Getting Started with Kubernetes

Third Edition

Extend your containerization strategy by orchestrating and managing large-scale container deployments

Packt
www.packt.com

Jonathan Baier and Jesse White

Getting Started with Kubernetes
Third Edition

Extend your containerization strategy by orchestrating and managing large-scale container deployments

Jonathan Baier
Jesse White

BIRMINGHAM - MUMBAI

Getting Started with Kubernetes
Third Edition

Commissioning Editor: Vijin Boricha
Acquisition Editor: Rahul Nair
Content Development Editor: Sharon Raj
Technical Editor: Komal Karne
Copy Editor: Safis Editing
Project Coordinator: Drashti Panchal
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Tom Scaria
Production Coordinator: Shantanu Zagade

First published: December 2015
Second edition: May 2017
Third edition: October 2018

Production reference: 2061118

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78899-472-9

www.packtpub.com

Dedicated to my loving and talented wife, Kaitlyn. Thank you for your support while writing this book, and for all the good work you do in this world.

- Jesse White

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

Packt.com

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

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

Contributors

About the authors

Jonathan Baier is an emerging technology leader living in Brooklyn, New York. He has had a passion for technology since an early age. When he was 14 years old, he was so interested in the family computer (an IBM PCjr) that he pored over the several hundred pages of BASIC and DOS manuals. Then, he taught himself to code a very poorly-written version of Tic-Tac-Toe. During his teenage years, he started a computer support business. Throughout his life, he has dabbled in entrepreneurship. He currently works as Senior Vice President of Cloud Engineering and Operations for Moody's corporation in New York.

I'd like to thank my wonderful wife, Tomoko, and my playful son, Nikko. You both gave me incredible support and motivation during the writing process for both editions of this book. Your smiles move mountains I could not on my own. You are my True North and guiding light in the storm.

I'd also like to thank my co-author, Jesse, for all the hard work in updating and adding new chapters to this edition. You not only made this edition possible, but also took the book to the next level!

Jesse White is a 15-year veteran and technology leader in New York City's very own Silicon Alley, where he is a pillar of the vibrant engineering ecosystem. As founder of DockerNYC and an active participant in the open source community, you can find Jesse at a number of leading industry events, including DockerCon and VelocityConf, giving talks and workshops.

About the reviewer

Jakub Pavlik is a co-founder, former CTO, and chief architect of TCP Cloud (acquired by Mirantis in 2016). Jakub and his team worked for several years on the IaaS cloud platform based on the OpenStack-Salt, Kubernetes, and Open Contrail projects, which they deployed and operated for global service providers. Leveraging his skills in architecture, implementation, and operation, his TCP Cloud team was acquired by #1 pure play OpenStack company Mirantis. Currently a director of engineering, together with other skilled professionals, Jakub builds and operates a new generation of edge-computing platforms at Volterra Inc. He is also an enthusiast of Linux OS, ice hockey (with Pepa), and films, and loves his wife, Hanulka.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

This book is a guide to getting started with Kubernetes and overall container management. We will walk you through the features and functions of Kubernetes and show how it fits into an overall operations strategy. You'll learn what hurdles lurk in moving a container off the developer's laptop and managing them at a larger scale. You'll also see how Kubernetes is the perfect tool to help you face these challenges with confidence.

Who this book is for

Whether you've got your head down in development, you're up to your neck in operations, or you're looking forward as an executive, Kubernetes and this book are for you. *Getting Started with Kubernetes* will help you understand how to move your container applications into production with best practices and step-by-step walkthroughs tied to a real-world operational strategy. You'll learn how Kubernetes fits into your everyday operations, which can help you prepare for production-ready container application stacks.

Having some familiarity with Docker containers, general software development, and operations at a high level will be helpful.

What this book covers

`Chapter 1`, *Introduction to Kubernetes*, is a brief overview of containers and the how, what, and why of Kubernetes orchestration, exploring how it impacts your business goals and everyday operations.

`Chapter 2`, *Building a Foundation with Core Kubernetes Constructs*, uses a few simple examples to explore core Kubernetes constructs, namely pods, services, replication controllers, replica sets, and labels. Basic operations, including health checks and scheduling, will also be covered.

`Chapter 3`, *Working with Networking, Load Balancers, and Ingress*, covers cluster networking for Kubernetes and the Kubernetes proxy. It also takes a deeper dive into services, and shows a brief overview of some higher-level isolation features for multi-tenancy.

`Chapter 4`, *Implementing Reliable, Container-Native Applications*, covers both long-running application deployments and short-lived jobs. We will also look at using DaemonSets to run containers on all or subsets of nodes in the cluster.

`Chapter 5`, *Exploring Kubernetes Storage Concepts*, covers storage concerns and persistent data across pods and the container life cycle. We will also look at new constructs for working with stateful applications in Kubernetes.

`Chapter 6`, *Application Updates, Gradual Rollouts, and Autoscaling*, is a quick look at how to roll out updates and new features with minimal disruption to uptime. We will also look at scaling for applications and the Kubernetes cluster.

`Chapter 7`, *Designing for Continuous Integration and Delivery*, explains how to integrate Kubernetes into your continuous delivery pipeline. We will see how to use a K8s cluster with gulp.js and Jenkins as well.

`Chapter 8`, *Monitoring and Logging*, teaches how to use and customize built-in and third-party monitoring tools on your Kubernetes cluster. We will look at built-in logging and monitoring, the Google Cloud Monitoring/Logging service, and Sysdig.

`Chapter 9`, *Operating Systems, Platforms, and Cloud and Local Providers,* starts off by covering Open Container Project and its mission to provide an open container specification, looking at how having open standards encourages a diverse ecosystem of container implementations (such as Docker, rkt, Kurma, and JetPack). The second half of this chapter will cover available OSes, such as CoreOS, Project Atomic, and their advantages as a host OSes, including performance and support for various container implementations.

`Chapter 10`, *Designing for High Availability and Scalability,* uncovers the Kubernetes Workload capability, which allows us to leverage all App Workload APIs, such as the DaemonSet, Deployment, ReplicaSet, and StatefulSet APIs, in order to create foundations for long-running, stateless, and stateful workloads. We will describe and implement admission control to validate and/or mutate objects within the cluster.

`Chapter 11`, *Kubernetes SIGs, Incubation Projects, and the CNCF,* discusses the new globally distributed collaboration model of Kubernetes and its partner projects. We'll describe the three tiers of organization around SIGs, the different between incubating and graduated projects, and how the CNCF is evolving the idea of an open source project into a distributed foundation.

`Chapter 12`, *Cluster Federation and Multi-Tenancy*, explores the new federation capabilities and how to use them to manage multiple clusters. We will also cover the federated version of the core constructs and the integration to public cloud vendor DNS.

`Chapter 13`, *Cluster Authentication, Authorization, and Container Security*, gets into the options for container security, from the container run-time level to the host itself. We will discuss how to apply these concepts to workloads running in a Kubernetes cluster and some of the security concerns and practices that relate specifically to running your Kubernetes cluster.

`Chapter 14`, *Hardening Kubernetes, and How to Find Out More about Third-Party Extensions and Tools,* covers some of the extensions available from vendors for enterprise-grade deployments. Additionally, we'll look at a brief survey of some of the existing tools and services that work with Kubernetes for monitoring, security, and storage.

`Chapter 15`, *Kubernetes Infrastructure Management,* focuses on how to make changes to the infrastructure that powers your Kubernetes infrastructure, whether it be a purely public cloud platform or a hybrid installation. We'll discuss methods for handling underlying instance and resource instability, and strategies for running highly available workloads on partially available underlying hardware.

To get the most out of this book

This book will cover downloading and running the Kubernetes project. You'll need access to a Linux system (VirtualBox will work if you are on Windows) and some familiarity with the command shell.

Additionally, you should have a Google Cloud Platform account. You can sign up for a free trial here: `https://cloud.google.com/`.

Also, an AWS account is necessary for a few sections of the book. You can sign up for a free trial here: `https://aws.amazon.com/`.

Download the example code files

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

You can download the code files by following these steps:

1. Log in or register at `www.packt.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

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

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

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/9781788994729_ColorImages.pdf`.

Conventions used

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

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The last two main pieces of the Master nodes are `kube-controller-manager` and `cloud-controller-manager`."

