwt` Secret and then move on to the others:

```
$ kubectl create secret generic jwt -n stock-trader \  
  --from-literal=audience=stock-trader \  
  --from-literal=issuer=http://stock-trader.ibm.com \  
secret "jwt" created
```

Next, we create the Secret for the database. Here, we use values that we used originally to create our Helm release for Db2. If you used different values, update the following command to meet your selections:

```
$ kubectl create secret generic db2 \  
  --namespace=stock-trader \  
  --from-literal=id=db2inst1 \  
  --from-literal=pwd=ThisIsMyPassword \  
  --from-literal=host=stocktrade-ibm-db2oltp-dev \  
  --from-literal=port=50000 \  
  --from-literal=db=STRADER \  
secret "db2" created
```

Build the Image and Push to an Image Registry

With the Secrets created, we are now ready to build the Java web application and the Docker image:

```
cd portfolio \  
mvn package \  
docker build -t portfolio:latest .
```

NOTE

To build the package, there are three prerequisites: Java, Maven, and Docker. To read about how to configure your development environment, see [Appendix B](#).

Now that we've built the image, we need to publish it to a location where the cluster can access it. Just as Db2 is made available as an image in a registry, we're going to do the same for our *portfolio* microservice.

Docker image tags are named references. We can create multiple names for an image. The image name can also refer to a registry by including the hostname for the registry in the name.

For the sake of example, we tag and push the image into an IBM Cloud Container Registry and also into our own local cluster registry. Both of these steps are described in the following sections.

Pushing to a Built-In IBM Cloud Private Cluster Image Registry

To push the image to a built-in IBM Cloud Private cluster registry, perform the following:

```
$ docker login mycluster.icp:8500
Username (admin):
Password:
Login Succeeded

$ docker tag portfolio:latest \
  mycluster.icp:8500/stock-trader/portfolio:latest
$ docker push mycluster.icp:8500/stock-trader/portfolio:latest
```

When working with any remote registry that is outside of the cluster, be sure to create the appropriate Image Pull Secret. Within IBM Cloud Private, Namespaces automatically have access to images pushed into the image repository for that Namespace (e.g., images in the `mycluster.icp:8500/stock-trader` repository will automatically be available to be pulled by Pods in the `stock-trader` Namespace).

Pushing to an IBM Cloud Container Registry

For an IBM Cloud Container Registry, it's the same process but with a different name:

```
$ ibmcloud cr login

Logging in to 'registry.ng.bluemix.net'...
Logged in to 'registry.ng.bluemix.net'.

OK

$ docker tag portfolio:latest \
  registry.ng.bluemix.net/mdelder/portfolio:latest

$ docker push registry.ng.bluemix.net/mdelder/portfolio:latest
```

If you are using a local Kubernetes cluster with a remote IBM Cloud Container Registry, you can create a token to manage your images. Using a token helps avoid the need to store your own credentials as a Secret. In addition, using a token has the benefit of creating read-only tokens, which, if ever compromised, will not permit anyone to tamper with images in your registry:

```

$ ibmcloud cr token-add --description \
"Image Pull Token for local Kubernetes cluster"

Requesting a registry token...
Token identifier 3619bb24-9a5d-5976-
9cb8-2e9ca6d700bf
Token [OBSCURED]
OK

$ kubectl create secret docker-registry ibm-cloud-registry \
--docker-username=token \
--docker-password=$(ibmcloud cr token-get -q 3619bb24-9a5d-\
5976-9cb8-2e9ca6d700bf) \
--docker-email=youremail@mailserver.com \
--namespace=stock-trader
secret "ibm-cloud-registry"created

```

Deploying the Manifest for the portfolio Microservice

Now that the image is available to cluster, let's create our Deployment:

```

$ cd portfolio/manifests
$ kubectl apply --namespace=stock-trader -f deploy.yaml
deployment.extensions "portfolio" created
service "portfolio-service" created
ingress.extensions "portfolio-ingress" created

```

You can validate that the Deployment came online correctly by reviewing the logs (you can identify the Pod, which will have a random identifier as a suffix, directly by its labels):

```

$ kubectl logs --namespace=stock-trader \
--selector="app=portfolio,solution=stock-trader"

...
[INFO    ] SRVE0169I: Loading Web Module: Portfolio.
[INFO    ] SRVE0250I: Web Module Portfolio has been bound to
default_host.
[INFO    ] SESN0172I: The session manager is using the Java
default SecureRandom
implementation for session ID generation.
[INFO    ] SESN0176I: A new session context will be created
for application key
default_host/portfolio
[AUDIT   ] CWWKT0016I: Web application available
(default_host): http://portfolio-
6b98585ff6-cx8sn:9080/portfolio/
...

```

Or you can simply just check the status of the Deployment by using the `kubectl get deployments` command:

```
$ kubectl get deployments \
  --namespace=stock-trader \
  --selector="app=portfolio,solution=stock-trader" \
  -o wide
```

NAME	DESIRED	CURRENT	UP-TO-DATE	AVAILABLE	AGE
CONTAINERS					
IMAGES					SELECTOR
portfolio	1	1	1	1	6h
portfolio					
mycluster.icp:8500/stock-trader/portfolio:latest					\
app=portfolio,					
solution=stock-trader					

Deploying the trader Microservice Web Frontend

Let's now deploy a web frontend for our *portfolio* microservice that will allow us to validate the basic behavior.

For this example, we will not build the image; instead, we simply apply a Deployment manifest that references an image that has been prebuilt on DockerHub.

The *deploy.yaml* file for *trader* includes resources for Deployment, Ingress, and Service. We have already discussed Deployments, but let's take a closer look at Ingress and Service:

```
$ git clone \
  https://github.com/kubernetes-in-the-enterprise/trader.git
$ kubectl apply --namespace=stock-trader \
  -f trader/manifests/deploy.yaml
```

Now we can visit the web frontend using two entry points:

Ingress rule

A web context root configured on the cluster, backed by a Service. Kubernetes is doing a lot of heavy lifting for us to enable this flow. An Ingress rule can specify annotations to help you configure TLS certificates or customize behaviors like session affinity. Behind the scenes, a Kubernetes Service provides an in-cluster load balancer in front of the Pods that expose services on networking ports. The result is a powerful abstraction that is easy to describe in YAML:

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: trader-ingress
  annotations:
    kubernetes.io/ingress.class: "nginx"
    ingress.kubernetes.io/affinity: "cookie"
    ingress.kubernetes.io/session-cookie-name: "route"
    ingress.kubernetes.io/session-cookie-hash: "sha1"
    ingress.kubernetes.io/secure-backends: "true"
    ingress.kubernetes.io/app-root: "/trader"
spec:
  rules:
  - http:
      paths:
      - path: /trader
        backend:
          serviceName: trader-service
          servicePort: 9443

```

Your ingress route will be available at [https://\[master-ip\]/trader](https://[master-ip]/trader) on IBM Cloud Private.

Here's what's happening in this code:

annotations

These are specific behaviors that the Ingress Controller uses when defining the various configurations in NGINX for this specific web context (“/trader”). Many annotations are available on [GitHub](#).

rules

The rules define a set of routes, the appropriate protocol (HTTP/HTTPS), and the Service name that satisfies the incoming request.

serviceName

The name of a Service available within the Namespace which will react when incoming requests are received.

Service

Recall that Services were discussed earlier and that clusters support different kinds of Services. The trader Service is exposed as a NodePort, meaning that it can be accessed from outside the cluster. This is shown in the following YAML specification:

```

apiVersion: v1
kind: Service

```



```
metadata:
  name: trader-service
  labels:
    app: trader
spec:
  type: NodePort
  ports:
    - name: http
      protocol: TCP
      port: 9080
      targetPort: 9080
      nodePort: 32388
    - name: https
      protocol: TCP
      port: 9443
      targetPort: 9443
      nodePort: 32389
  selector:
    app: trader
```

Your NodePort Service will be available at *https://[master-ip]:32389/trader*.

type

There are various kinds of Services, including ClusterIP (all traffic exposed only within the cluster), NodePort (an external port is assigned to the Pod enabling incoming requests), and LoadBalancer (an external load balancer is updated with the relevant configuration to access the Pod).

ports

The ports specify a mapping between the port used by consumers (`targetPort`) and the port exposed within the container (`port`). The `nodePort` is a requested port to be enabled on the Host for external requests.

Deploying a Containerized MQ Series Manager as a StatefulSet

MQ Series handles distributed messaging at scale for many enterprises. Here, we'll stand up MQ as a Kubernetes StatefulSet. Just like Db2, a PersistentVolume will store information for MQ topics and queues, ensuring that if a failure occurs, our messaging service will recover. We use Helm to install MQ Series by performing the following:

```

$ helm install --name appmsg ibm-charts/ibm-mqadvanced-server\
-dev --tls
--set license=accept \
--set persistence.enabled=true \
--set persistence.useDynamicProvisioning=true \
--set dataPVC.storageClassName=glusterfs \
--set queueManager.name=STQMGR \
--set queueManager.dev.adminPassword=ThisIsMyPassword \
--set queueManager.dev.appPassword=ThisIsMyPassword \
--set nameOverride=stmq

```

NOTE

Consider using `ibmc-block-gold` storage class if deploying on IBM Cloud Kubernetes Service

Next, we define our initial messaging queue, which serves messages about changes in a user's loyalty program status:

```

$ kubectl exec -it appmsg-stmq-0 /bin/bash
$ runmqsc
DEFINE QLOCAL (NotificationQ)
SET AUTHREC PROFILE('NotificationQ') OBJTYPE(QQUEUE) PRINCIPAL(
'app')
AUTHADD(PUT,GET,INQ,BROWSE)
end

```

After we have customized our messaging service, we create a Secret for our microservices to configure connections. Notice the references for password (`pwd`) and queue-manager (`STQMGR`) that must match the values specified in the preceding example during the installation of the Helm chart:

```

$ kubectl create secret generic mq \
--from-literal=id=app \
--from-literal=pwd=ThisIsMyPassword \
--from-literal=host=appmsg-stmq \
--from-literal=port=1414 \
--from-literal=channel=DEV.APP.SVRCONN \
--from-literal=queue-manager=STQMGR \
--from-literal=queue=NotificationQ

```

Deploying Supporting Services for the portfolio Microservice

Our *portfolio* microservice depends on two backend services:

stock-quote

Returns the current value in US dollars (USD) of a stock symbol.

loyalty

Returns the level of loyalty program status achieved by the user based on their total portfolio value under management.

For each of these services, we are going to deploy them using the publicly available image on DockerHub.

Deploying the stock-quote Microservice

Begin by cloning the *stock-quote* Git repository. We will bypass the build and push of the Docker image and instead deploy an image from the public DockerHub registry. Of course, you can always build the image locally and push it to your own private registry with the same outcome. We clone *stock-quote* by using the following Git clone command:

```
$ git clone \
https://github.com/kubernetes-in-the-enterprise/stock-quote.git
```

The *manifests/deploy.yaml* resource describes the Kubernetes Deployment for stock-quote:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: stock-quote
  # namespace: stock-trader
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: stock-quote
        solution: stock-trader
    spec:
      containers:
        - name: stock-quote
          image: ibmstocktrader/stock-quote:latest # DockerHub
          env:
            - name: REDIS_URL
              valueFrom:
                secretKeyRef:
                  name: redis
                  key: url
                  optional: true
```

```

- name: JWT_AUDIENCE
  valueFrom:
    secretKeyRef:
      name: jwt
      key: audience
- name: JWT_ISSUER
  valueFrom:
    secretKeyRef:
      name: jwt
      key: issuer
ports:
- containerPort: 9080
- containerPort: 9443
imagePullPolicy: Always

```

Like *portfolio*, the *stock-quote* Service also exposes two options for incoming network traffic. The Service exposes traffic for internal and external consumers, as shown here:

```

apiVersion: v1
kind: Service
metadata:
  name: stock-quote-service
  # namespace: stock-trader
  labels:
    app: stock-quote
spec:
  type: NodePort
  ports:
    - name: http
      protocol: TCP
      port: 9080
      targetPort: 9080
    - name: https
      protocol: TCP
      port: 9443
      targetPort: 9443
  selector:
    app: stock-quote

```

The Ingress provides a route on the cluster ingress management controller. The following is the YAML specification for the Ingress:

```

apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: stock-quote-ingress
spec:
  rules:
    - http:
        paths:

```

```
- path: /stock-quote/.*
  backend:
    serviceName: stock-quote-service
    servicePort: 9080
```

We deploy *stock-quote* as we did with *portfolio*, using the following `kubectl` command:

```
$ cd stock-quote/manifests
$ kubectl apply -f deploy.yaml --namespace=stock-trader
deployment.extensions "stock-quote" created
service "stock-quote-service" created
ingress.extensions "stock-quote-ingress" created
```

As we see from the command output, we have created several Kubernetes resources from a single file. Be sure to deploy the *stock-quote* microservice in the `stock-trader` Namespace, or else *portfolio* would be unable to find it by the name `stock-quote-service`.

Deploying the *loyalty* Microservice

Deploying the *loyalty* microservice follows the same architectural patterns and commands that we used for *stock-quote*:

```
$ git clone https://github.com/kubernetes-in-the-enterprise/
loyalty-level.git
$ cd loyalty-level/manifests
$ kubectl apply -f deploy.yaml --namespace=stock-trader
deployment.extensions "loyalty-level" created
service "loyalty-level-service" created
ingress.extensions "loyalty-level-ingress" created
```

Just as before, we have created several Kubernetes resources, including a Deployment that manages the Pods, and two options for incoming network traffic.

Putting It All together: Accessing Our Fully Configured Application

We have now deployed all of our components for the application, which you can see listed in [Table 4-1](#).

Table 4-1. Microservices and supporting middleware for the Portfolio application

Name	Namespace	Kubernetes resources	Description
<i>portfolio</i>	stock-trader	<ol style="list-style-type: none"> 1. Deployment 2. Service 3. Ingress Rule 	A Java-based microservice that provides the heart of our business logic for creating, updating, and removing stock portfolios
db2	stock-trader	Secret	Provides credentials for access to the Db2 database
mq	stock-trader	Secret	Provides credentials for our microservices to send and receive messages via our IBM MQ container
jwt	stock-trader	Secret	Provides credentials which are shared amongst our Java microservices for generating and sharing JSON Web Token (JWT) authorization tokens
dockerhub	stock-trader-data	Secret (Image Pull Secret)	An Image Pull Secret that allows our cluster to access images on DockerHub for which we have valid subscriptions
stocktrade-ibm-db2oltp-dev	stock-trader-data	<ol style="list-style-type: none"> 1. StatefulSet 2. Service 3. Secret 	A Helm chart-based deployment of our enterprise database
stocktrade-ibm-db2oltp-dev	stock-trader	Service	An ExternalName Service that enables our <i>portfolio</i> microservice in the stock-trader Namespace to find our Db2 database in the stock-quote-data Namespace.
<i>trader</i>	stock-trader	<ol style="list-style-type: none"> 1. Deployment 2. Service 3. Ingress rule 	A Java-based microservice that provides the web frontend for users to access our application.
appmsg-stmq	stock-trader	<ol style="list-style-type: none"> 1. StatefulSet 2. Service 3. Secret 	A Helm chart-based deployment of IBM MQ that is used by our microservices for message-driven API and notifications

Name	Namespace	Kubernetes resources	Description
<i>stock-quote</i>	stock-trader	<ol style="list-style-type: none"> 1. Deployment 2. Service 3. Ingress rule 	A Java-based microservice that provides a stock symbol lookup service
<i>loyalty-level</i>	stock-trader	<ol style="list-style-type: none"> 1. Deployment 2. Service 3. Ingress rule 	A Java-based microservice that provides the loyalty program level for a given user based on the total value of the user's portfolio

When we deployed the *trader* microservice, we exposed an Ingress and a Service of type NodePort. Let's open the web UI from one of these endpoints. For convenience, here are the endpoints that are exposed by default:

- Your ingress route will be available at *https://[master-ip]/trader* on IBM Cloud Private. If you defined a custom domain, you can use the domain name instead of the IP. The default hostname will be *mycluster.icp*.
- Your NodePort Service will be available at *https://[master-ip]:32388/trader*.

Figure 4-2 depicts the user login page for the IBM Stock Trader application. The default username and password are as follows:

- Username: *stock*
- Password: *trader*

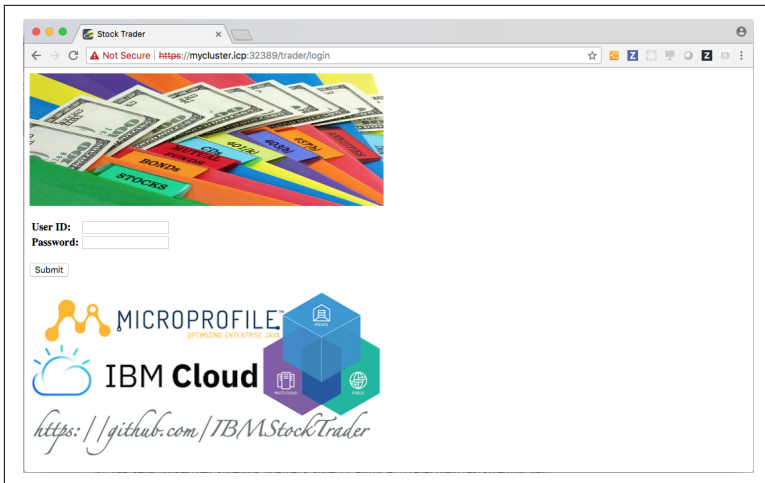


Figure 4-2. The IBM Stock Trader application login page.

With your new app, you can create, update, and delete portfolios. As you interact with the *trader* web frontend, API calls are made to *portfolio*, which, in turn, interacts with its Db2 database StatefulSet, MQ messaging StatefulSet, and the *loyalty* and *stock-quote* microservices. Three example portfolios have been created in [Figure 4-3](#). As each of these portfolios is created, API calls are made from *trader* to *portfolio*. In turn, *portfolio* updates the database to save the updates. The total value of each portfolio is calculated based on results from the *stock-quote* service via API calls as well.

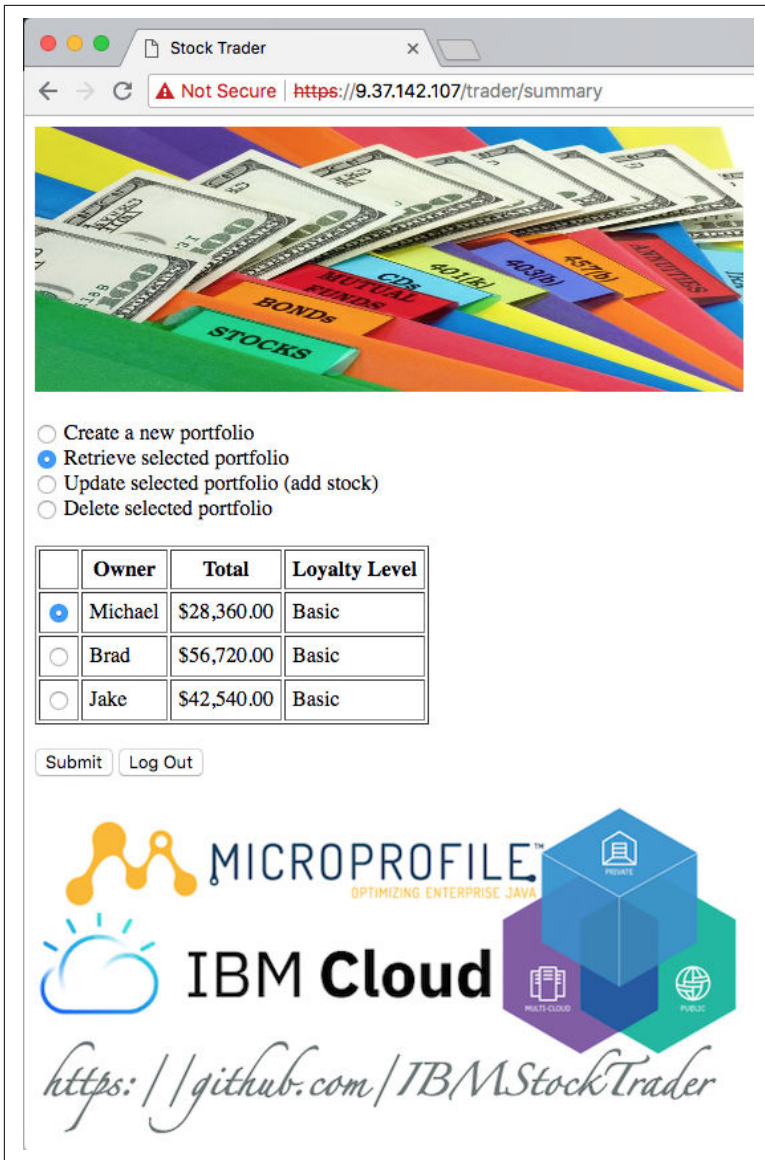


Figure 4-3. The IBM Stock Trader application portfolio summary page.

Summary

In this chapter, we deployed a production-quality enterprise Kubernetes application called IBM Stock Trader that is a composition of

several microservices and uses numerous key Kubernetes resources such as Deployments, StatefulSets, Services, and Secrets. In [Chapter 5](#), we explore how to take advantage of Continuous Delivery approaches for your Kubernetes applications.

Continuous Delivery

Entire books have been written about Continuous Delivery and DevOps in general. We are not going to repeat why Continuous Delivery is important. With that limited scope in mind, let's review how container images and Kubernetes support the following DevOps principles:

Small batch changes

All changes should be incremental and finite. When failures occur, small batch changes are typically easier to recover than large disruptive changes.

Source control all the things

A history of all changes is helpful to understand what changes have been made and to identify the cause of regressions in the code base or configuration.

Production-like environments

Developers should have access to environments and tools that are representative of production. Production environments typically operate at larger scales than development or quality assurance (QA) and with more complex configuration. The variance can mean that features that work fine in early stages do not work correctly in production—which is the only place it matters.

Shift-left of operational practices

We should expose behaviors for health management, log collection, change management, and so on earlier in the development process.

Continuous integration of changes

All changes should be built and deployed together on an ongoing basis to identify when the intermingling of various changes leads to an unforeseen issue or application programming interface (API) incompatibility.

Highly automated testing with continuous feedback

To manage velocity, you need to automate your testing and validation work so that you can always be testing (ABT).

Image Build

Containers are the ideal unit of delivery because they encapsulate all aspects of your application, middleware, and operating system (OS) packages into a single package. You can create container images in several ways, and the most popular means of creating an image is by a Dockerfile. The Dockerfile describes all of the relevant details necessary to package the image. These instructions are just plain text until converted by the image build, and you should always manage these and other declarative resources we have learned about in Kubernetes in a source control repository such as Git.

In [Chapter 4](#), we built a container image. The act of building used our Dockerfile as a set of instructions to construct the image. Let's take a closer look inside the Dockerfile:

```
FROM openliberty/open-liberty:microProfile1

RUN groupadd -g 999 adminusr && \
    useradd -r -u 999 -g adminusr adminusr
RUN chown adminusr:adminusr -R /opt/ol /logs /config
USER 999

COPY --chown=adminusr db2jcc4.jar /config/db2jcc4.jar
ADD --chown=adminusr \
    http://repo1.maven.org/maven2/com/ibm/mq/wmq.jmsra/9.1.0.0/wmq.
    jmsra-9.1.0.0.rar /config/wmq.jmsra.rar
COPY --chown=adminusr key.jks /config/resources/security/key.jks
COPY --chown=adminusr keystore.xml /config/configDropins/
    defaults/keystore.xml
COPY --chown=adminusr server.xml /config/server.xml
```

```
COPY --chown=adminusr target/portfolio-1.0-SNAPSHOT.war /config/
apps/Portfolio.war
```

FROM *statements*

They declare the foundation for your container. Generally, base images are application runtimes (e.g., `openliberty/openliberty:microProfile1` or `node:latest`) or operating systems (e.g., `ubuntu:16.04`, `rhel:7`, or `alpine:latest`).

RUN *statements*

They execute commands and save the resulting changes to the filesystem as a distinct layer in the container image. To optimize build time, move commands that need to adjust the container image more frequently (e.g., adding application binaries) toward the end of the file.

USER *statements*

They allow you to specify under what identity on the OS the container should be launched. Building your container to run as `nonroot` is considered best practice. Whenever specifying a user for Kubernetes, we recommend that you use the UID (numerical user ID from the OS) and create an alias via `groupadd/useradd` commands or equivalents for your OS.

COPY and ADD *statements*

They move files from your local workspace into the container file system. `COPY` should always be your default; `ADD` is useful when pulling directly from URLs, which is not supported by the `COPY` directive.

Each line in the Dockerfile creates a unique layer that is reused for subsequent builds where no changes have occurred. Reusing layers creates a very efficient process that you can use for continuous integration builds.

As you saw in [Chapter 4](#), you build container images as follows:

```
$ docker build -t repository:tag .
```

The last period is significant because it denotes the current directory. All files in the directory are shipped off to the Docker runtime to build the image layers. You can limit files from being delivered by adding a `.dockerignore` file. The `.dockerignore` file specifies files by name pattern to either be excluded (the default) or included (by beginning the line with an exclamation mark `!`).

Programmability of Kubernetes

The declarative model of Kubernetes objects makes them ideal for source control, meaning that you have a record (throughout the history of your source control system) of all changes that were made and by whom.

Using the command `kubectl apply`, you can push updates easily into your cluster. Some resource types do not easily support rolling or uninterrupted updates (like DaemonSets), but a majority do. You don't need to stop at just one object, though; you can apply entire directories:

```
$ kubectl apply -Rf manifests/
```

In this snippet, we use the `-R` option, which indicates to recursively process all files in the `manifests/` directory.

Kubernetes makes it easy to create consistent clusters in development, QA, and production. Consistency means that developers and testers can develop and test against production-like environments in earlier stages. In addition, Helm charts make it possible to also deploy supporting Services like databases, messaging, and caching much easier and more accessible to developers and testers.

General Flow of Changes

The only constant is change. Many tools exist to help you on your **Continuous Delivery** journey. The general flow for all of these will be as follows:

1. Register a Git post-commit hook to trigger a build for all commits delivered into the source control repository.
2. The build will prepare the container image associated with the Git repository and publish it to an image repository.
3. The build service will create a Kubernetes cluster, create a Kubernetes Namespace within a cluster, or reuse an existing cluster or Namespace based on convention to deploy the Kubernetes objects described in the manifest.
4. Optionally, the build can package a Helm chart and push it into a Helm repository and deploy it from there, with references to the images published earlier.

5. Run automated tests against the deployed system to verify the changes have not created any regressions in the source code or supporting configuration files.
6. Optionally, drive an automated rolling continuous update to the next stage in the pipeline to validate the change for release to production.

For more detailed instruction on using automation tools to facilitate Continuous Delivery, the [IBM Garage Method](#) website includes courses such as “[Use Jenkins in a Kubernetes cluster to continuously integrate and deliver on IBM Cloud Private](#)” that walk you through the aforementioned steps by providing an easy-to-follow tutorial. After you become comfortable with Continuous Delivery, the next step is to focus on operating your enterprise application. In [Chapter 6](#), we provide an overview of several tools that reduce the complexity of operating enterprise applications in a Kubernetes cluster.

Enterprise Application Operations

In this chapter, we provide an overview of several key tools that are critical to operating your enterprise application. We begin with a discussion of *Logstash* and *Fluentd*, which are two popular enterprise-quality tools for performing a centralized log collection for your microservices running distributed across a cluster. We then describe *Elasticsearch*, which is a log storage repository with advanced knowledge discovery and analysis features. Then, we introduce *Kibana*, which is a dashboard-based log visualization tool. We complete our discussion of log collection tools with an overview of the log collection support provided by both the IBM Cloud Kubernetes Service and IBM Cloud Private. In the second half of this chapter, we describe the health-management facilities for monitoring enterprise applications. We conclude this chapter with a description of monitoring capabilities provided by both IBM Cloud Kubernetes Service and IBM Cloud Private.

Log Collection and Analysis for Your Microservices

Logs are the universal debugger. When all else fails, check the logs and hope the developer was paying attention to the serviceability of their code.

In all seriousness, logging is a critical component for the operation and service of any Kubernetes-based application. Unlike traditional monolithic applications, developers and operators desire a central-

ized logging solution that will aggregate logs from their distributed applications to prevent the need to chase down dozens or hundreds of Pods using `kubectl logs -f --tail=1000 mypodname`.

With microservices, you break down your core functions into smaller building blocks, which means that your logged information is going to be more decentralized. There are several common patterns for collecting logs from containerized platforms.