A block of code is set as follows:

```
"conditions": [
  {
    "type": "Ready",
    "status": "True"
  }
```

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

```
"conditions": [
  {
    "type": "Ready",
    "status": "True"
  }
```

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

```
$ kubectl describe pods/node-js-pod
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Click on **Jobs** and then **long-task** from the list, so we can see the details."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

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

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

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

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

Reviews

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

For more information about Packt, please visit `packt.com`.

1
Introduction to Kubernetes

In this book, we will help you build, scale, and manage production-ready Kubernetes clusters. Each section of this book will empower you with the core container concepts and the operational context of running modern web services that need to be available 24 hours of the day, 7 days a week, 365 days of the year. As we progress, you'll be given concrete, code-based examples that you can deploy into running clusters in order to get real-world feedback on Kubernetes' many abstractions. By the end of this book, you will have mastered the core conceptual building blocks of Kubernetes, and will have a firm understanding of how to handle the following paradigms:

- Orchestration
- Scheduling
- Networking
- Security
- Storage
- Identity and authentication
- Infrastructure management

This chapter will set the stage for *why Kubernetes?* and give an overview of modern container history, diving into how containers work, as well as why it's important to schedule, orchestrate, and manage a container platform well. We'll tie this back to concrete objectives and goals for your business and product. This chapter will also give a brief overview of how Kubernetes orchestration can enhance our container management strategy and how we can get a basic Kubernetes cluster up, running, and ready for container deployments.

In this chapter, we will cover the following topics:

- Introducing container operations and management
- The importance of container management
- The advantages of Kubernetes

- Downloading the latest Kubernetes
- Installing and starting up a new Kubernetes cluster
- The components of a Kubernetes cluster

Technical requirements

You'll need to have the following tools installed:

- Python
- AWS CLI
- Google Cloud CLI
- Minikube

We'll go into the specifics of these tools' installation and configuration as we go through this chapter. If you already know how to do this, you can go ahead and set them up now.

A brief overview of containers

Believe it or not, containers and their precursors have been around for over 15 years in the Linux and Unix operating systems. If you look deeper into the fundamentals of how containers operate, you can see their roots in the chroot technology that was invented all the way back in 1970. Since the early 2000s, FreeBSD, Linux, Solaris, Open VZ, Warden, and finally Docker all made significant attempts at encapsulating containerization technology for the end user.

While the VServer's project and first commit (*running several general purpose Linux server on a single box with a high degree of independence and security* (`http://ieeexplore.ieee.org/document/1430092/?reload=true`)) may have been one of the most interesting historical junctures in container history, it's clear that Docker set the container ecosystem on fire back in late 2013 when they went full in on the container ecosystem and decided to rebrand from dotCloud to Docker. Their mass marketing of container appeal set the stage for the broad market adoption we see today and is a direct precursor of the massive container orchestration and scheduling platforms we're writing about here.

Over the past five years, containers have grown in popularity like wildfire. Where containers were once relegated to developer laptops, testing, or development environments, you'll now see them as the building blocks of powerful production systems. They're running highly secure banking workloads and trading systems, powering IoT, keeping our on-demand economy humming, and scaling up to millions of containers to keep the products of the 21st century running at peak efficiency in both the cloud and private data centers. Furthermore, containerization technology permeates our technological zeitgest, with every technology conference in the world devoting a significant portion of their talks and sessions devoted to building, running, or developing in containers.

At the beginning of this compelling story lies Docker and their compelling suite of developer-friendly tools. Docker for macOS and Windows, Compose, Swarm, and Registry have been incredibly powerful tools that have shaped workflows and changed how companies develop software. They've built a bridge for containers to exist at the very heart of the **Software Delivery Life Cycle** (**SDLC**), and a remarkable ecosystem has sprung up around those containers. As Malcom McLean revolutionized the physical shipping world in the 1950s by creating a standardized shipping container, which is used today for everything from ice cube trays to automobiles, Linux containers are revolutionizing the software development world by making application environments portable and consistent across the infrastructure landscape.

We'll pick this story up as containers go mainstream, go to production, and go big within organizations. We'll look at what makes a container next.

What is a container?

Containers are a type of operating system virtualization, much like the virtual machines that preceded them. There's also lesser known types of virtualization such as Application Virtualization, Network Virtualization, and Storage Virtualization. While these technologies have been around since the 1960s, Docker's encapsulation of the container paradigm represents a modern implementation of resource isolation that utilizes built-in Linux kernel features such as chroot, **control groups** (**cgroups**), UnionFS, and namespaces to fully isolated resource control at the process level.

Containers use these technologies to create lightweight images that act as a standalone, fully encapsulated piece of software that carries everything it needs inside the box. This can include application binaries, any system tools or libraries, environment-based configuration, and runtime. This special property of isolation is very important, as it allows developers and operators to leverage the all-in-one nature of a container to run without issue, regardless of the environment it's run on. This includes developer laptops and any kind of pre-production or production environment.

This decoupling of application packaging mechanism from the environment on which it runs is a powerful concept that provides a clear separation of concerns between engineering teams. This allows developers to focus on building the core business capabilities into their application code and managing their own dependencies, while operators can streamline the continuous integration, promotion, and deployment of said applications without having to worry about their configuration.

At the core of container technology are three key concepts:

- cgroups
- Namespaces
- Union filesystems

cgroups

cgroups work by allowing the host to share and also limit the resources each process or container can consume. This is important for both resource utilization and security, as it prevents **denial-of-service** (**DoS**) attacks on the host's hardware resources. Several containers can share CPU and memory while staying within the predefined constraints. cgroups allow containers to provision access to memory, disk I/O, network, and CPU. You can also access devices (for example, `/dev/foo`). cgroups also power the soft and hard limits of container constraints that we'll discuss in later chapters.

There are seven major cgroups:

- **Memory cgroup**: This keeps track of page access by the group, and can define limits for physical, kernel, and total memory.
- **Blkio cgroup**: This tracks the I/O usage per group, across the read and write activity per block device. You can throttle by group per device, on operations versus bytes, and for reads versus writes.
- **CPU cgroup**: This keeps track of user and system CPU time and usage per CPU. This allows you to set weights, but not limits.