First, you need a way to collect log information from your containers. Let's take a look at a couple of tools that can help you with this:

Logstash

Logstash is a very well-established log aggregator with the capabilities to collect logs, parse, and then transform them to meet your standards.

Fluentd

Fluentd is an open source data collector that provides an independent and unifying layer between different types of log inputs and outputs. Fluentd was recently contributed to the Cloud Native Computing Foundation (CNCF). Although Fluentd is younger than Logstash, its alignment with the CNCF has driven its increasing popularity.

Both of these options provide a robust way to collect logs from multiple sources—from containers to nodes—and then ship them into a searchable datastore.

After your logs are aggregated, you need a place to store them for discovery and analysis. Elasticsearch is the de facto standard for collecting, indexing, and providing an ongoing store for plain-text or JSON log output. Elasticsearch provides several key features that make it a great option for your log data:

JSON-based query language, accessible via REST API

You can consume information from Elasticsearch easily for discovery and analysis. Often, Elasticsearch ends up being part of machine learning applications because of its simple but powerful model of querying and updates.

Log retention policy control

You might need to keep only a week of logs for postmortems when bad things happen. In other cases, compliance or audit requirements might require you to keep logs for months or

years. Elasticsearch will automatically age out data that is no longer relevant for your needs.

Move logs to long-term storage

With Elasticsearch, you can keep a portion of log data where it's easily accessible and searchable, but then move older logs into a longer-term store.

Finally, now that you have collected the logs and stored them, it's time to visualize them. **Kibana** provides a flexible way to visualize your logs with a variety of widgets from tables to graphs. Kibana connects to Elasticsearch and creates Indexer patterns that provide access to data efficiently. With the Kibana dashboard, you can do the following:

Discover the data available in the system

From the Discover view in the Kibana dashboard, you have access to all of the information in the system for which you have authorization to view. You can define table views for log information and filter based on any log attributes. For example, you might create a filter for `kubernetes.namespace: "stock-trader"` to see all of the log output for all Pods in the stock-trader Namespace.

Create reusable widgets to visualize specific aspects

Perhaps you always want to view a table with specific log attributes or you want a line graph of how many logs contain the word "error" in the stock-trader Namespace. Each visualization is a reusable widget that you can use across one or more dashboards.

Create dashboards

Dashboards allow you to collect visualizations into a single view.

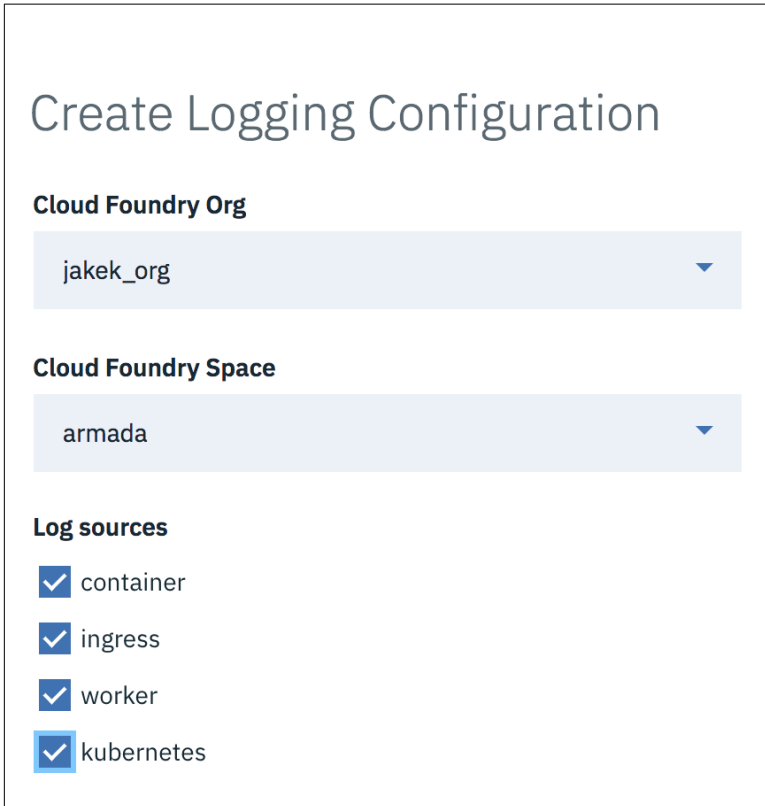
Create timeline analysis

You can also use Kibana to do time-series analysis from multiple data sources.

IBM Cloud Kubernetes Service Log Analysis Support

IBM Cloud Kubernetes Service uses Fluentd and IBM Cloud Log Analysis service for collecting and analyzing logs. On this platform, there are many logging configuration options outlined in the **documentation**. Here, you will find a variety of command-line options

that you can use to configure very detailed filtering mechanisms. Common options available include filtering to display only error logs and sending logs from different Kubernetes Namespaces to different logging tenants. The simplest process for getting started is through the IBM Cloud Kubernetes Service user interface. Simply choose “enable logging” from the cluster overview page and then select the log sources and destination to get started with a basic logging configuration, as depicted in [Figure 6-1](#).



Create Logging Configuration

Cloud Foundry Org

jakek_org

Cloud Foundry Space

armada

Log sources

- container
- ingress
- worker
- kubernetes

Figure 6-1. Configuring IBM Cloud Kubernetes Service logging.

In addition to container, ingress, worker node, and Kubernetes infrastructure logs, there are other options for logging Kubernetes events via an event-exporter, [IBM Cloud Kubernetes Service Activity Tracker events](#), and [Kubernetes audit logs](#). [Figure 6-2](#) provides an example of IBM Cloud Kubernetes service log data for audit logs.

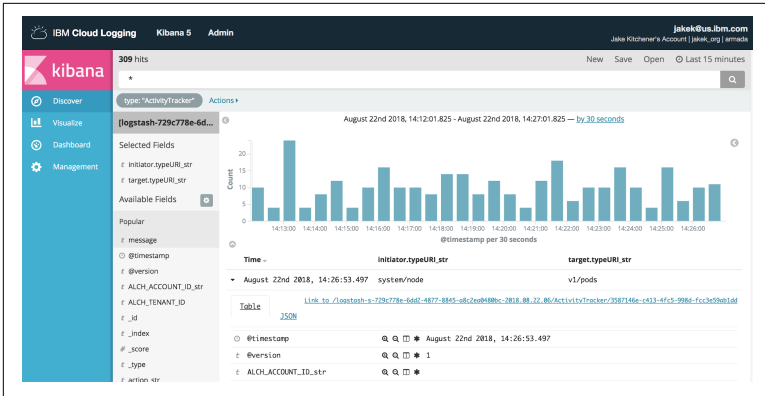


Figure 6-2. Viewing IBM Cloud Log Analysis data for Kubernetes audit log.

IBM Cloud Private Log Analysis Support

IBM Cloud Private uses Elasticsearch-Logstash-Kibana (ELK) to collect log information. You can customize whether these components are installed in the `config.yaml` file for the installer or add them post-installation from the Catalog.

If you configure these components during installation or update the cluster using the installer’s add-on action, you will see references for the Kibana dashboard exposed in the navigation bar on the left, as illustrated in Figure 6-3.

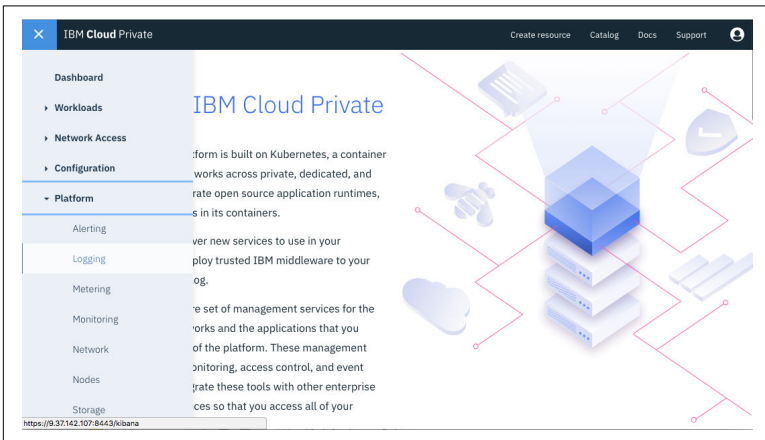


Figure 6-3. Configuring logging in IBM Cloud Private.

A deep dive of the full capabilities of Kibana is beyond the scope of this book, but to get started, try using the Discover view to filter logs by Kubernetes Namespaces. [Figure 6-4](#) illustrates this view filtering option.

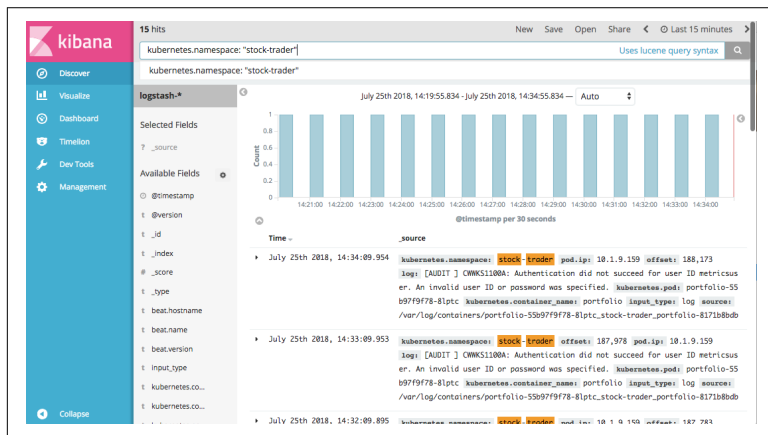


Figure 6-4. Sample display in Kibana of logs filtered by Namespace.

Health Management for Your Microservices

After you've deployed your microservice, Kubernetes takes over much of the basic mechanics of application reliability—but you're not completely off the hook. Managing your application still involves managing its health, investigating issues that arise during an event, performing postmortem analysis, or fulfilling audit compliance.

Fortunately, there are options on how to monitor the health, define alerts, and view an integrated dashboard of logs across all microservices. We will now look at how to consume these management services in one of two modes: deployed on your own cluster or consumed as part of a Software as a Service (SaaS) offering in a public cloud.

Metrics help you to understand the current state of your application. There are many kinds of metrics that provide your application a way to inform observers about its health. You then can use metrics to drive alerts or simply visualize trends on a dashboard.

Prometheus is an open source monitoring and alerting tool. It was contributed to the CNCF in May 2016. Prometheus is one of the

CNCF's more mature projects and was promoted to the **status level of “graduated” in August 2018**. It is the de facto standard for monitoring when working with Kubernetes. Prometheus makes it easy for any web-based app to expose an additional endpoint, which returns metrics for the application using a simple text-based format.

Prometheus will then *scrape* the endpoints to collect these metrics and store them. For non-REST-centric workloads (e.g., a database or messaging app), Prometheus supports exporters, which collect information and make it available in the correct format.

Metrics can be kept in memory for a given window of time or saved to long-term storage. You should consider whether long-term trending information is useful for your needs. In many cases, a window of only several hours is kept around in memory.

There are several kinds of **metrics supported by Prometheus**:

Counter

A counter is an increasing value that allows you to track the total number of times something has happened. Examples might include the number of requests for your application services, or the total number of widgets that your application programming interface (API) has created.

Gauge

A gauge provides an arbitrary reading that can go up or down. Think of a gauge as like a thermometer. The temperature reading can go up or down, and prior readings do not have a direct impact on current readings. You might use gauges to track the number of active transactions the system is processing or the total number of active user sessions engaged with your application.

Histogram

A histogram allows you to capture information that is time-series related. A histogram might declare multiple buckets for grouping observed data. Examples might include the duration of a transaction like a stock trade. Histograms help perform some fundamental heavy lifting for you like calculating the sum of observed values or counting the total number of data points in a given bucket.

Summary

A summary is similar in behavior to a histogram but automatically calculates quantiles over a sliding time window. Measuring the quantiles can be useful to group metrics like response times (*how many responses completed in less than 200 ms? 50%? 95%?*).

Kubernetes enables a rich set of metrics out of the box:

Node Exporter

Captures information about the nodes in your cluster. Metrics such as CPU utilization, disk I/O, network behavior, and filesystem information (nfs, xfs, etc.) are automatically exported.

cAdvisor

Captures information about containers running on the node. Metrics such as container CPU utilization, filesystem I/O, network behavior, and start time are exported.

Heapster

Captures specific metrics about Pods and other Kubernetes resources.

Your microservices can also emit Prometheus metrics to provide application-specific details. For instance, our *portfolio* microservice might emit metrics on the number of trades requested and completed. We can often understand whether the total system is healthy by observing the trends in higher-level metrics such as portfolio trades. Netflix has talked about its use of the *stream-starts-per-second* (SPS) metric to understand whether the end-to-end system is functioning normally; the rate of video streams started is well understood by the team, and variations from the norm tend to indicate that a problem exists, even if the origin of the problem is not with the service itself.¹

Although it's beyond the scope of this book, enabling metrics for your specific application in Kubernetes is a very straightforward task that we strongly recommend.

IBM Cloud Kubernetes Service Monitoring Capabilities

Monitoring capabilities in IBM Cloud Kubernetes Service are handled via IBM Cloud Monitoring. You will find a link to the monitor-

¹ Ranjit Mavinkurve, Justin Becker, and Ben Christensen, “Improving Netflix’s Operational Visibility with Real-Time Insight Tools”, *Medium.com*.

ing dashboard for IBM Cloud Kubernetes Service from your cluster overview page. This brings you to a Grafana dashboard with basic cluster monitoring information, as demonstrated in [Figure 6-5](#).

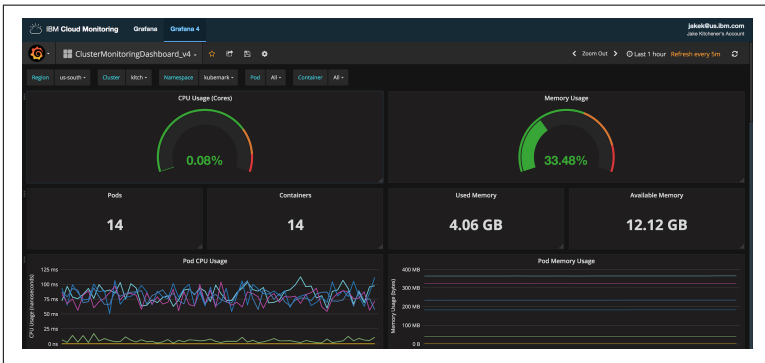


Figure 6-5. Data for IBM Cloud Monitoring is collected via a custom Fluentd plug-in (cAdvisor). No additional configuration or agent is required.