- **Freezer cgroup**: This is useful in batch management systems that are often stopping and starting tasks in order to schedule resources efficiently. The SIGSTOP signal is used to suspend a process, and the process is generally unaware that it is being suspended (or resumed, for that matter.)
- **CPUset cgroup**: This allows you to pin a group to a specific CPU within a multi-core CPU architecture. You can pin by application, which will prevent it from moving between CPUs. This can improve the performance of your code by increasing the amount of local memory access or minimizing thread switching.
- **Net_cls/net_prio cgroup**: This keeps tabs on the egress traffic class (`net_cls`) or priority (`net_prio`) that is generated by the processes within the cgroup.
- **Devices cgroup**: This controls what read/write permissions the group has on device nodes.

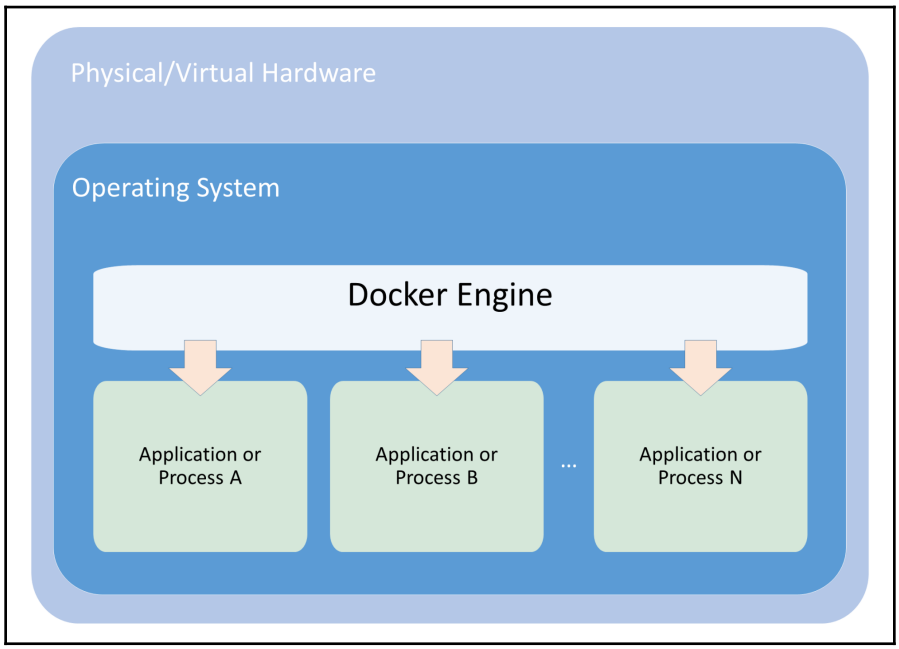
Namespaces

Namespaces offer another form of isolation for process interaction within operating systems, creating the workspace we call a container. Linux namespaces are created via a syscall named `unshare`, while `clone` and `setns` allow you to manipulate namespaces in other manners.

`unshare()` allows a process (or thread) to disassociate parts of its execution context that are currently being shared with other processes (or threads). Part of the execution context, such as the mount namespace, is shared implicitly when a new process is created using FORK(2) (for more information visit `http://man7.org/linux/man-pages/man2/fork.2.html`) or VFORK(2) (for more information visit `http://man7.org/linux/man-pages/man2/vfork.2.html`), while other parts, such as virtual memory, may be shared by explicit request when creating a process or thread using CLONE(2) (for more information visit `http://man7.org/linux/man-pages/man2/clone.2.html`).

Namespaces limit the visibility a process has on other processes, networking, filesystems, and user ID components. Container processes are limited to seeing only what is in the same namespace. Processes from containers or the host processes are not directly accessible from within this container process. Additionally, Docker gives each container its own networking stack that protects the sockets and interfaces in a similar fashion.

If cgroups limit how much of a thing you can use, namespaces limit what things you can see. The following diagram shows the composition of a container:

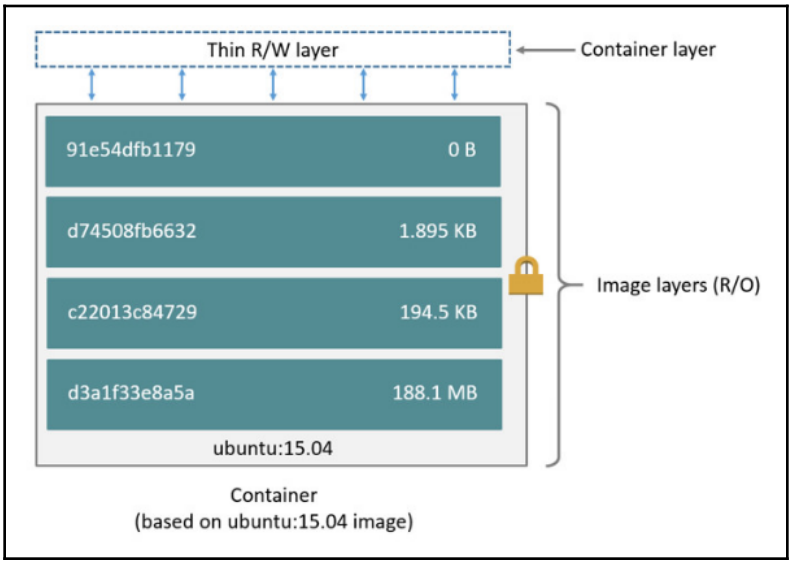


In the case of the Docker engine, the following namespaces are used:

- `pid`: Provides process isolation via an independent set of process IDs from other namespaces. These are nested.
- `net`: Manages network interfaces by virtualizing the network stack through providing a loopback interface, and can create physical and virtual network interfaces that exist in a single namespace at a time.
- `ipc`: Manages access to interprocess communication.
- `mnt`: Controls filesystem mount points. These were the first kind of namespaces created in the Linux kernel, and can be private or shared.
- `uts`: The Unix time-sharing system isolates version IDs and kernel by allowing a single system to provide different host and domain naming schemes to different processes. The processes `gethostname` and `sethostname` use this namespace.
- `user`: This namespace allows you to map UID/GID from container to host, and prevents the need for extra configuration in the container.

Union filesystems

Union filesystems are also a key advantage of using Docker containers. Containers run from an image. Much like an image in the VM or cloud world, it represents state at a particular point in time. Container images snapshot the filesystem, but tend to be much smaller than a VM. The container shares the host kernel and generally runs a much smaller set of processes, so the filesystem and bootstrap period tend to be much smaller—though those constraints are not strictly enforced. Second, the union filesystem allows for the efficient storage, download, and execution of these images. Containers use the idea of *copy-on-write storage*, which is able to create a brand new container immediately, without having to wait on copying out a whole new filesystem. This is similar to thin provisioning in other systems, where storage is allocated as needed:



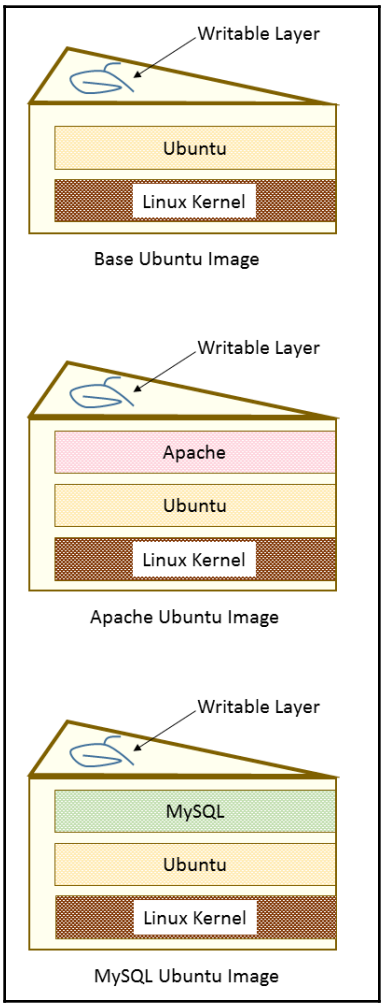
Copy-on-write storage keeps track of what's changed, and in this way is similar to **distributed version control systems** (**DVCS**) such as Git. There are a number of options available to the end user that leverage copy-on-write storage:

- AUFS and overlay at the file level
- Device mapper at the block level
- BTRFS and ZFS and the filesystem level

The easiest way to understand union filesystems is to think of them like a layer cake with each layer baked independently. The Linux kernel is our base layer; then, we might add an OS such as Red Hat Linux or Ubuntu.

Next, we might add an application such as nginx or Apache. Every change creates a new layer. Finally, as you make changes and new layers are added, you'll always have a top layer (think frosting) that is a writable layer. Union filesystems leverage this strategy to make each layer lightweight and speedy.

In Docker's case, the storage driver is responsible for stacking these layers on top of each other and providing a single pane of glass to view these systems. The thin writable layer on the top of this stack of layers is where you'll do your work: the writable container layer. We can consider each layer below to be container image layers:



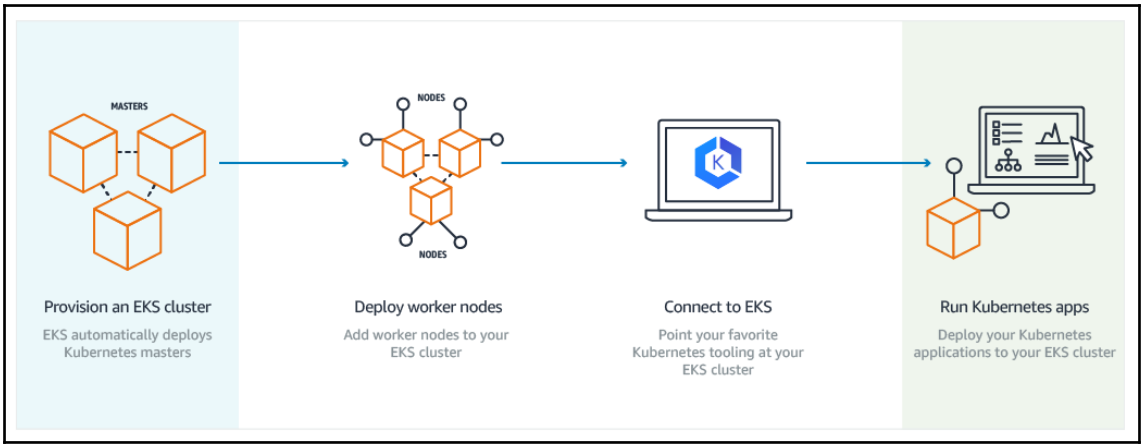
What makes this truly efficient is that Docker caches the layers the first time we build them. So, let's say that we have an image with Ubuntu and then add Apache and build the image. Next, we build MySQL with Ubuntu as the base. The second build will be much faster because the Ubuntu layer is already cached. Essentially, our chocolate and vanilla layers, from the preceding diagram, are already baked. We simply need to bake the pistachio (MySQL) layer, assemble, and add the icing (the writable layer).

Why are containers so cool?

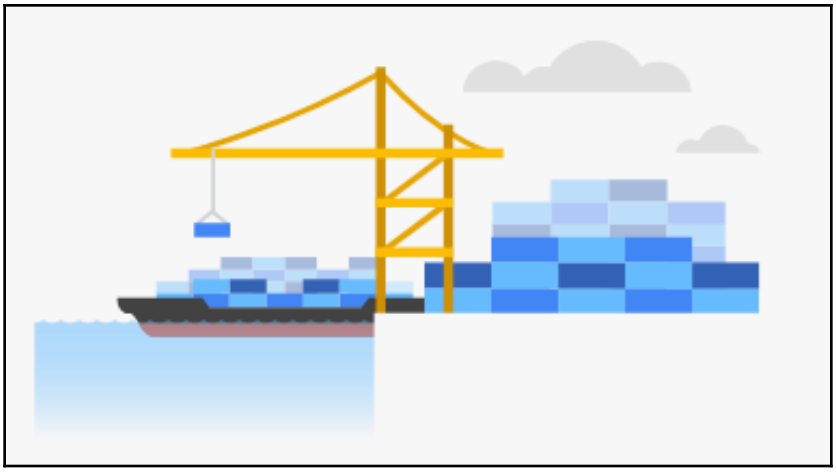
What's also really exciting is that not only has the open source community embraced containers and Kubernetes, but the cloud providers have also deeply embraced the container ecosystem, and invested millions of dollars in supporting tooling, ecosystem, and management planes that can help manage containers. This means you have more options to run container workloads, and you'll have more tools to manage the scheduling and orchestration of the applications running on your clusters.

We'll explore some specific opportunities available to Kubernetes users, but at the time of this book's publishing, all of the major **cloud service providers** (**CSPs**) are offering some form of hosted or managed Kubernetes:

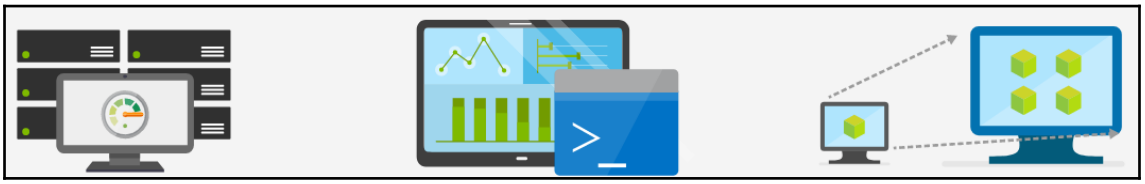
- **Amazon Web Services**: AWS offers **Elastic Container Service for Kubernetes** (**EKS**) (for more information visit `https://aws.amazon.com/eks/`), a managed service that simplifies running Kubernetes clusters in their cloud. You can also roll your own clusters with kops (for information visit `https://kubernetes.io/docs/setup/custom-cloud/kops/`). This product is still in active development:



- **Google Cloud Platform**: GCP offers the **Google Kubernetes Engine** (**GKE**) (for more information visit `https://cloud.google.com/kubernetes-engine/`), a powerful cluster manager that can deploy, manage, and scale containerized applications in the cloud. Google has been running containerized workloads for over 15 years, and this platform is an excellent choice for sophisticated workload management:



- **Microsoft Azure**: Azure offers the **Azure Container Service** (**AKS**) (for more information visit `https://azure.microsoft.com/en-us/services/kubernetes-service/`), which aims to simplify the deployment, management, and operations of a full-scale Kubernetes cluster. This product is still in active development:



When you take advantage of one of these systems, you get built-in management of your Kubernetes cluster, which allows you to focus on the optimization, configuration, and deployment of your cluster.

The advantages of Continuous Integration/Continuous Deployment

ThoughtWorks defines Continuous Integration as a development practice that requires developers to integrate code into a shared repository several times a day. By having a continuous process of building and deploying code, organizations are able to instill quality control and testing as part of the everyday work cycle. The result is that updates and bug fixes happen much faster and the overall quality improves.