IBM Cloud Private Monitoring Capabilities

IBM Cloud Private includes a built-in Prometheus to collect metrics from the platform and workloads, and a built-in Grafana to visualize those metrics in easy-to-use dashboards.

From the web console, you can launch the Grafana to view, add, or modify dashboards to meet your needs. As shown in [Figure 6-6](#), simply select the Monitoring option from the Platform menu to launch the monitoring dashboards.

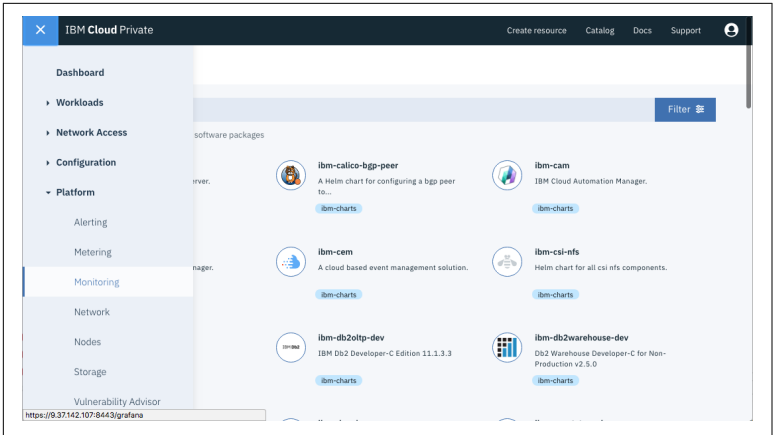


Figure 6-6. Clicking the Monitoring option opens the monitoring dashboards.

Several dashboards are available out of the box. You can always browse community dashboards available for Prometheus and add your own. Figure 6-7 demonstrates available dashboards in IBM Cloud Private out of the box. You can create additional dashboards to fit your needs or import them from the community website and customize them as you need.

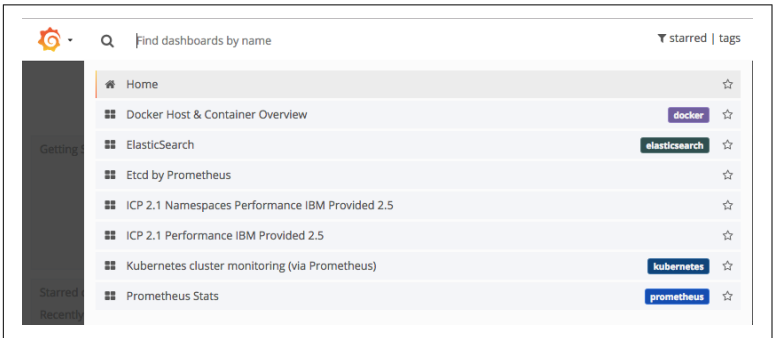


Figure 6-7. Available dashboards out of the box in IBM Cloud Private 2.1.0.3.

Let's highlight two of these dashboards:

Kubernetes cluster monitoring (via Prometheus)

Provides an overview of the entire cluster. As Figure 6-8 illustrates, you can see gauges for cluster memory, CPU, and filesystems.

tem usage, followed by information about the containers running across the cluster.

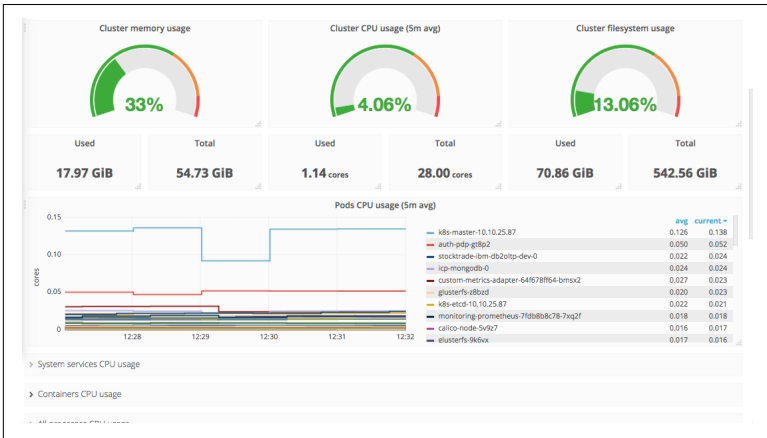


Figure 6-8. Kubernetes cluster monitoring dashboard.

ICP 2.1 Namespaces Performance IBM Provided 2.5

This dashboard, shown in Figure 6-9, provides information for a specific Namespace. It displays information on container CPU, memory, readiness status, and other details. At the top of the dashboard, you can adjust the selected Namespace to view details about another Namespace.

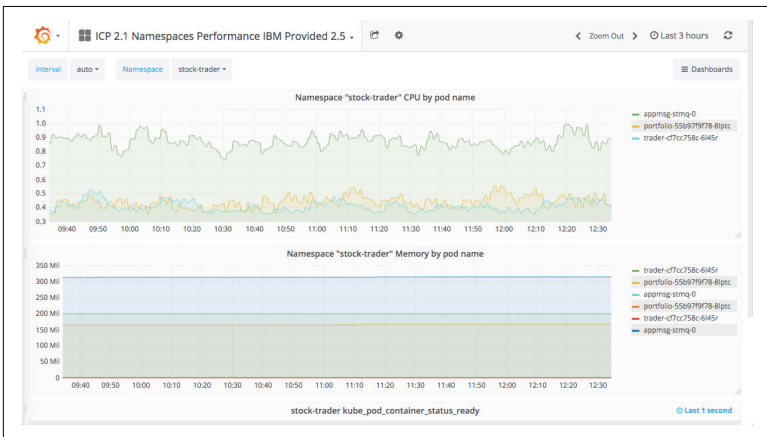


Figure 6-9. Namespace performance dashboard provided by IBM Cloud Private.

Summary

In this chapter, we presented several key tools that are critical to operating your enterprise application. We began with a discussion of popular tool options for the centralized log collection, analysis, and visualization of the data generated by your microservices running across a cluster. We also provided an overview of Prometheus, which is the de facto standard tool for performing monitoring and providing alert information for Kubernetes applications. Also included in this chapter was a discussion of how to utilize these operations-based tools from both IBM Cloud Kubernetes Service and IBM Cloud Private. In [Chapter 7](#), we look at cluster operations issues and key considerations that need to be addressed when running in hybrid cloud environments.

Cluster Operations and Hybrid Cloud

There are several key considerations that cloud operators must examine when administering Kubernetes clusters. In addition, cloud operators encounter even more complexity when they begin to support hybrid cloud environments. In this chapter, we begin with a brief description of hybrid cloud and the common motivations for adopting a Kubernetes-based hybrid cloud environment. We then take a more in-depth look at several key operational aspects of Kubernetes clusters, including access control, performance considerations, networking, storage, logging, and metrics integration. For each of these operational topics, we include a discussion of the issues that you must address when moving to hybrid cloud environments.

Hybrid Cloud Overview

What is hybrid cloud? Typically, hybrid cloud refers to any organization that is using a combined cloud platform that spans on-premises datacenters as well as a public cloud. In this book, we have been looking at a combination of IBM Cloud Kubernetes Service (a public cloud Kubernetes as a Service offering) and IBM Cloud Private (a software solution that can be deployed into on-premises datacenters).

As enterprises make the transition from on-premises datacenters to public cloud, there are many challenges that development and oper-

ations teams will encounter. Although Kubernetes on its own is capable of helping to simplify many of the concerns these teams might have, there are still some significant complexities that are unique to hybrid cloud environments.

A hybrid cloud environment is ideal for companies that are making the transition from on-premises to a public cloud. Many organizations initially begin by using public cloud environments for their development needs, and in this domain compliance and data residency issues are not a factor. Over time, most teams become accustomed to the simplified operations and reduced cost of the public cloud and they begin the journey of migrating other components, such as their stateless applications and auxiliary services, as well. In this chapter, we provide an overview of a variety of cluster operations and hybrid cloud–related issues that are typically encountered in enterprise environments.

Access Control

Access control consists of concepts and features relating to how you authenticate and authorize users for Kubernetes. There are any number of operational and hybrid considerations to account for. The following sections describe operational and hybrid considerations for both authentication and authorization.

Authentication

Users and operators will encounter a wide variety of authentication solutions for Kubernetes. There could be multiple authentication methods in use simultaneously in different portions of a hybrid cloud environment:

Client certificates

X.509 client certificates are Secure Sockets Layer (SSL) certificates. The common name of the certificate is used to identify the user. The organization of the certificate is used to identify group membership. Often, this is used as the “superuser” authentication method.

Static tokens

These are statically defined bearer tokens that are passed in during apiserver startup. In general, they are not commonly used.

Bootstrap tokens

These are dynamically generated bearer tokens that are created by the apiserver via an API. Typically, these are used for bootstrapping clusters, especially joining worker nodes to the cluster.

Static password file

Just what you think. This is a file passed to the apiserver on startup that defines a static list of users, passwords, and groups. We do not recommend this approach.

Service account tokens

These are tokens that are generated automatically by the apiserver for all service accounts that are created. The functionality is built in to the Kubernetes ServiceAccount architecture. Pods running in cluster use this approach to authenticate with the apiserver.

OpenID Connect Tokens (OIDC)

OIDC is a variant of OAuth2 that uses a standalone authentication provider to authenticate users and generate JWT tokens for presentation to Kubernetes for authentication. OIDC is a very frequently encountered solution. It is often used to integrate with external identity providers. It is supported by Microsoft Azure Active Directory, Google, IBMid, CoreOS dex (LDAP, GitHub, SAML, etc.), and several others.

All of these authentication solutions provide user and group information (identity). This is critical to the Kubernetes authorization model. Upon authentication, authorization is handled by Kubernetes Role-Based Access Control (RBAC) support.

Authorization and RBAC

Authorization in Kubernetes is integrated into the platform as of Kubernetes v1.7. Kubernetes authorization uses an RBAC model and provides a fully featured authorization platform that allows operators to define various roles via Kubernetes objects, ClusterRole, and Role, and bind them to users and groups using ClusterRoleBinding and RoleBinding. It's worth deeper discussion of when to use each of these objects because there is often confusion regarding what is the best practice.

In this section, we look at a common use case: the need to provide TravisCI access to our cluster for provisioning new versions of our application.

ClusterRoles consist of a list of objects and verbs to determine the scope of the role. No objects are off limits for ClusterRoles, and they can include traditional objects like Pods, ReplicaSets, DaemonSets, Deployments, and so on. What makes a ClusterRole unique is that it can also refer to cluster objects such as Namespaces, Nodes, ClusterRoleBindings, and so forth. Note that a ClusterRole is never scoped to a Namespace.

In the following example, we have created a ClusterRole that will allow access to Deployments and ReplicaSets, which allows our TravisCI server to push new applications versions:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRole
metadata:
  name: cicd-apps
rules:
- apiGroups:
  - apps
  - extensions
resources:
- deployments
- replicaset
verbs:
- create
- delete
- deletecollection
- get
- list
- patch
- update
- watch
```

Roles are similar to ClusterRoles but are limited to the Namespace scoped objects such as Pod, ReplicaSet, Deployment, and RoleBinding. Also note that when a Role is created it *must* be created in a specific namespace. Here, we look at a Role equivalent to our previous example. The interesting thing to note here is that this Role could be created and managed by one of our DevOps engineers who might not have the ability to manage ClusterRoles. We have found that ClusterRole is more convenient because you can reuse it across Namespaces or apply it cluster wide. However, access to Role management, scoped to a single Namespace, can be common in a Kuber-

netes environment that is shared and administered by a number of teams where providing users with cluster-wide privileges is insecure. Note that the only difference is the kind (Role) and the inclusion of a Namespace in the metadata. Creation of this object only requires access to create Role objects in the team-a Namespace that is referenced in the following example. The preceding ClusterRole requires create access at the cluster level, and this level of access might be available to only a far more privileged user.

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: Role
metadata:
  name: teama-cicd-apps
  namespace: team-a
rules:
- apiGroups:
  - apps
  - extensions
resources:
- deployments
- replicaset
verbs:
- create
- delete
- deletecollection
- get
- list
- patch
- update
- watch
```

ClusterRoleBinding allows binding a ClusterRole to a user, group, or ServiceAccount at a cluster-wide scope. In this next example, we use our ClusterRole to provide access to all Namespaces in our cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: travis-cluster-apps
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: cicd-apps
subjects:
- kind: User
  name: travis
  apiGroup: rbac.authorization.k8s.io
```

RoleBinding allows binding a Role or ClusterRole to a user, group, or ServiceAccount at a Namespace scope. This allows an admin to

create ClusterRoles to define access such as admin, developer, auditor, and so on. In the following example, we use our `teama-cicd-apps` Role to allow the `travis` user access to only the `team-a` Namespace:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: travis-apps
  namespace: team-a
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: teama-cicd-apps
subjects:
- kind: User
  name: travis
  apiGroup: rbac.authorization.k8s.io
```

Roles and RoleBindings are interesting in that they can be managed by users who have admin rights to only those objects within a given Namespace. Modification of ClusterRole and ClusterRoleBinding requires the user to have cluster-wide access to the objects. This might be applicable for a squad that has its own Namespace and would like to set up ServiceAccounts and Roles for its Continuous Integration/Continuous Delivery (CICD) tools ([Chapter 5](#)) or other use cases.

Hybrid

The key challenge that you must address is how to provide access control in a hybrid environment. There are several different feasible approaches that are possible in hybrid environments; two key approaches that we describe here are *federated identity* and *impersonation*.

Federated identity

Using a federated-identity provider might be the most straightforward solution for centralized identity management. This approach permits the centralized management of users and identity. Many enterprise customers already have experience with identity federation for other services.

When using the federated-identity approach, it is the responsibility of the operations team to determine a reasonable method for unifying authorization across different Kubernetes clusters.

It's easy to settle on a unified set of Roles and ClusterRoles across clusters because RBAC is common everywhere. Typically, the struggle is mapping users and groups to the bindings in a unified way. An on-premises cluster might authenticate using OIDC or similar backed by Lightweight Directory Access Protocol (LDAP), in which a public cloud cluster might use a cloud-provided OIDC mechanism backed by its own Identity Access Management (IAM) mechanism. Impersonation is a strategy that you can use to overcome these inconsistencies.

Impersonation

You might want to consider an option that uses **identity impersonation**. This concept has an operator build a proxy that does authentication against an existing identity provider. The proxy then uses a cluster-admin identity native to the target Kubernetes cluster and passes impersonation headers on each request. Here's an example:

1. Bob uses `kubectl` to make a request against the proxy for cluster A.
2. The proxy authenticates Bob using an identity provider. Bob is identified as bob and belongs to groups teama and teamb.
3. The proxy authenticates with cluster A using X.509 certificates or other method and passes impersonation headers:

```
Impersonate-User: jane.doe@example.com  
Impersonate-Group: developers  
Impersonate-Group: admins
```

4. Cluster A authorizes the request based on the impersonate user and groups.
5. The same flow can be used against cluster B.

We do not provide an implementation of the impersonation proxy here, because it is beyond the scope of this book. Currently, there is not a solution provided as part of the Kubernetes open source. The advantage of this solution for hybrid clouds is that you end up with a normalized solution for user and group management, regardless of the native authentication method of all clusters involved. This allows

for creation of a single set of RBAC objects that can be utilized uniformly across all clusters.

Although impersonation does require additional development and tooling for support, it results in a strategy that makes it easier to have uniform authentication and authorization across heterogenous Kubernetes clusters.

Performance, Scheduling, and Autoscaling

For this discussion, it is important to group performance and scheduling into a single topic. The key to dealing with performance differences between various Kubernetes environments that make up a hybrid cloud is to ensure that scheduling is performed properly. The Kubernetes scheduler is completely reliant on proper resource request definitions to accurately schedule and utilize resources.

Scheduling

Understanding how the Kubernetes scheduler makes scheduling decisions is critical in order to ensure consistent performance and optimal resource utilization. All scheduling in Kubernetes is done based upon a few key pieces of information. First, it is using the information about the worker Node to decide what the total capacity of the Node is. Using `kubectl describe node <node>` will give you all the information you need to understand regarding how the scheduler sees the world, as is demonstrated here:

```
Capacity:
  cpu:                4
  ephemeral-storage:  103079200Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:              16427940Ki
  pods:                110
Allocatable:
  cpu:                3600m
  ephemeral-storage:  98127962034
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:              14932524020
  pods:                110
```

In this example, we see what the scheduler sees as being the total capacity of the worker Node, as well as the allocatable capacity. The allocatable numbers factor in kubelet settings for Kubernetes and

system reserved space. `Allocatable` represents the total space the scheduler has to work with for a given node.

Next, we need to look at how we instruct the scheduler about our workload. It is important to note that Kubernetes does not consider *actual* CPU and memory utilization of a workload. It acts on only the resource descriptions provided by the developer or operator. Here is an example:

```
resources:
  limits:
    cpu: 100m
    memory: 170Mi
  requests:
    cpu: 100m
    memory: 170Mi
```

These are the specifications provided at the *container* level. The developer must provide these specifications on a per-container basis, not per Pod. What do these specifications mean? The `limits` are considered only by the kubelet and are not a factor during scheduling. This indicates that the cgroup of this container will be set to limit CPU utilization to 10% of a single CPU core, and if memory utilization exceeds 170 MB, the process will be killed and restarted; there is no “soft” memory limit in Kubernetes use of cgroups. The `requests` are used by the scheduler to determine the best worker on which to place this workload. Note that the scheduler is adding the resource requests of all containers in the Pod to determine where to place it. The kubelet is enforcing limits on a per-container basis.

We now have enough information to understand the basic resource-based scheduling logic that Kubernetes uses. When a new pod is created, the scheduler looks at the total resource requests of the Pod and then attempts to find the worker Node that has the most *available* resources. This is tracked by the scheduler for each Node, as seen in `kubectl describe node`:

```
Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)
CPU Requests   CPU Limits    Memory Requests  Memory Limits
-----
1333m (37%)    2138m (59%)   1033593344 (6%)  1514539264 (10%)
```

You can investigate the exact details of the Kubernetes scheduler [via the source code](#). There are two key concepts in scheduling. On the

first pass, the scheduler attempts to filter the Nodes that are capable of running a given Pod based on resource requests and other scheduling requirements. On the second pass, the scheduler weighs the eligible nodes based on absolute and relative resource utilization of the nodes and other factors. The highest weighted eligible Node is selected for scheduling of the Pod.

In this chapter thus far, we have not factored in the wide variety of other scheduling hints that are available in Kubernetes to filter Nodes such as Node/Pod affinity/anti-affinity. These tools are used to adjust the pool of Nodes from which the resource-based scheduler will pull. You can learn more about this in the [Kubernetes documentation](#).

Hybrid considerations for scheduling

When managing workloads across a heterogeneous set of worker Nodes and clusters, the most important thing to take away is that not all CPU cores are created equal. One CPU core cannot always do the same amount of work as a different one. This can be caused by variations in clock speed and CPU process generation. Kubernetes does not have any way to see or manage these performance variations, so you need to consider a few things. Here are some options:

- Create resource requests and limits based on the lowest common denominator. Find the least performant target Kubernetes cluster, and create resource requests and limits based on these workers. The good news is that all memory is created equal for capacity, so this should translate one-to-one between clusters. CPU is variable depending on the physical chips underlying the worker. Therefore, you can end up underutilizing the more performant workers.
- Create custom CPU requests/limits per cluster/worker.
- Use autoscaling. We discuss several approaches for performing autoscaling in the next section.

Autoscaling

There are four common forms of autoscaling in use today, falling into two categories: monitoring-based autoscalers (*Horizontal Pod Autoscalers* and *Vertical Pod Autoscalers*), and cluster scale-based

autoscalers (*cluster proportional autoscalers* and *addon-resizer autoscalers*).

All four forms attempt to use various inputs to dynamically determine the proper replica count or resource requests. We cover each of these in more detail in the next sections.

Horizontal Pod Autoscaler

The most common form of autoscaling used is Horizontal Pod Autoscaling (HPA). Autoscaling factors in the actual CPU utilization of a Pod based upon metrics provided via the metrics API *metrics.k8s.io* (or directly from heapster pre-Kubernetes 1.11 only). With this approach, the resource requests and limits just need to be reasonable for the given workload, and the autoscaler will look at real-world CPU utilization to determine when to scale. Let's look at an example via our application:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  labels:
    run: hello
  name: hello
spec:
  selector:
    matchLabels:
      run: hello
  template:
    metadata:
      labels:
        run: hello
    spec:
      containers:
        - image: kitch/hello-app:1.0
          name: hello
          resources:
            requests:
              cpu: 20m
              memory: 50Mi
---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: hello
  name: hello
spec:
  ports:
```

```
- port: 80
  protocol: TCP
  targetPort: 8080
selector:
  run: hello
```

We now have a simple web app to begin exploring autoscaling. Let's see what we can do from a scaling perspective. Step one—create an autoscaling policy for this deployment:

```
$ kubectl autoscale deploy hello --min=1--max=5--cpu-percent=80
deployment.apps "hello" autoscaled
$ kubectl get hpa hello
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS
hello Deployment/hello 0%/80% 1 5 1
```

Excellent! We are ready to scale! Let's throw some load at our fancy new web application and see what happens next. Now when we check to see the CPU utilization of our Pod, we can see it is using 43m cores:

```
$ kubectl top pods -l run=hello
NAME CPU(cores) MEMORY(bytes)
hello-7b68c766c6-mgtdk 43m 6Mi
```

This is more than double the resource request we specified:

```
$ kubectl get hpa hello
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS
hello Deployment/hello 215%/80% 1 5 1
```

Note that the HPA has increased the number of replicas:

```
$ kubectl get hpa hello
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS
hello Deployment/hello 86%/80% 1 5 3
```

Utilization is still above our policy limit, and thus in time the HPA will continue to scale up once more and reduce the load below the threshold of the policy:

```
$ kubectl get hpa hello
NAME REFERENCE TARGETS MINPODS MAXPODS REPLICAS
hello Deployment/hello 62%/80% 1 5 4
```

It's important to note that the metrics collection and HPA are not real-time systems. The Kubernetes documentation speaks in a bit more detail about the [controller-manager settings and other intricacies of the HPA](#).

Finally, we reduce the load on the deployment, and it is automatically scaled down again:


```
$ kubectl get hpa hello
NAME      REFERENCE          TARGETS  MINPODS  MAXPODS  REPLICAS
hello     Deployment/hello   0%/80%   1         5         1
```

How does this help us in hybrid scenarios? It ensures that regardless of the inherent performance of any one cluster or worker Node, the autoscaler will ensure that we have the appropriate resources allocated to support our workload.

Vertical Pod Autoscaler

The Vertical Pod Autoscaler (VPA) is an excellent solution for situations in which you have a Deployment that needs to scale up rather than out. Whereas the HPA adds more replicas as memory and CPU utilization increase, the VPA increases the memory and CPU requests of your deployment. For this example, let's reuse our hello example again. We begin by installing the VPA **according to the steps provided**. If you recall, we started with requests of 20 millicores. First, let's apply our VPA:

```
apiVersion: poc.autoscaling.k8s.io/v1alpha1
kind: VerticalPodAutoscaler
metadata:
  name: hello-vpa
spec:
  selector:
    matchLabels:
      run: hello
  updatePolicy:
    updateMode: Auto
```

Now, let's apply load to the application and observe the appropriate response:

```
$ kubectl top pods -l run=hello
NAME                                CPU(cores)  MEMORY(bytes)
hello-7b68c766c6-mgtdk             74m         6Mi
```

We can then view the resource requests for our hello Deployment and see that they have been automatically adjusted to match real-world utilization of our application:

```
resources:
  requests:
    cpu: 80m
    memory: 50Mi
```

Cluster Proportional Autoscaler

There are a couple of other common autoscaler implementations that are used in addition to the HPA. The first of these is the **Cluster Proportional Autoscaler**, which looks at the size of the cluster in terms of workers and resource capacity to decide how many replicas of a given service are needed. Famously, this is used by **KubeDNS**; for example:

```
spec:
  containers:
  - command:
    - /cluster-proportional-autoscaler
    - --namespace=kube-system
    - --configmap=kube-dns-autoscaler
    - --target=Deployment/kube-dns-amd64
    - --default-
  params={"linear":{"coresPerReplica":256,"nodesPerReplica":16,
"preventSinglePointFailure":true}}
    - --logtostderr=true
    - --v=2
```

The number of cores and nodes are used to determine how many replicas of KubeDNS are needed.

addon-resizer

Another great example is the **addon-resizer** (aka `pod_nanny`), which performs vertical scaling of resource requests based upon cluster size. It scales the resource's requests of a singleton based on the number of workers in the cluster. This autoscaler has been used by `heapster`:

```
- command:
  - /pod_nanny
  - --config-dir=/etc/config
  - --cpu=80m
  - --extra-cpu=0.5m
  - --memory=140Mi
  - --extra-memory=4Mi
  - --threshold=5
  - --deployment=heapster
  - --container=heapster
  - --poll-period=300000
  - --estimator=exponential
```

Performance

How does all of this autoscaling help you? The key here is that these autoscaling options allow you to tie real-world resource utilization data back to the scale of your application. Because of this, even if the resource requests that you used to schedule a given Pod might not be 100% accurate to what the real-world utilization is, the autoscaler will adjust the total requested/utilized resources of the application. The end result of this interaction between autoscaler and scheduler will be that regardless of the performance and utilization of any individual Pod or worker Node, the system of all Pods for a given application will be balanced properly to meet your performance requirements.

Networking

Kubernetes networking is both simple *and* complex. Simplicity comes from the concept that Kubernetes itself is doing very little networking magic. The complexity comes from the various networking plug points and concepts that are a part of Kubernetes. Here are the core Kubernetes networking concepts:

- Cluster (aka Pod) networking
- Services/kube-proxy/load balancers
- Ingress
- Network/security policy

In addition to the aforementioned core networking concepts, hybrid cloud environments must also integrate some form of support for Virtual Private Networks (VPNs). We discuss all of these topics in the sections that follow.

Pod Networking

The most central networking component of Kubernetes is the Pod networking. This is what makes each Pod (and the containers within it) network addressable. Kubernetes itself implements only the most basic networking for Pods. This is the Pod network Namespace, which is shared by all containers of the Pod. The Pod network Namespace allows all the containers of the Pod to communicate with one another as if they were both running in their own dedica-

ted host. Container A is able to communicate with Container B via localhost. All containers within the Pod share a single port space, just as two processes on one compute host must contend for available ports. The Pod network Namespace is provided by the Kubernetes pause container, which does nothing other than create and own the network Namespace.

Next up is the Container Network Interface (CNI) plug-in that connects the Pod network Namespace to the rest of the Pods in the Kubernetes cluster. There are a huge number of different CNI plug-ins available for various network implementations. See [the official documents](#) for more on CNI plug-ins. The most important thing to know is that the cluster networking is what provides Pod-to-Pod communication within the cluster and IP address management.

Services/kube-proxy/Load Balancers

Kubernetes leans heavily upon the concept of microservices; the idea being that there are many services within the cluster that provide functional units and are accessed from within the cluster or from outside the cluster as Services. Much has been written on the topic, and the [Kubernetes documentation details Services](#). The most common use case for Services is to create a Kubernetes Deployment or ReplicaSet and expose that group of Pods as a single, scalable, highly available Service.

The kube-proxy component is what makes all of this possible. We won't go into the details of the kube-proxy mechanics here, but, essentially, it provides very efficient and highly available load balancing of Kubernetes Services within the cluster.

Finally, we have *load balancers*, which are a concept within Kubernetes that allow access to a Kubernetes Service from outside the cluster. When creating a service of type=LoadBalancer, you are indicating that the service should be externally available. Your Kubernetes Service provider or bare-metal solution will determine exactly the implementation details of the load balancer. In all cases, you end up with an external IP address, which can be used to access the Service. You can find more details in the [Kubernetes documentation](#).

Of Services, kube-proxy, and load balancers, hybrid considerations typically arise only for load balancers. This is because in many cases

Load Balancers are cloud-provider specific and can have unique annotations and behaviors. You should investigate these to ensure that consistent behavior, security, and performance occurs across platforms.

Ingress

Ingress provides a Kubernetes object model to make Services available outside the Kubernetes cluster via a Layer 7 application load balancer. An example would be providing public internet access to a web service or web page. Your Ingress controller of choice will determine the exact implementation. You need only create an Ingress object to indicate the desire to expose a Kubernetes Service externally via an application load balancer. Here's an example object:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: hello-ingress
spec:
  rules:
  - host: kitch.us-east.containers.appdomain.cloud
    http:
      paths:
      - path: /
        backend:
          serviceName: hello
          servicePort: 80
```

This indicates to the Ingress controller that the hello Service should be exposed on the `kitch.us-east.containers.appdomain.cloud` hostname. It is then up to the Ingress controller to realize this desired state.