However, there has always been a challenge in creating development environments that match those of testing and production. Often, inconsistencies in these environments make it difficult to gain the full advantage of Continuous Delivery. Continuous Integration is the first step in speeding up your organization's software delivery life cycle, which helps you get your software features in front of customer quickly and reliably.

The concept of Continuous Delivery/Deployment uses Continuous Integration to enables developers to have truly portable deployments. Containers that are deployed on a developer's laptop are easily deployed on an in-house staging server. They are then easily transferred to the production server running in the cloud. This is facilitated due to the nature of containers, which build files that specify parent layers, as we discussed previously. One advantage of this is that it becomes very easy to ensure OS, package, and application versions are the same across development, staging, and production environments. Because all the dependencies are packaged into the layer, the same host server can have multiple containers running a variety of OS or package versions. Furthermore, we can have various languages and frameworks on the same host server without the typical dependency clashes we would get in a VM with a single operating system.

This sets the stage for Continuous Delivery/Deployment of the application, as the operations teams or the developers themselves can focus on getting deployments and application rollouts correct, without having to worry about the intricacies of dependencies.

Continuous Delivery is the embodiment and process wherein all code changes are automatically built, tested (Continuous Integration), and then released into production (Continuous Delivery). If this process captures the correct quality gates, security guarantees, and unit/integration/system tests, the development teams will constantly release production-ready and deployable artifacts that have moved through an automated and standardized process.

It's important to note that CD requires the engineering teams to automate more than just unit tests. In order to utilize CD in sophisticated scheduling and orchestration systems such as Kubernetes, teams need to verify application functionality across many dimensions before they're deployed to customers. We'll explore deployment strategies that Kubernetes has to offer in later chapters.

Lastly, it's important to keep in mind that utilizing Kubernetes with CI/CD reduces the risk of the many common problems that technology firms face:

- **Long release cycles**: If it takes a long time to release code to your users, then it's a potential functionality that they're missing out on, and this results in lost revenue. If you have a manual testing or release process, it's going to slow down getting changes to production, and therefore in front of your customers.
- **Fixing code is hard**: When you shorten the release cycle, you're able to discover and remediate bugs closer to the point of creation. This lowers the fixed cost, as there's a correlation between bug introduction and bug discovery times.
- **Release better**: The more you release, the better you get at releasing. Challenging your developers and operators to build automation, monitoring, and logging around the processes of CI/CD will make your pipeline more robust. As you release more often, the amount of difference between releases also decreases. A smaller difference allows teams to troubleshoot potential breaking changes more quickly, which in turn gives them more time to refine the release process further. It's a virtuous cycle!

Because all the dependencies are packaged into the layer, the same host server can have multiple containers running a variety of OS or package versions. Furthermore, we can have various languages and frameworks on the same host server without the typical dependency clashes we would get in a VM with a single operating system.

Resource utilization

The well-defined isolation and layer filesystem also makes containers ideal for running systems with a very small footprint and domain-specific purpose. A streamlined deployment and release process means we can deploy quickly and often. As such, many companies have reduced their deployment time from weeks or months to days and hours in some cases. This development life cycle lends itself extremely well to small, targeted teams working on small chunks of a larger application.

Microservices and orchestration

As we break down an application into very specific domains, we need a uniform way to communicate between all the various pieces and domains. Web services have served this purpose for years, but the added isolation and granular focus that containers bring have paved the way for microservices.

A definition for microservices can be a bit nebulous, but a definition from Martin Fowler, a respected author and speaker on software development, says this:

> *In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.*

As the pivot to containerization and as microservices evolve in an organization, they will soon need a strategy to maintain many containers and microservices. Some organizations will have hundreds or even thousands of containers running in the years ahead.

Future challenges

Life cycle processes alone are an important piece of operation and management. How will we automatically recover when a container fails? Which upstream services are affected by such an outage? How will we patch our applications with minimal downtime? How will we scale up our containers and services as our traffic grows?

Networking and processing are also important concerns. Some processes are part of the same service and may benefit from proximity to the network. Databases, for example, may send large amounts of data to a particular microservice for processing. How will we place containers near each other in our cluster? Is there common data that needs to be accessed? How will new services be discovered and made available to other systems?

Resource utilization is also key. The small footprint of containers means that we can optimize our infrastructure for greater utilization. Extending the savings started in the Elastic cloud will take us even further toward minimizing wasted hardware. How will we schedule workloads most efficiently? How will we ensure that our important applications always have the right resources? How can we run less important workloads on spare capacity?

Finally, portability is a key factor in moving many organizations to containerization. Docker makes it very easy to deploy a standard container across various operating systems, cloud providers, and on-premise hardware or even developer laptops. However, we still need tooling to move containers around. How will we move containers between different nodes on our cluster? How will we roll out updates with minimal disruption? What process do we use to perform blue-green deployments or canary releases?

Whether you are starting to build out individual microservices and separating concerns into isolated containers or you simply want to take full advantage of the portability and immutability in your application development, the need for management and orchestration becomes clear. This is where orchestration tools such as Kubernetes offer the biggest value.

Our first clusters

Kubernetes is supported on a variety of platforms and OSes. For the examples in this book, I used an Ubuntu 16.04 Linux VirtualBox (`https://www.virtualbox.org/wiki/Downloads`) for my client and **Google Compute Engine** (**GCE**) with Debian for the cluster itself. We will also take a brief look at a cluster running on **Amazon Web Services** (**AWS**) with Ubuntu.

To save some money, both GCP (`https://cloud.google.com/free/`) and AWS (`https://aws.amazon.com/free/`) offer free tiers and trial offers for their cloud infrastructure. It's worth using these free trials for learning Kubernetes, if possible.

Most of the concepts and examples in this book should work on any installation of a Kubernetes cluster. To get more information on other platform setups, refer to the Kubernetes getting started page, which will help you pick the right solution for your cluster: `http://kubernetes.io/docs/getting-started-guides/`.

Running Kubernetes on GCE

We have a few options for setting up the prerequisites for our development environment. While we'll use a Linux client on our local machine in this example, you can also use the Google Cloud Shell to simplify your dependencies and setup. You can check out that documentation at `https://cloud.google.com/shell/docs/`, and then jump down to the `gcloud auth login` portion of the tutorial.

Getting back to the local installation, let's make sure that our environment is properly set up before we install Kubernetes. Start by updating the packages:

```
$ sudo apt-get update
```

You should see something similar to the following output:

```
$ sudo apt update
[sudo] password for user:
Hit:1 http://archive.canonical.com/ubuntu xenial InRelease
Ign:2 http://dl.google.com/linux/chrome/deb stable InRelease
Hit:3 http://archive.ubuntu.com/ubuntu xenial InRelease
Get:4 http://security.ubuntu.com/ubuntu xenial-security InRelease [102 kB]
Ign:5 http://dell.archive.canonical.com/updates xenial-dell-dino2-mlk
InRelease
Hit:6 http://ppa.launchpad.net/webupd8team/sublime-text-3/ubuntu xenial
InRelease
Hit:7 https://download.sublimetext.com apt/stable/ InRelease
Hit:8 http://dl.google.com/linux/chrome/deb stable Release
Get:9 http://archive.ubuntu.com/ubuntu xenial-updates InRelease [102 kB]
Hit:10 https://apt.dockerproject.org/repo ubuntu-xenial InRelease
Hit:11 https://deb.nodesource.com/node_7.x xenial InRelease
Hit:12 https://download.docker.com/linux/ubuntu xenial InRelease
Ign:13 http://dell.archive.canonical.com/updates xenial-dell InRelease
<SNIPPED...>
Fetched 1,593 kB in 1s (1,081 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
120 packages can be upgraded. Run 'apt list --upgradable' to see them.
$
```

Install Python and `curl` if they are not present:

```
$ sudo apt-get install python
$ sudo apt-get install curl
```

Install the `gcloud` SDK:

```
$ curl https://sdk.cloud.google.com | bash
```

We will need to start a new shell before `gcloud` is on our path.

Configure your GCP account information. This should automatically open a browser, from where we can log in to our Google Cloud account and authorize the SDK:

```
$ gcloud auth login
```

If you have problems with login or want to use another browser, you can optionally use the `--no-launch-browser` command. Copy and paste the URL to the machine and/or browser of your choice. Log in with your Google Cloud credentials and click **Allow** on the permissions page. Finally, you should receive an authorization code that you can copy and paste back into the shell, where the prompt will be waiting.

A default project should be set, but we can verify this with the following command:

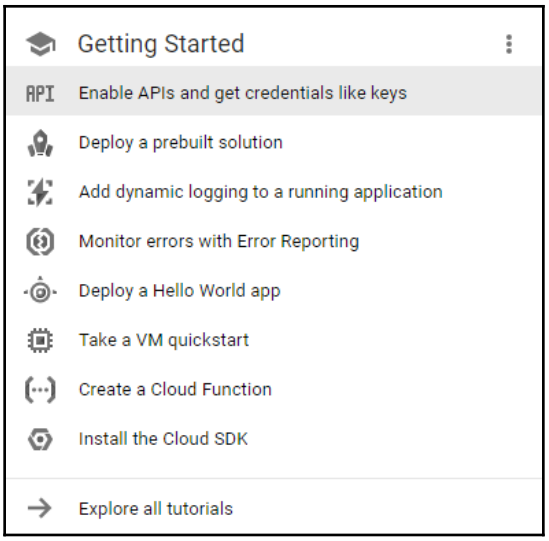
```
$ gcloud config list project
```

We can modify this and set a new default project with the following command. Make sure to use project ID and not project name, as follows:

```
$ gcloud config set project <PROJECT ID>
```

We can find our project ID in the console at the following URL: `https://console.developers.google.com/project`. Alternatively, we can list the active projects with `$ gcloud alpha projects list`.

You can turn on API access to your project at this point in the GCP dashboard, `https://console.developers.google.com/project`, or the Kubernetes script will prompt you to do so in the next section:



Next, you want to change to a directory when you can install the Kubernetes binaries. We'll set that up and then download the software:

```
$ mkdir ~/code/gsw-k8s-3
$ cd ~/code/gsw-k8s-3
```

Installing the latest Kubernetes version is done in a single step, as follows:

```
$ curl -sS https://get.k8s.io | bash
```

It may take a minute or two to download Kubernetes depending on your connection speed. Earlier versions would automatically call the `kube-up.sh` script and start building our cluster. In version 1.5, we will need to call the `kube-up.sh` script ourselves to launch the cluster. By default, it will use the Google Cloud and GCE:

```
$ kubernetes/cluster/kube-up.sh
```

If you get an error at this point due to missing components, you'll need to add a few pieces to your local Linux box. If you're running the Google Cloud Shell, or are utilizing a VM in GCP, you probably won't see this error:

```
$ kubernetes_install cluster/kube-up.sh...
Starting cluster in us-central1-b using provider gce
... calling verify-prereqs
missing required gcloud component "alpha"
missing required gcloud component "beta"
$
```

You can see that these components are missing and are required for leveraging the `kube-up.sh script`:

```
$ gcloud components list
Your current Cloud SDK version is: 193.0.0
The latest available version is: 193.0.0
┌──────────────────────────────────────────────────────────────────────────────
──────────────────────────────────────────────────────────────────────────────
───────────────────────────────────┐
│ Components │
├───────────────────────────┬──────────────────────────────────────────────────
─────────────────────────────────────────────┬────────────────────────────────
──────────────┬────────────────────┤
│ Status │ Name │ ID │ Size │
├───────────────────────────┼──────────────────────────────────────────────────
─────────────────────────────────────────────┼────────────────────────────────
──────────────┼────────────────────┤
│ Not Installed │ App Engine Go Extensions │ app-engine-go │ 151.9 MiB │
│ Not Installed │ Cloud Bigtable Command Line Tool │ cbt │ 4.5 MiB │
│ Not Installed │ Cloud Bigtable Emulator │ bigtable │ 3.7 MiB │
│ Not Installed │ Cloud Datalab Command Line Tool │ datalab │ < 1 MiB │
│ Not Installed │ Cloud Datastore Emulator │ cloud-datastore-emulator │
17.9 MiB │
│ Not Installed │ Cloud Datastore Emulator (Legacy) │ gcd-emulator │ 38.1
MiB │
│ Not Installed │ Cloud Pub/Sub Emulator │ pubsub-emulator │ 33.4 MiB │
│ Not Installed │ Emulator Reverse Proxy │ emulator-reverse-proxy │ 14.5
MiB │
│ Not Installed │ Google Container Local Builder │ container-builder-
local │ 3.8 MiB │
│ Not Installed │ Google Container Registry's Docker credential helper │
docker-credential-gcr │ 3.3 MiB │
│ Not Installed │ gcloud Alpha Commands │ alpha │ < 1 MiB │
│ Not Installed │ gcloud Beta Commands │ beta │ < 1 MiB │
│ Not Installed │ gcloud app Java Extensions │ app-engine-java │ 118.9
MiB │
│ Not Installed │ gcloud app PHP Extensions │ app-engine-php │  │
│ Not Installed │ gcloud app Python Extensions │ app-engine-python │ 6.2
MiB │
│ Not Installed │ gcloud app Python Extensions (Extra Libraries) │ app-
engine-python-extras │ 27.8 MiB │
│ Not Installed │ kubectl │ kubectl │ 12.3 MiB │
│ Installed │ BigQuery Command Line Tool │ bq │ < 1 MiB │
│ Installed │ Cloud SDK Core Libraries │ core │ 7.3 MiB │
│ Installed │ Cloud Storage Command Line Tool │ gsutil │ 3.3 MiB │
└───────────────────────────┴──────────────────────────────────────────────────
─────────────────────────────────────────────┴────────────────────────────────
──────────────┴────────────────────┘
```

```
To install or remove components at your current SDK version [193.0.0], run:
  $ gcloud components install COMPONENT_ID
  $ gcloud components remove COMPONENT_ID
To update your SDK installation to the latest version [193.0.0], run:
  $ gcloud components update
```

You can update the components by adding them to your shell:

```
$ gcloud components install alpha beta
Your current Cloud SDK version is: 193.0.0
Installing components from version: 193.0.0
┌──────────────────────────────────────────────────────────────────────────────
──┐
│ These components will be installed. │
├──────────────────────────────────────────────────┬──────────────────┬────────
──┤
│ Name │ Version │ Size │
├──────────────────────────────────────────────────┼──────────────────┼────────
──┤
│ gcloud Alpha Commands │ 2017.09.15 │ < 1 MiB │
│ gcloud Beta Commands │ 2017.09.15 │ < 1 MiB │
└──────────────────────────────────────────────────┴──────────────────┴────────
──┘
For the latest full release notes, please visit:
  https://cloud.google.com/sdk/release_notes
Do you want to continue (Y/n)? y
╔══════════════════════════════════════════════════════════════════════════════
══════════════════════════╗
╠═══ Creating update staging area ═══╣
╠══════════════════════════════════════════════════════════════════════════════
══════════════════════════╣
╠═══ Installing: gcloud Alpha Commands ═══╣
╠══════════════════════════════════════════════════════════════════════════════
══════════════════════════╣
╠═══ Installing: gcloud Beta Commands ═══╣
╠══════════════════════════════════════════════════════════════════════════════
══════════════════════════╣
╠═══ Creating backup and activating new installation ═══╣
╚══════════════════════════════════════════════════════════════════════════════
══════════════════════════╝
Performing post processing steps...done.
Update done!
```

After you run the `kube-up.sh` script, you will see quite a few lines roll past. Let's take a look at them one section at a time:

```
... Starting cluster in us-central1-b using provider gce
... calling verify-prereqs

All components are up to date.