Hybrid strategies are heavily affected by Ingress because the Ingress controller of each provider supports a wide variety of annotations that are associated with each implementation. Because of this variability between providers, the Ingress objects are often managed by the operations team rather than the development team. In this scenario, the developers might request to have a specific Service exposed via a specific Ingress; the operators can then create the Ingress object, and the development team is free to make changes and update the backend service implementation at any time. One option to provide consistency between Kubernetes clusters is to use a community Ingress controller with a Kubernetes Load Balancer

Service. To do this, you would deploy an Ingress controller such as the Kubernetes `ingress-nginx` with multiple replicas and then expose it by using a Service such as:

```
$ kubectl expose deploy ingress-nginx --port 443 --type
LoadBalancer
```

When this is complete, the Ingress controller can be used as normal. This provides a uniform ingress experience across heterogeneous clusters. Note that if you go this route, you will either need to disable other Ingress controllers or use an annotation in your Ingress objects to specify which controller should handle the object. You can specify the Kubernetes `ingress-nginx` controller using the following:

```
annotations:
  kubernetes.io/ingress.class: "nginx"
```

There is [additional documentation available](#) for introducing multiple instances of the same type of controller.

Network Security/Policy

There are two levels of network security that we need to discuss as it relates to Kubernetes: securing the worker Nodes and securing the Pods. Securing Pods in Kubernetes is the domain of Kubernetes NetworkPolicy, which, when used with a [CNI plug-in that supports policy](#), allows users and operators to control network access controls at the Pod level. It's also possible to use Istio for Pod network policy, which we discuss in [“Istio” on page 129](#). Network Security of the worker Nodes is not the domain of Kubernetes. You must secure the Nodes with external tools such as a [Virtual Private Cloud \(VPC\)](#), network Access Control List (ACL), or security groups from your cloud provider, iptables, or a cloud security solution such as [Project Calico](#).

NetworkPolicy

NetworkPolicy objects are used to control networking traffic entering and leaving Kubernetes Pods. Entering network traffic is commonly referred to as ingress traffic, and networking traffic leaving the Pod is commonly referred to as egress traffic. The NetworkPolicy object is quite flexible and its capabilities have grown considerably over time. We won't go into the details of how to construct

NetworkPolicy or the underlying implementation, [the official Kubernetes documentation](#) does an excellent job of this.

Let's look at a few examples that operators are likely to encounter. One topic that is not discussed extensively in the Kubernetes documentation is using NetworkPolicy to provide access control into the cluster or out of the cluster. It's not uncommon to start with a deny-all policy for a Namespace, such as demonstrated here:

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

In many cases, there might be a desire to allow egress to an external web service. In this case, you can use an `ipBlock` to allow egress for a service. In this example, we have purchased a public cloud-based Redis service. Our team is going to deploy Pods that need to access this service. By default, this traffic would be blocked. This policy will allow the in-cluster Pods to access the internet Redis service. Suppose that the Redis service that we are trying to access is available at 10.10.126.48/28 on port 6379. A simple policy will allow this:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-redis
  namespace: teama
spec:
  podSelector: {}
  policyTypes:
  - Egress
egress:
  - to:
    - ipBlock:
        cidr: 10.10.126.48/28
      ports:
        - protocol: TCP
          port: 6379
```

If you use a LoadBalancer or Ingress solution that is capable of supporting source IP preservation, you can also use NetworkPolicy to restrict the addresses that can access Pods. This is a commonly

desired solution for hybrid scenarios because some services might be hosted publicly, but operators might desire to allow only on-premises users to access the service. We can use our default-deny policy to get started. Now we want to allow our corporate network (159.27.17.32/17) to access our HR application:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-corporate-to-hr-frontend
  namespace: teama
spec:
  podSelector: {}
  policyTypes:
  - Ingress
  ingress:
  - from:
    - ipBlock:
        cidr: 159.27.17.32/17
    - podSelector:
        matchLabels:
          app: hr-frontend
  ports:
  - protocol: HTTP
    port: 8080
```

This is just one example of the power of `NetworkPolicy`. Here, we looked at ingress/egress via `ipBlock`. Many times, `NetworkPolicy` is used to secure and isolate between Kubernetes Pods as well. In these cases, we are able to use the `podSelector` option rather than `ipBlock` to create a simple policy to secure our Pods from one another.

Worker Node network security

As mentioned in this chapter, there are any number of possible solutions available to secure the network of your worker Nodes. We won't go into the details of the cloud provider solutions here, given that they are unique to each provider. That said, in hybrid you often have your own "bare-metal" clusters in which you want to secure your worker Nodes. A solution like Project Calico is great in these environments. Calico has its own native `GlobalNetworkPolicy` object. You might even find it desirable to use in cloud provider environments to have a consistent method for managing worker security in hybrid scenarios.

Let's look at a very brief example that might be used to allow access to a nodeport range on the workers of ports 30000 through 32767. We'll also allow all egress and use Kubernetes Network policy to control egress:

```
apiVersion: projectcalico.org/v3
kind: GlobalNetworkPolicy
metadata:
  name: allow-nodeports
spec:
  selector: role == 'all'
  types:
  - Ingress
  - Egress
  ingress:
  - action: Allow
    protocol: TCP
    destination:
      ports:
      - 30000:32767
  egress:
  - action: Allow
```

Calico implements these network policies using iptables rules. The core advantage of using Calico GlobalNetworkPolicy is that it allows the admin to programmatically control access across an entire range of nodes using the selectors.

Istio

We talk only briefly about **Istio** here, given that we could write an entire book on this topic alone. In this section, we do provide a short overview of the features of Istio so that you can make your own decision as to whether it is worth exploring this framework in greater depth.

At its heart, **Istio** is a service mesh framework that provides some of the same microservices features that you will find in Kubernetes itself, such as service discovery and load balancing. In addition, Istio brings with it traffic management, service identity and security, policy enforcement, and telemetry. Two Istio concepts that operators might find particularly appealing are **mutual TLS**, to secure Pod-to-Pod traffic using centrally managed certificates, and **egress policy**. Egress policy in Istio is especially appealing because it removes the need to create policy based on external service Classless Inter-Domain Routing (CIDR) blocks and allows policy definitions based

on URL. Allowing access to an external Compose MongoDB service is as simple as doing the following:

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: compose-mongodb
spec:
  hosts:
    - sl-us-south-1-portal.27.dblayer.com
    - sl-us-south-1-portal.28.dblayer.com
  ports:
    - number: 47400
      name: mongodb
      protocol: TCP
```

This policy allows all of our Pods and Services running in the Kubernetes cluster to have egress access to our publicly hosted MongoDB Service. This allows you to access this Service regardless of whether it might be hosted behind a Content Delivery Network (CDN) or other proxy-based solution. No more chasing IPs.

There is a large array of operational and architectural problems that are neatly solved by Istio. We strongly recommend that you investigate it as your use of Kubernetes grows and becomes more advanced.

Virtual Private Networks

In hybrid cloud environments, one of the most important challenges to overcome is network connectivity. You can have Kubernetes clusters running in a variety of networking environments with extremely limited network connectivity. Virtual Private Networks (VPNs) are a common solution to solve these challenges. A VPN can create connectivity solutions between firewalled networking environments. Let's take a look at an example in which we have a publicly hosted Kubernetes cluster, and one in our on-premises datacenter. In this scenario, we want to keep all services and access to the cluster as near to fully private as possible. VPN to the rescue!

You'll find many different VPN solutions for Kubernetes out there. We're a bit partial to IBM Cloud solutions, and it just so happens there is a fantastic StrongSwan-based IPsec VPN solution to allow connecting IBM Cloud Kubernetes Service clusters to IBM Cloud Private. You can find stunningly detailed [documentation on deploying this VPN solution](#). Regardless of what solution you go with for

VPN, you might consider using a single solution across all your clusters.

Storage

Earlier, we spoke about some of the storage concepts in Kubernetes, including Volumes, PersistentVolumes (PVs), and PersistentVolumeClaims (PVCs). From an operations standpoint, there are not too many issues that need to be addressed. One consideration is how the storage is managed. In some shops, there might be a storage admin who has the proper RBAC to create PVs and PVCs. The developer only has access to refer to the PVCs. Of course, a trusting admin might provide RBAC for developers to create their own PVCs directly, especially in environments that support dynamic storage provisioning because it dramatically simplifies the ability to rapidly deploy and scale applications that are dependent on storage.

One area where storage knowledge is critical is when dealing with **StorageClasses**. StorageClasses are used to specify the performance and other characteristics of the backing physical storage. In hybrid environments, you are certain to see variability in Storage Classes, and here is where a knowledgeable storage admin and performance engineer can help to find and create comparable classes for use across hybrid clusters. This allows the developer to get consistent performance characteristics.

Kubernetes Volume Plug-ins

Key to storage support in Kubernetes are volume plug-ins. Kubernetes supports two types of out-of-tree volume plug-ins: CSI and FlexVolume. Both of these options provide ways to extend Kubernetes to support various backing storage solutions for PVs. Here, we review a few of the volume plug-ins that you will encounter with IBM Cloud.

IBM Cloud Kubernetes Service

IBM Cloud provides two FlexVolume plug-ins for use with IBM Cloud Kubernetes Service: IBM File Storage for IBM Cloud and IBM Block Storage for IBM Cloud. These drivers allow users to easily access IBM Cloud-managed storage options for persistence. In addition, these plug-ins provide a variety of StorageClasses supporting a wide range of capacity and performance needs. More

details are available in the [IBM Cloud Kubernetes Service documentation](#).

GlusterFS

GlusterFS is a distributed filesystem that can be exposed to Kubernetes. GlusterFS is an example of software-defined storage, in which arbitrary block storage devices (called *bricks*) have a replicated filesystem managed across them. The advantage of course is that if Nodes that are hosting a portion or replica of the filesystem fail, other Nodes will still be able to access the data from surviving replicas. There are a few abstraction layers to be aware of, so let's go through them.

- On each worker Node, a filesystem client allows the worker to mount a replica from GlusterFS for local availability.
- A Kubernetes plug-in makes these filesystems available to be mounted into containers.
- Heketi, an API abstraction layer, allows the dynamic provisioning of distributed volumes in GlusterFS. Heketi interacts with Kubernetes as a dynamic storage provisioner. A Storage Class is registered which provides a dynamic endpoint, which is serviced by the Heketi API.
- Each PVC declares its expected Storage Class as well, along with attributes like the required size of the volume, the required shared read/write characteristics, and the expected recycle behavior.

When Kubernetes detects a new PVC with our GlusterFS Storage Class, an API request is made to Heketi to create a matching PV. Finally, the PV is mounted into the container and the local container process is able to read and write data to it. Because we've chosen GlusterFS as our filesystem, each write is replicated across a number of storage locations, which is controlled by a setting in the Storage Class.

Quotas

Resource quotas are a critical aspect whenever managing shared resources. Kubernetes allows you to control resource allocations at many levels, including fine-grained controls per Pod such as

restricting CPU and memory assigned to containers (see “[Performance, Scheduling, and Autoscaling](#)” on page 116) or per Namespace.

Let’s take a look at an example quota declaration for our stock-trader Namespace:

```
---
kind: ResourceQuota
apiVersion: v1
metadata:
  name: stock-trader-quota
  namespace: stock-trader
spec:
  hard:
    limits.cpu: '8'
    limits.memory: 8Gi
    requests.cpu: '4'
    requests.memory: 4Gi
    services: '25'
    persistentvolumeclaims: '25'
```

In this example, we set the bounds for requests (how much to reserve) and limits (how much to allow). Expressing the lower bound is helpful to ensure that your Kubernetes cluster will have sufficient capacity for all intended workloads.

`limits.cpu`

The sum of all CPU shares used by Pods in the Namespace may not exceed this value. You may express fractional values in the range of 0 to 1 (e.g., 0.5 is one-half of a CPU), or use the suffix “m” to denote millishares (e.g., 800m is equivalent to 0.8 or 8/10 of a CPU).

`limits.memory`

The sum of all memory shares used by Pods in the Namespace may not exceed this value. The unit expressed is typically in gigabytes (“Gi”). You may also specify alternative units including E, P, T, G, M, or K for expressing the number of bytes in powers of 10, or Ei, Pi, Ti, Gi, Mi, or Ki for expressing the number of bytes in powers of 2.

`requests.cpu`

The amount of CPU capacity to reserve on the cluster. You may express units as you do with `limit.cpu`.

`requests.memory`

The amount of memory to reserve on the cluster. You may express units as you do with `limit.memory`.

`services`

A count of the total Services allowed in the Namespace.

`persistentvolumeclaims`

A count of the total PVCs allowed in the Namespace.

The most important thing to remember is to set `requests` for quotas and Pods. Requests ensure that sufficient capacity is reserved and is critical for effective, balanced scheduling of Pods across the nodes in your cluster. More details on `requests` are provided in [“Performance, Scheduling, and Autoscaling” on page 116](#).

The previous example shows `services` and `persistentvolumeclaims`, but most Kubernetes resources can be expressed to provide an absolute limit on the number of these kinds of resources allowed in the Namespace. We express these two because we find they are the most important to limit.

Note that `resourcequota` is its own kind and it is Namespace scoped. Hence, the quota that governs a Namespace is defined within the Namespace. When you define Kubernetes Roles and Role Bindings, be sure to limit the ability to create and modify `resourcequota` objects to only the administrators or operators who govern control of the cluster. Otherwise, you might expose yourself to an enterprising developer who wants more capacity than intended.

You apply quotas like all other resources in Kubernetes:

```
$ kubectl apply -f resource-quota.yaml
resourcequota "stock-trader-quota" created
```

If you are using IBM Cloud Private, you can also use the web console to make it easier to create resource quotas, under `Manage > Quotas`. [Figure 7-1](#) demonstrates the creation of a resource quota for the `stock-trader` Namespace using the tools provided by IBM Cloud Private.

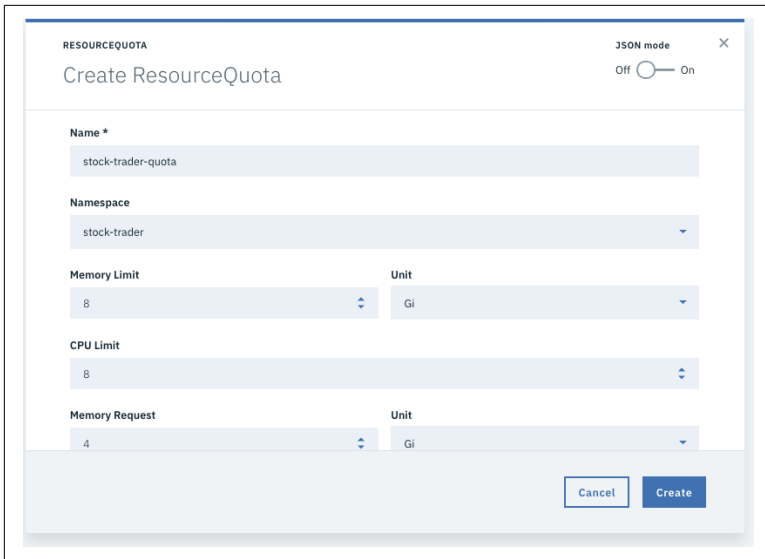


Figure 7-1. Using IBM Cloud Private to create resource quotas.

Audit and Compliance

There are two key areas for which audit should be considered for Kubernetes, the kube-apiserver and the worker Nodes. For the apiserver, there are two factors to consider. First is the **Audit Policy**, which determines which requests will be audited and how much data for the request will be audited. A careful balance is needed to maintain performance while still meeting any compliance needs. Your cloud provider might include settings for the apiserver audit logging. The second consideration for the kube-apiserver auditing is where those audit events are stored. You can choose either a local log file or a webhook-based backend. Typically, your cloud provider can provide some integrated capabilities. IBM Cloud Kubernetes Service allows the operator to send logs either to IBM Log Analysis or use the webhook for **delivering the audit to a third-party**.

In addition to the kube-apiserver, we recommend that you also collect audit information from all worker Nodes. The standard for audit log collection on Linux is auditd. You can find any number of tutorials on **configuring and auditing** or **system monitoring** with auditd. The key with Kubernetes is to configure auditd on all worker Nodes and use Fluentd for collection of that auditd data. Cloud providers might cover worker audit as part of their service.

Kubernetes Federation

Kubernetes Federation is a must to cover for operations whenever you are covering hybrid scenarios. Federation allows you to manage resources of two or more Kubernetes clusters via a single API endpoint.

There are quite a few caveats to consider in a federated configuration. One issue is how authentication and authorization are performed in federation, given that it's difficult to manage their hierarchy across clusters.

Federation works by replicating or distributing objects across the clusters included in the federation. Thus, if a ConfigMap is created, it is replicated across all clusters in the federation. If a ReplicaSet is created, it is replicated across all clusters in the federation with an equal number of Pods in each cluster.

In addition, federation is responsible for configuring DNS servers in each cluster to enable cross-cluster service discovery. Depending on the network configuration of clusters in the federation, this might or might not enable some level of cross-cluster communication. Users will also see varying levels of cross-cluster load balancer and ingress configurations.

NOTE

“Maturity: The **federation project** is relatively new and is not very mature. Not all resources are available and many are still alpha. There are **issues** that enumerate known issues with the system that the team is busy solving.”

It is also worth noting that as of this publication, the Kubernetes sig-federation is working on a **v2 proposal**. This work hopes to address scheduling, management, and authentication/authorization struggles from the original federation work.

Contributor Experience

By this point in the book, hopefully we have convinced you that Kubernetes is a popular and exciting technology that will bring substantial value to your enterprise. But, if you really want excitement, we highly recommend that you become a contributor to the Kubernetes community. The Kubernetes contributor community is an amazing group of very friendly people with a tremendous amount of collective knowledge about cloud computing and container technologies. By becoming a contributor, you will have the opportunity to expand your skills and strengthen your professional network. Contributing to Kubernetes can take several forms, including contributing to source code, opening bug reports, and contributing to documentation.

Learning how to contribute to an open source project like Kubernetes can be quite intimidating if you have never contributed to an open source project in the past. Fortunately, there are several resources available to help accelerate your journey to being a contributor to the Kubernetes community. In this chapter, we highlight several of these excellent online resources.

Kubernetes Website

The primary [Kubernetes website](#) is an excellent place to begin looking for information on Kubernetes. As shown in [Figure 8-1](#), the home page for Kubernetes provides links for more information on topics such as documentation, community, blogs, and case studies. In the community section of the Kubernetes website, you will find

more information on how to join the large number of Kubernetes Special Interests Groups (SIGs). Each SIG focuses on a specific aspect of Kubernetes, and hopefully you can find a group that excites you and matches your interests.

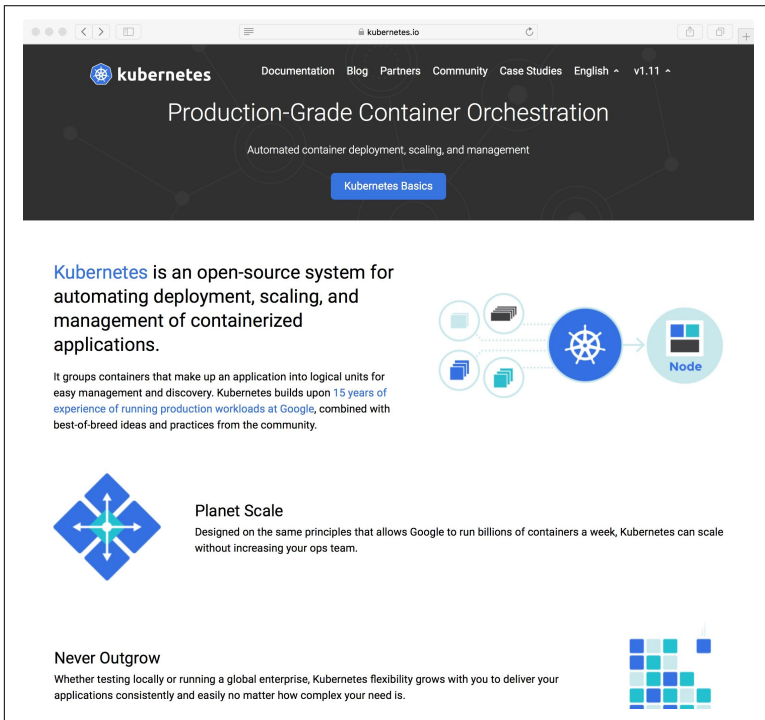


Figure 8-1. *Kubernetes website home page.*

The Cloud Native Computing Foundation Website

Figure 8-2 shows the **Cloud Native Computing Foundation** (CNCF) website. It provides a large amount of information on a variety of cloud-native computing projects that are hosted by the CNCF. The projects under the umbrella of the CNCF include Kubernetes, Prometheus, Envoy, Containerd, Fluentd, Helm, and several others. In addition, the CNCF provides several Kubernetes educational training modules and also has a Certified Kubernetes Application Developer (CKAD) Program.

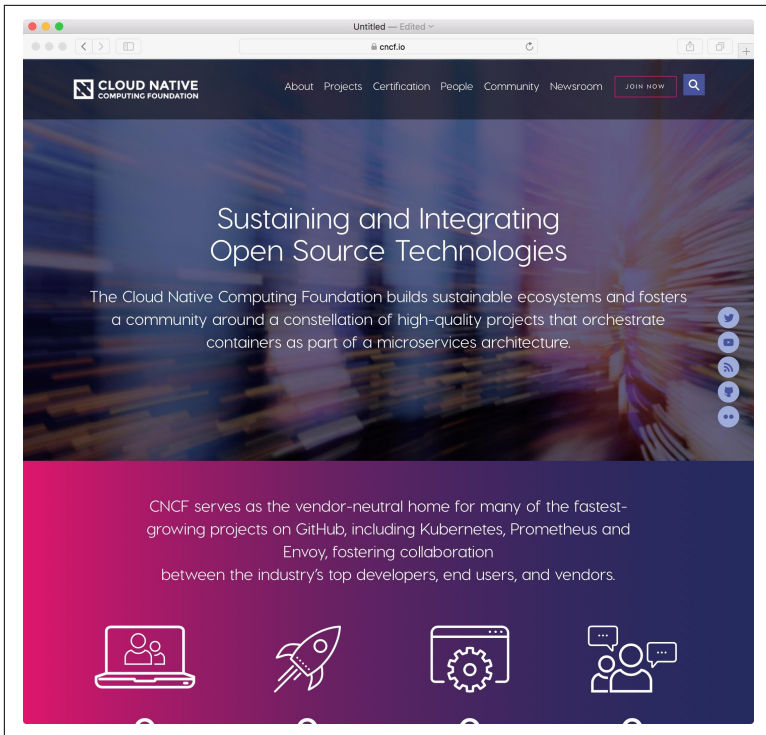


Figure 8-2. *Cloud Native Computing Foundation website home page.*

IBM Developer Website

The **IBM Developer Website** provides a large number of Kubernetes-based code patterns. This website semantically links code, content, and community to empower developers and provide them with support as they learn new open source technologies. At IBM Developer, developers can take guided learning paths through a variety of open source topics and solutions to progress their technical depth and expand their personal eminence in open source. A large portion of the IBM Developer Website is focused on Kubernetes code samples and tutorials, as well as its use by adjacent technologies. **Figure 8-3** depicts the Kubernetes portion of the IBM Developer Website.

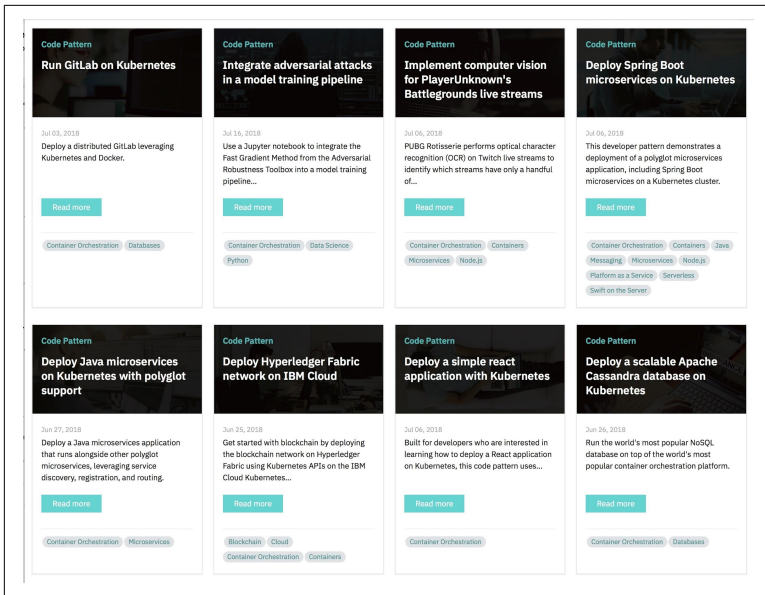


Figure 8-3. IBM Developer Website Kubernetes code patterns page.

Kubernetes Contributor Experience SIG

The Kubernetes community takes the happiness of its contributors very seriously. In fact, they have an entire SIG, the *Contributor Experience SIG*, dedicated to improving the contributor experience. The Contributor Experience SIG is an amazing group of folks who want to know more about you and understand the issues you might be encountering as you become a Kubernetes contributor. The Contributor Experience SIG website, shown in [Figure 8-4](#), is located on [Kubernetes community GitHub](#). Take some time to visit this website for more information on how to contact the Contributor Experience SIG and learn more about the contributor topics it focuses on.

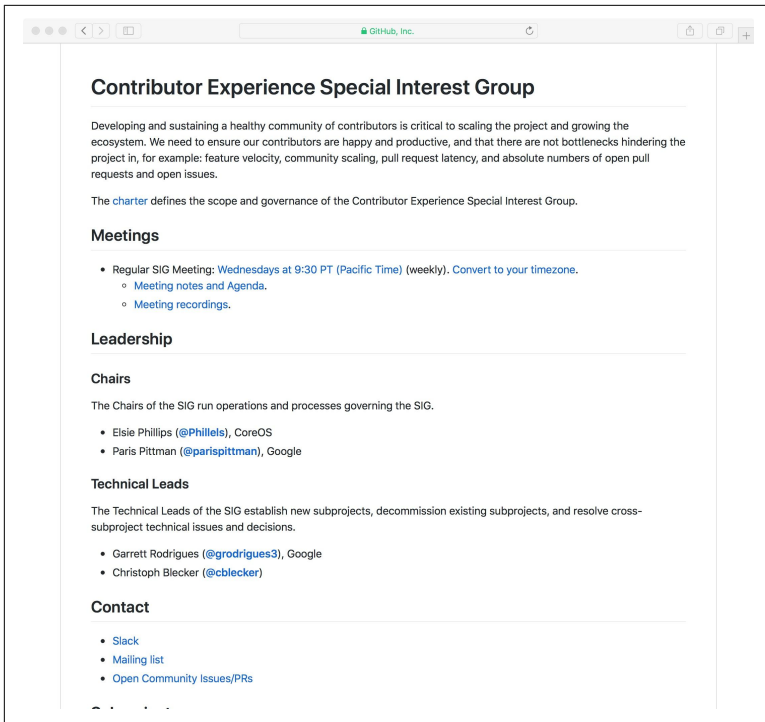


Figure 8-4. *Kubernetes Contributor Experience SIG website home page.*

Kubernetes Documentation SIG

If you don't feel comfortable contributing to Kubernetes source code but still have a strong desire to participate in the Kubernetes community as a contributor, the Kubernetes Documentation SIG may be the perfect fit for you. The **Kubernetes Documentation SIG** maintains the Kubernetes documentation repository. Figure 8-5 presents a snapshot of the home page for this SIG. The GitHub pull request process this team utilizes to accept contributions is essentially the same process used by Kubernetes code repositories. Because of this, the skills you acquire learning how to be a documentation contributor will also help you even if your long-term goal is to be a Kubernetes code contributor. In addition, the Kubernetes Documentation SIG will typically run Documentation Sprints at all the KubeCon/CloudNativeCon conferences. At these sprints, you get hands-on training from Kubernetes Documentation Maintainers on how to become a Kubernetes documentation contributor in a small team

setting. This environment is perfect for new potential contributors who need a little extra help getting started or feel more comfortable learning in smaller groups.

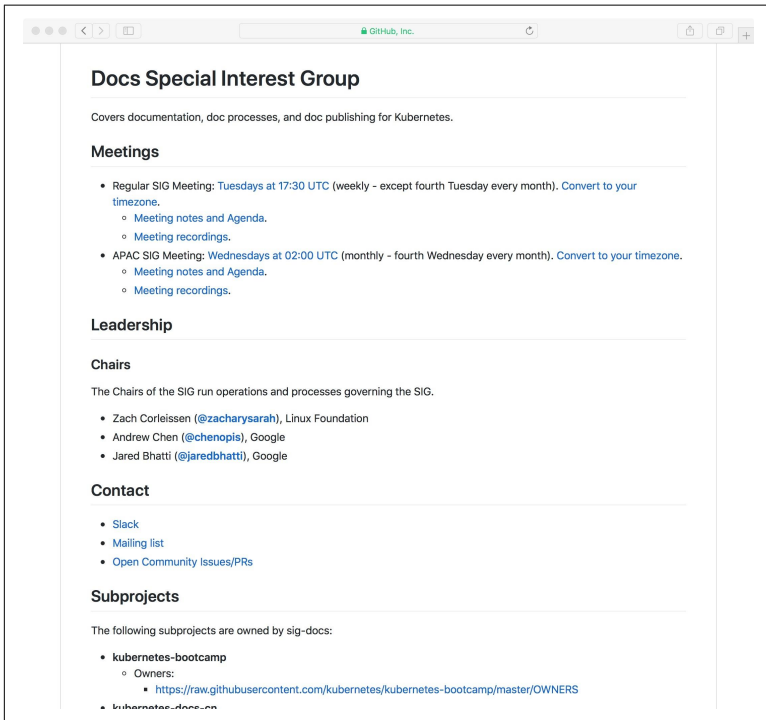


Figure 8-5. *Kubernetes Documentation SIG website home page.*

Kubernetes IBM Cloud SIG

If you are interested in following the evolution of the IBM Cloud Kubernetes Service and IBM Cloud Private platforms, this is the group for you. Many developers and leaders from IBM Cloud work openly in this group to determine the future of IBM contributions and involvement in the Kubernetes community. You can also interact directly with the team that builds and operates IBM Cloud. You can find more information on the group and its meetings at its [GitHub page](#).

The Future of Kubernetes

Spend some time at a KubeCon/CloudNativeCon conference these days, and you will quickly come to the conclusion that Kubernetes' future is very bright. Attendance at KubeCon/CloudNativeCon conferences continues to experience an explosive level of growth. In addition, the **Kubernetes open source community of contributors** continues to expand and strengthen. The number of industries that are adopting Kubernetes is just astounding. In this chapter, we make some predictions on what the future holds for Kubernetes. Specifically, we expect to see Kubernetes growth to occur in the areas of legacy application migration to cloud-native applications, high-performance computing, machine learning and deep learning applications, and hybrid cloud environments.

Increased Migration of Legacy Enterprise Applications to Cloud-Native Applications

As enterprise success stories with Kubernetes continue to be publicized and highlighted at KubeCon/CloudNativeCon conferences, we anticipate that more and more enterprise legacy applications will be moved to run as cloud-native computing applications. Accelerating this transformation will be tooling and containerized enterprise middleware tailored toward simplifying the process of moving enterprise applications to Kubernetes-based cloud-native environments. More and more enterprise customers will experience the benefits of improved application quality, reduced defects, reduced deployment times, and improved automation and DevOps that

become possible when embracing a [Kubernetes container-based development methodology](#).

Increased Adoption of Kubernetes for High-Performance Computing

Kubernetes and its container-based approach provides several benefits that make the environment well suited for high-performance computing applications. Because Kubernetes is container based, the platform experiences less overhead to start up new tasks, and the tasks can be a finer-grained operation than those supported by virtual machine (VM)-based cloud computing environments. The reduction of latency associated with the creation and destruction of computational tasks that occurs when using containers instead of VMs improves the scalability of a high-performance computing environment. Furthermore, the increased efficiency that is possible by packing a larger number of containers onto a physical server in contrast to the limited number of VMs that can be placed on a physical server is another critical advantage for high-performance applications.

In addition to reduced latency, Kubernetes environments also support a parallel work queue model. You can find an excellent overview of the Kubernetes work queue model in *Kubernetes Up and Running*, by Kelsey Hightower, Brendan Burns, and Joe Beda (O'Reilly). The work queue model described in this book is essentially the “bag of tasks” parallel computing model. Research has shown that this parallel computing model is a superior approach for the execution of high-performance parallel applications in a cluster environment.¹ Because of all these factors, and also the large number of cloud computing environments that offer Kubernetes-based environments, we expect a huge growth in adoption of Kubernetes by the high-performance computing community.

¹ Schmidt BK, Sunderam VS., (1994). “Empirical Analysis of Overheads in Cluster Environments,” *Concurrency Practice & Experience*, 6: 1–32.

Kubernetes Will Become the de Facto Platform for Machine Learning and Deep Learning Applications

Machine learning and deep learning applications typically require highly scalable environments, and data scientists with expertise in these domains might have limited expertise running in production at scale. Similar to our justification provided in the previous section for adoption of Kubernetes for high-performance computing, we anticipate machine learning and deep learning environments to greatly benefit from adopting Kubernetes-based environments as their primary platform. In fact, initiatives such as [Kubeflow](#), which are focused on providing an open source Kubernetes-based platform for machine learning applications, are already attracting a significant number of contributors to their open source project.

Kubernetes Will Be the Platform for Multicloud

If you have taken the time to read this book, this last prediction should not be a surprise. With Kubernetes experiencing huge growth and being made available in numerous public cloud and private cloud offerings, and with its focus on interoperability and the ease of container-based workload migration, Kubernetes is well positioned to be the ideal platform for multicloud environments. Kubernetes future looks very bright, and exciting times are ahead!

Conclusions

In this book, we have covered a broad number of Kubernetes topics. We provided a historical overview of the rise of both containers and Kubernetes and the positive impact of the Cloud Native Computing Foundation. We described the architecture of Kubernetes, its core concepts, and its more advanced capabilities. We then walked through an enterprise-level production application and discussed approaches for continuous delivery. We then explored operating applications in enterprise environments with a focus on log collection and analysis, and health management. We also looked at Kubernetes cluster operations and hybrid cloud-specific considerations and issues. Finally, we presented several resources that are available

to help you become a contributor to the Kubernetes community, and we ended with a short discussion on what the future holds for Kubernetes. We hope that you have found this book helpful as you begin your journey of deploying enterprise quality Kubernetes applications into production environments, and hope it accelerates your ability to fully exploit Kubernetes-based hybrid cloud environments.

Configuring Kubernetes as Used in This Book

Throughout this book, we use two Kubernetes providers: one to demonstrate Kubernetes as a managed service, which you can run in IBM's worldwide datacenters; and the second to demonstrate Kubernetes as a software package that you can install on your infrastructure of choice.

Configuring IBM Cloud Private in Your Datacenter

The following section describes how to configure IBM Cloud Private and the supporting command-line interface to use when running the examples discussed in this book.

Configuring an IBM Cloud Private Kubernetes Cluster

There are a number of ways to get started with your own enterprise-grade Kubernetes cluster on your own infrastructure.

First, as a software distribution of Kubernetes, you can deploy IBM Cloud Private on your own infrastructure (VMware, bare metal, OpenStack) or various public cloud providers. Visit the [GitHub repository](#) for ready-to-go automation.

For local experiments, you can simulate a multiworker cluster on your own laptop via the following code:

```
git clone https://github.com/IBM/deploy-ibm-cloud-private.git
cd deploy-ibm-cloud-private
```

Open the Vagrantfile and customize it for your machine's capacity:

```
# Vagrantfile
...
# most laptops have at least 8 cores nowadays (adjust based
# on your laptop hardware)
cpus = '2'

# this will cause memory swapping in the VM
# performance is decent with SSD drives but may not be with
# spinning disks
#memory = '4096'

# use this setting for better performance if you have the ram
# available on your laptop
# uncomment the below line and comment out the above line
# "#memory = '4096'"
memory = '10240'
...
```

Now, just bring up the Vagrant VirtualBox machine. As it comes up, IBM Cloud Private will be configured using the Community Edition available on DockerHub:

```
vagrant up
```

Configuring the IBM Cloud Private Kubernetes Command-Line Interface

The `kubect` command-line interface, which assists with authorization and other product-specific tasks, is available for download from the web console:

```
sudo curl -ko /usr/local/bin/bx-pr https://mycluster.icp:8443/
api/cli/icp-linux-amd64
sudo chmod u+x /usr/local/bin/bx-pr
```

To ensure that you have a compatible version of `kubect` and `Helm`, you can also copy each binary out of the IBM Cloud Private inception container used to configure the cluster:

```
sudo docker cp $(docker ps -qa --latest --filter \
"label=org.label-schema.name=icp
inception-amd64"):usr/local/bin/kubect \
/usr/local/bin/kubect

sudo docker cp $(docker ps -qa --latest --filter \
"label=org.label-schema.name=icp-inception-amd64"): \
```

```
/usr/local/bin/helm
/usr/local/bin/helm
```

To authorize your command-line environment to work with Kubernetes, use `bx-pr` to login and then configure `kubectl` and `Helm`:

```
bx-pr login -a https://mycluster.icp:8443/ \
-u admin --skip-ssl-validation
API endpoint: https://mycluster.icp:8443/
```

```
Password>
Authenticating...
OK
```

```
Select an account:
1. mycluster Account (id-mycluster-account)
Enter a number> 1
Targeted account mycluster Account (id-mycluster-account)
```

```
Configuring helm and kubectl...
Configuring kubectl: /Users/mdelder/.bluemix/plugins/icp\
/clusters
/mycluster/kube-config
Property "clusters.mycluster" unset.
Property "users.mycluster-user" unset.
Property "contexts.mycluster-context" unset.
Cluster "mycluster" set.
User "mycluster-user" set.
Context "mycluster-context" created.
Switched to context "mycluster-context".
```

Cluster mycluster configured successfully.

```
Configuring helm: /Users/mdelder/.helm
Helm configured successfully
```

```
OK
```

Follow the prompts to enter your password and select your cluster. Confirm that you now have access by running a command with `kubectl`, such as the following:

```
kubectl get pods
```

IBM Cloud Kubernetes Service

We recommend referencing the IBM Cloud Kubernetes Service documentation for information on how to get the CLI installed and running quickly. You can find supporting documents at <http://ibm.biz/iks-cli>. After you've completed the configuration, you can

quickly and easily get a Kubernetes configuration file using `ibmcloud ks cluster-config <myclustername>`.

Configuring Your Development Environment

Configuring Java

Java provides a robust, enterprise-grade language for the development of all kinds of applications. To build Java applications from source, you need a Java Software Development Kit (Java SDK). To run Java applications, which are compiled into Java Archives (*.jar, *.war, *.ear), you need a Java Runtime Environment (JRE).

There are many options for Java. We recommend [IBM's Java SDK](#).

Configuring Maven

Apache Maven is a build tool that is very popular for Java applications. You can [download and configure Maven from Apache's website](#).

Configuring Docker

The examples in this book use Docker to create Open Container Initiative (OCI)-compatible images. Docker runs OCI-compliant images and provides an easy-to-use API and tools for working with these images. You can [configure Docker for your platform from Docker's website](#).

Configuring Docker to Push or Pull from an Insecure Registry

The Docker runtime establishes trust of a remote image registry based on the validity of its Transport Layer Security (TLS) certificate. If your cluster uses a self-signed certificate, Docker will consider it “insecure” by default.

You can confirm the allowed insecure registries for your Docker runtime by using the `docker info` command, as demonstrated here:

```
docker info | grep -A 20 "Insecure Registries"
Insecure Registries:
  mycluster.icp:8500
  127.0.0.0/8
Live Restore Enabled: false
```

Configuring the insecure registries for your platform may vary a bit, but the basic flow is to extend the `DOCKER_OPTS` to explicitly list each insecure registry that the Docker runtime is allowed to interact with.

Edit the Docker daemon configuration to add the alias for your IBM Cloud Private cluster, which will be `mycluster.icp:8500`, by default. Depending on your installation and platform, your configuration file might be at `/etc/docker/daemon.json`, `~/.docker/daemon.json`, or `C:\ProgramData\docker\config\daemon.json`.

```
cat ~/.docker/daemon.json
{
  "debug" : true,
```

```
"insecure-registries" : [  
  "mycluster.icp:8500"  
],  
"experimental" : true  
}
```

Then, update your `/etc/hosts` configuration to alias this hostname (provided by the certificate when Docker connects to the endpoint) to the specific public IP of your cluster:

```
cat /etc/hosts | grep mycluster.icp  
1.1.1.1mycluster.icp
```

Restart your Docker runtime to make this change effective.

To find more details for your platform, refer to the [Docker docs](#).

Generating an API Key in Docker Cloud

Images that are managed by the Docker Store require authorization to access. You will need an account to deploy some of the examples used in this book. As of this writing, Docker Cloud has been deprecated, but no equivalent capability to create an API Key exists. So, in the meantime, here is how to create an API key:

1. After subscribing to your image from the **Docker store**, navigate to the **Swarm website**.
2. In the upper-right corner, click your account drop-down, and then select Account Settings.
3. Scroll down to the API Keys section, and then click Add API key.
4. Enter the API key, and then click OK. The API key is displayed.
5. Store your API key in a secure location for reference; *it is displayed only once*.

About the Authors

Michael Elder is an IBM Distinguished Engineer. He provides technical leadership and oversight of IBM Private Cloud Platform with a strong focus on Kubernetes and enterprise requirements.

Jake Kitchener is an IBM Senior Technical Staff Member (STSM) and provides technical leadership for the IBM Cloud Kubernetes Service. His focus is on user experience, scalability, availability, and system architecture.

Dr. Brad Topol is an IBM Distinguished Engineer leading efforts focused on open technologies and developer advocacy. Brad is a Kubernetes contributor, serves as a member of the Kubernetes Conformance Workgroup, and is a Kubernetes documentation maintainer. He received a PhD in Computer Science from the Georgia Institute of Technology in 1998.