All components are up to date.

All components are up to date.
```

If your `gcloud` components are not up to date, you may be prompted to update them.

The preceding screenshot shows the checks for prerequisites, as well as making sure that all components are up to date. This is specific to each provider. In the case of GCE, it will verify that the SDK is installed and that all components are up to date. If not, you will see a prompt at this point to install or update:

```
... calling kube-up
Your active configuration is: [default]

Project: dynamic-nomad-152102
Zone: us-central1-b
gs://kubernetes-staging-549d6b8d9c/kubernetes-devel/
+++ Staging server tars to Google Storage: gs://kubernetes-staging-549d6b8d9c/kub
ernetes-devel
+++ kubernetes-server-linux-amd64.tar.gz uploaded (sha1 = 5df19e3745bbc8c7d1a5bf6
d61d9e1b0d189db64)
+++ kubernetes-salt.tar.gz uploaded (sha1 = 95e855d893e4549b935aed8736f3a2372ae7c
cd3)
+++ kubernetes-manifests.tar.gz uploaded (sha1 = e9c52530a14612c91f45e017743925a0
dba6dcc8)
INSTANCE_GROUPS=
NODE_NAMES=
```

Now, the script is turning up the cluster. Again, this is specific to the provider. For GCE, it first checks to make sure that the SDK is configured for a default project and zone. If they are set, you'll see those in the output:

You may see an output that the bucket for storage hasn't been created. That's normal! The creation script will go ahead and create it.

```
BucketNotFoundException: 404 gs://kubernetes-staging-22caacf417 bucket does
not exist.
```

Next, it uploads the server binaries to Google Cloud storage, as seen in the **Creating gs:...** lines:

```
Looking for already existing resources
Starting master and configuring firewalls
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/zon
es/us-central1-b/disks/kubernetes-master-pd].
NAME                 ZONE           SIZE_GB  TYPE    STATUS
kubernetes-master-pd  us-central1-b  20       pd-ssd  READY

New disks are unformatted. You must format and mount a disk before it
can be used. You can find instructions on how to do this at:

https://cloud.google.com/compute/docs/disks/add-persistent-disk#formatting

Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/glo
bal/firewalls/kubernetes-master-https].
NAME                    NETWORK  SRC_RANGES  RULES    SRC_TAGS  TARGET_TAGS
kubernetes-master-https  default  0.0.0.0/0   tcp:443            kubernetes-mast
er
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/reg
ions/us-central1/addresses/kubernetes-master-ip].
Generating certs for alternate-names: IP:23.251.158.223,IP:10.0.0.1,DNS:kubernet
es,DNS:kubernetes.default,DNS:kubernetes.default.svc,DNS:kubernetes.default.svc.
cluster.local,DNS:kubernetes-master
```

It then checks for any pieces of a cluster already running. Then, we finally start creating the cluster. In the output in the preceding screenshot, we can see it creating the master server, IP address, and appropriate firewall configurations for the cluster:

```
+++ Logging using Fluentd to gcp
WARNING: You have selected a disk size of under [200GB]. This may result in poor
 I/O performance. For more information, see: https://developers.google.com/compu
te/docs/disks#pdperformance.
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/glo
bal/firewalls/kubernetes-minion-all].
NAME                    NETWORK  SRC_RANGES     RULES                     SRC_TAG
S  TARGET_TAGS
kubernetes-minion-all  default  10.244.0.0/14  tcp,udp,icmp,esp,ah,sctp
   kubernetes-minion
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/zon
es/us-central1-b/instances/kubernetes-master].
NAME              ZONE           MACHINE_TYPE   PREEMPTIBLE  INTERNAL_IP  EXTER
NAL_IP     STATUS
kubernetes-master  us-central1-b  n1-standard-1               10.128.0.2   23.25
1.158.223  RUNNING
Creating minions.
Attempt 1 to create kubernetes-minion-template
WARNING: You have selected a disk size of under [200GB]. This may result in poor
 I/O performance. For more information, see: https://developers.google.com/compu
te/docs/disks#pdperformance.
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/glo
bal/instanceTemplates/kubernetes-minion-template].
NAME                         MACHINE_TYPE   PREEMPTIBLE  CREATION_TIMESTAMP
kubernetes-minion-template  n1-standard-2               2016-12-10T04:25:37.527-
08:00
Created [https://www.googleapis.com/compute/v1/projects/dynamic-nomad-152102/zon
es/us-central1-b/instanceGroupManagers/kubernetes-minion-group].
NAME                    LOCATION       SCOPE  BASE_INSTANCE_NAME       SIZE  TA
RGET_SIZE  INSTANCE_TEMPLATE           AUTOSCALED
kubernetes-minion-group  us-central1-b  zone   kubernetes-minion-group  0     3
           kubernetes-minion-template  no
Waiting for group to become stable, current operations: creating: 3
Waiting for group to become stable, current operations: creating: 3
Waiting for group to become stable, current operations: creating: 1
Group is stable
```

Finally, it creates the minions or nodes for our cluster. This is where our container workloads will actually run. It will continually loop and wait while all the minions start up. By default, the cluster will have four nodes (minions), but K8s supports having more than 1,000 (and soon beyond). We will come back to scaling the nodes later on in this book:

```
Attempt 1 to create kubernetes-minion-template
WARNING: You have selected a disk size of under [200GB]. This may result in
poor I/O performance. For more information, see:
https://developers.google.com/compute/docs/disks#performance.
Created
[https://www.googleapis.com/compute/v1/projects/gsw-k8s-3/global/instanceTe
mplates/kubernetes-minion-template].
```

```
NAME MACHINE_TYPE PREEMPTIBLE CREATION_TIMESTAMP
kubernetes-minion-template n1-standard-2 2018-03-17T11:14:04.186-07:00
Created
[https://www.googleapis.com/compute/v1/projects/gsw-k8s-3/zones/us-central1
-b/instanceGroupManagers/kubernetes-minion-group].
NAME LOCATION SCOPE BASE_INSTANCE_NAME SIZE TARGET_SIZE INSTANCE_TEMPLATE
AUTOSCALED
kubernetes-minion-group us-central1-b zone kubernetes-minion-group 0 3
kubernetes-minion-template no
Waiting for group to become stable, current operations: creating: 3
Group is stable
INSTANCE_GROUPS=kubernetes-minion-group
NODE_NAMES=kubernetes-minion-group-176g kubernetes-minion-group-s9qw
kubernetes-minion-group-tr7r
Trying to find master named 'kubernetes-master'
Looking for address 'kubernetes-master-ip'
Using master: kubernetes-master (external IP: 104.155.172.179)
Waiting up to 300 seconds for cluster initialization.
```

Now that everything is created, the cluster is initialized and started. Assuming that everything goes well, we will get an IP address for the master:

```
... calling validate-cluster
Validating gce cluster, MULTIZONE=
Project: gsw-k8s-3
Network Project: gsw-k8s-3
Zone: us-central1-b
No resources found.
Waiting for 4 ready nodes. 0 ready nodes, 0 registered. Retrying.
No resources found.
Waiting for 4 ready nodes. 0 ready nodes, 0 registered. Retrying.
Waiting for 4 ready nodes. 0 ready nodes, 1 registered. Retrying.
Waiting for 4 ready nodes. 0 ready nodes, 4 registered. Retrying.
Found 4 node(s).
NAME STATUS ROLES AGE VERSION
kubernetes-master Ready,SchedulingDisabled <none> 32s v1.9.4
kubernetes-minion-group-176g Ready <none> 25s v1.9.4
kubernetes-minion-group-s9qw Ready <none> 25s v1.9.4
kubernetes-minion-group-tr7r Ready <none> 35s v1.9.4
Validate output:
NAME STATUS MESSAGE ERROR
etcd-1 Healthy {"health": "true"}
scheduler Healthy ok
controller-manager Healthy ok
etcd-0 Healthy {"health": "true"}
Cluster validation succeeded
```

Also, note that configuration along with the cluster management credentials are stored in `home/<Username>/.kube/config`.

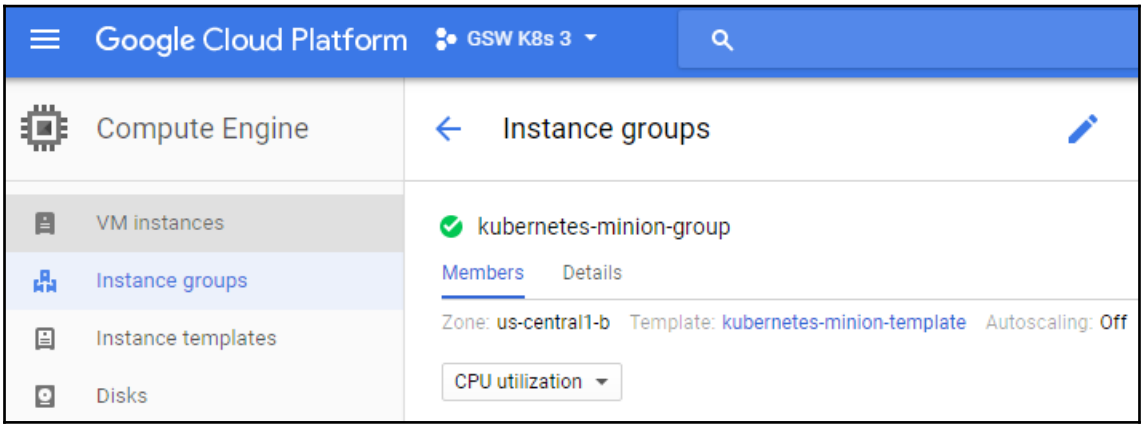
Then, the script will validate the cluster. At this point, we are no longer running provider-specific code. The validation script will query the cluster via the `kubectl.sh` script. This is the central script for managing our cluster. In this case, it checks the number of minions found, registered, and in a ready state. It loops through, giving the cluster up to 10 minutes to finish initialization.

After a successful startup, a summary of the minions and the cluster component health is printed on the screen:

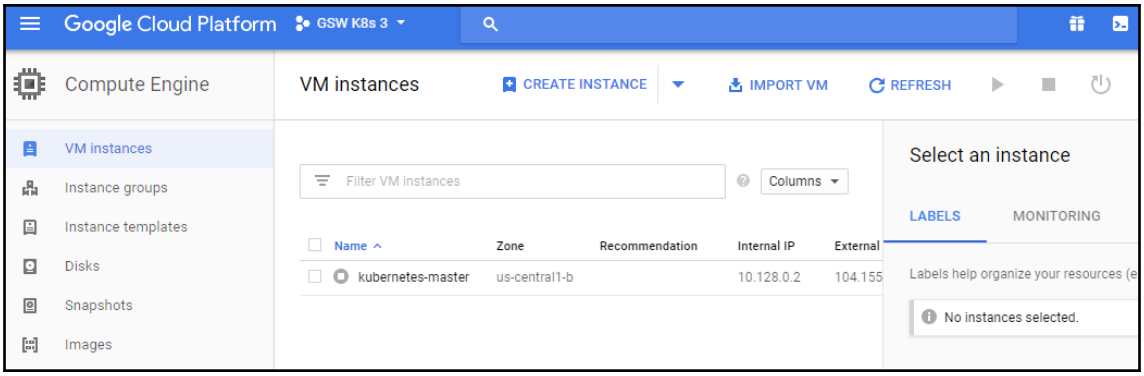
```
Done, listing cluster services:
Kubernetes master is running at https://104.155.172.179
GLBCDefaultBackend is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/default-http
-backend:http/proxy
Heapster is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/heapster/pro
xy
KubeDNS is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/kube-dns:dns
/proxy
kubernetes-dashboard is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/https:kubern
etes-dashboard:/proxy
Metrics-server is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/https:metric
s-server:/proxy
Grafana is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/monitoring-g
rafana/proxy
InfluxDB is running at
https://104.155.172.179/api/v1/namespaces/kube-system/services/monitoring-i
nfluxdb:http/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info
dump'.
```

Finally, a `kubectl cluster-info` command is run, which outputs the URL for the master services, including DNS, UI, and monitoring. Let's take a look at some of these components.

If you'd like to get further debugging and/or diagnose cluster problems, you can use `kubectl cluster-info dump` to see what's going on with your cluster. Additionally, if you need to pause and take a break and want to conserve your free hours, you can log into the GUI and set the `kubernetes-minion-group` instance group to zero, which will remove all of the instances. The pencil will edit the group for you; set it to zero. Don't forget to set it back to three if you want to pick up again!



You can simply stop the manager as well. You'll need to click the stop button to shut it down:



If you'd like to start the cluster up again, start the servers again to keep going. They'll need some time to start up and connect to each other.

If you want to work on more than one cluster at a time or you want to use a different name than the default, see the `<kubernetes>/cluster/gce/config-default.sh` file for more fine-grained configuration of your cluster.

Kubernetes UI

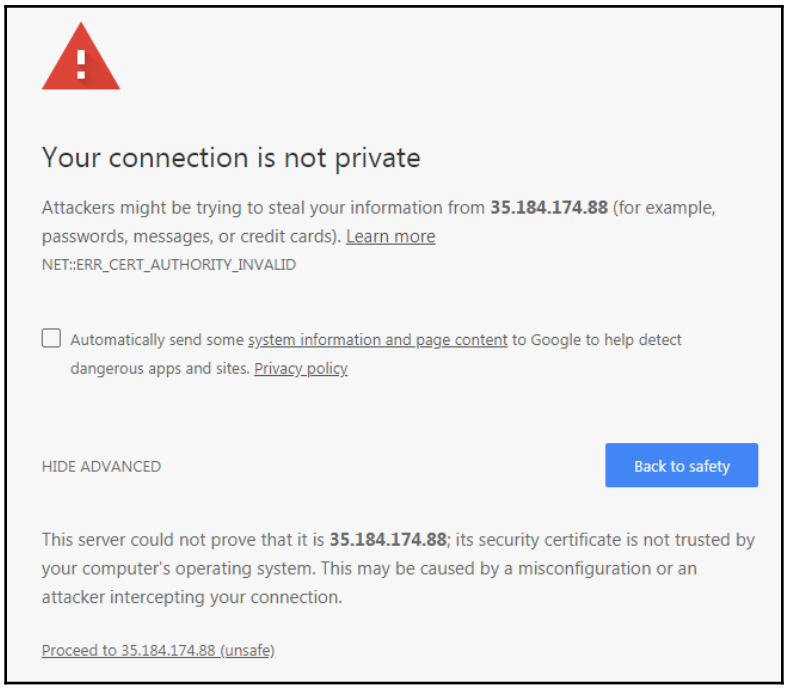
Since Kubernetes v1.3.x, you can no longer authenticate through public IP addresses to the GUI. To get around this, we'll use the `kubectl proxy` command. First, grab the token from the configuration command, and then we'll use it to launch a local proxy version of the UI:

```
$ kubectl config view |grep token
 token: RvoYTIn4rExi1bNRzk56g0PU0srZbzOf
$ kubectl proxy --port=8001
```

Open a browser and enter the following URL: `https://localhost/ui/`.

You can also type these commands to open a browser window automatically if you're on macOS: `$ open https://localhost/ui/` or `$ xdg-open https://localhost/ui` if you're on Linux.

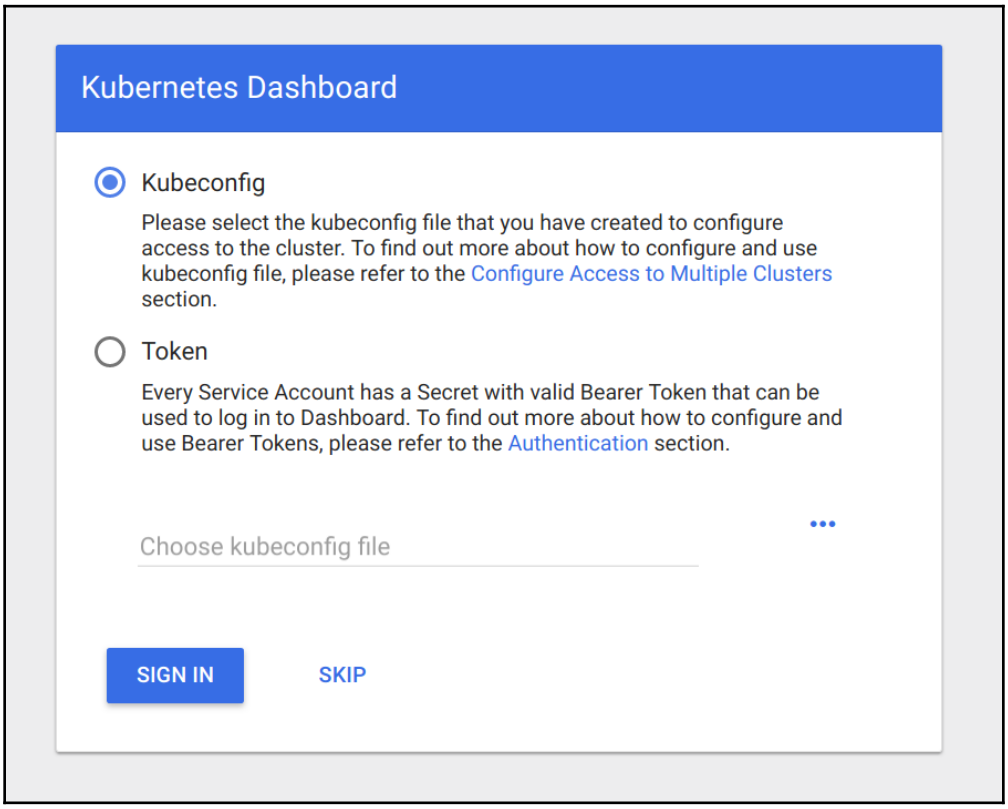
The certificate is self-signed by default, so you'll need to ignore the warnings in your browser before proceeding. After this, we will see a login dialog:



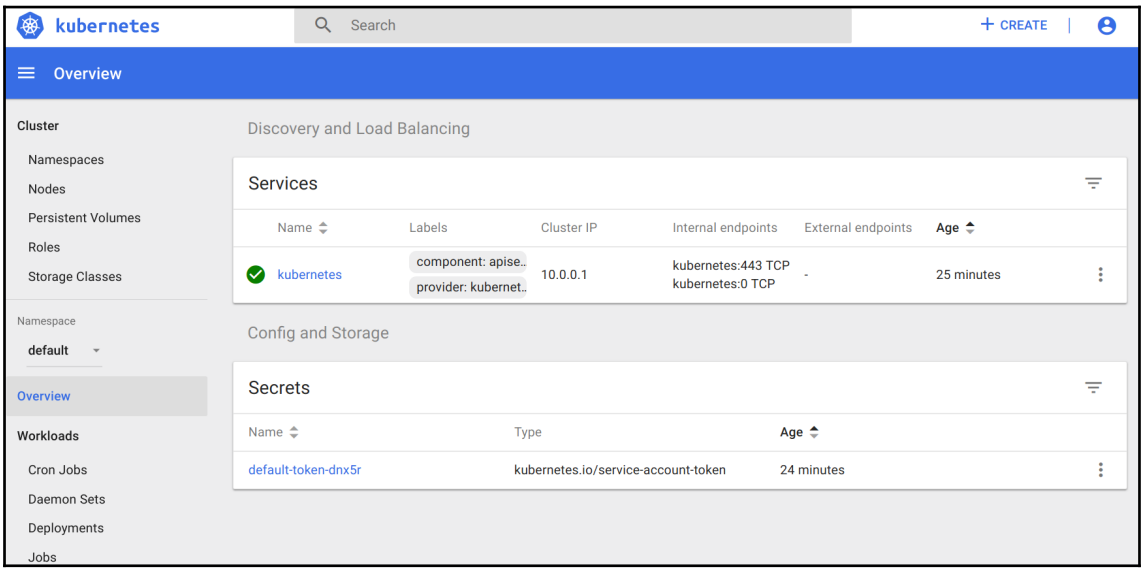
At this login dialog, you'll need to input the token that you grabbed in the aforementioned command.

This is where we use the credentials listed during the K8s installation. We can find them at any time by simply using the `config` command `$ kubectl config view`.

Use the **Token** option and log in to your cluster:

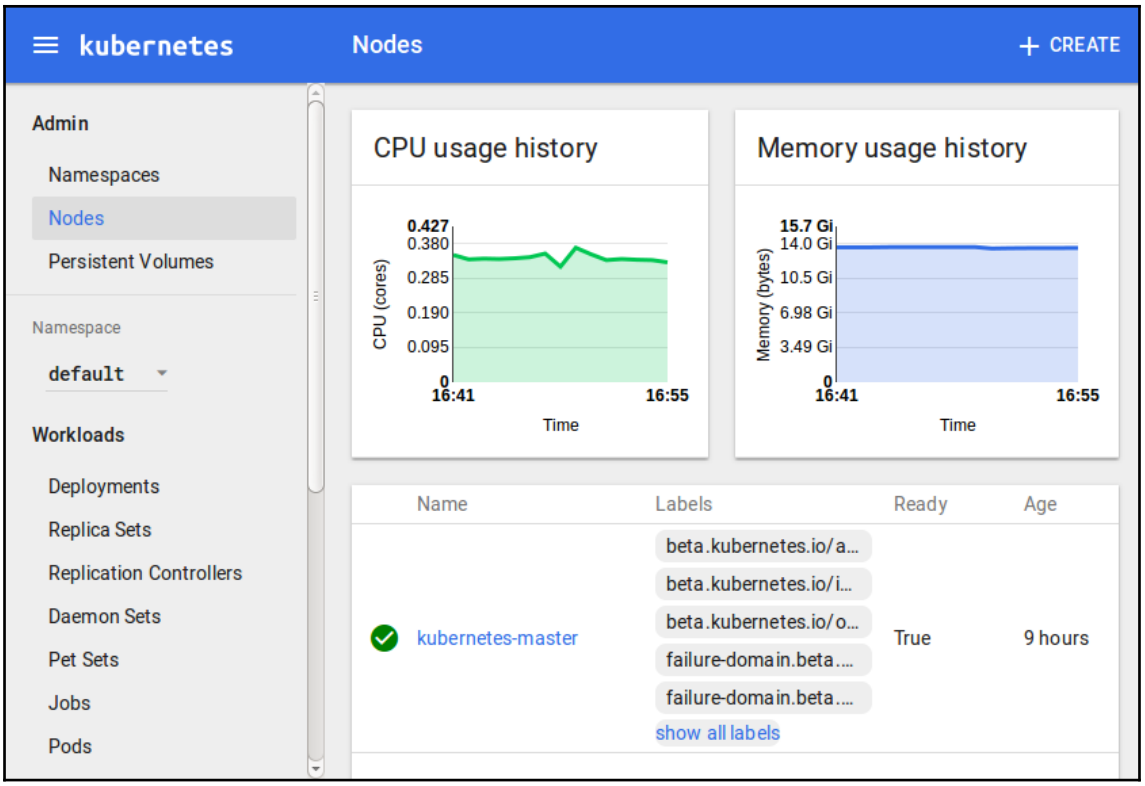


Now that we have entered our token, you should see a dashboard like the one in the following screenshot:



The main dashboard takes us to a page with not much display at first. There is a link to deploy a containerized app that will take you to a GUI for deployment. This GUI can be a very easy way to get started deploying apps without worrying about the YAML syntax for Kubernetes. However, as your use of containers matures, it's a good practice to use the YAML definitions that are checked in to source control.

If you click on the **Nodes** link on the left-hand side menu, you will see some metrics on the current cluster nodes:



At the top, we can see an aggregate of the CPU and memory use followed by a listing of our cluster nodes. Clicking on one of the nodes will take us to a page with detailed information about that node, its health, and various metrics.

The Kubernetes UI has a lot of other views that will become more useful as we start launching real applications and adding configurations to the cluster.

Grafana

Another service installed by default is Grafana. This tool will give us a dashboard to view metrics on the cluster nodes. We can access it using the following syntax in a browser:

```
https://localhost/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana
```

The Grafana dashboard should look like this:

From the main page, click on the **Home** drop-down and select **Cluster**. Here, Kubernetes is actually running a number of services. Heapster is used to collect the resource usage on the pods and nodes, and stores the information in InfluxDB. The results, such as CPU and memory usage, are what we see in the Grafana UI. We will explore this in depth in `Chapter 8`, *Monitoring and Logging*.

Command line

The `kubectl` script has commands for exploring our cluster and the workloads running on it. You can find it in the `/kubernetes/client/bin` folder. We will be using this command throughout the book, so let's take a second to set up our environment. We can do so by putting the binaries folder on our `PATH`, in the following manner:

```
$ export PATH=$PATH:/<Path where you downloaded K8s>/kubernetes/client/bin
$ chmod +x /<Path where you downloaded K8s>/kubernetes/client/bin
```

You may choose to download the `kubernetes` folder outside your home folder, so modify the preceding command as appropriate. It is also a good idea to make the changes permanent by adding the `export` command to the end of your `.bashrc` file in your home directory.

Now that we have `kubectl` on our path, we can start working with it. It has quite a few commands. Since we have not spun up any applications yet, most of these commands will not be very interesting. However, we can explore two commands right away.

First, we have already seen the `cluster-info` command during initialization, but we can run it again at any time with the following command:

```
$ kubectl cluster-info
```

Another useful command is `get`. It can be used to see currently running services, pods, replication controllers, and a lot more. Here are the three examples that are useful right out of the gate:

- Lists the nodes in our cluster:

```
$ kubectl get nodes
```

- Lists cluster events:

```
$ kubectl get events
```

- Finally, we can see any services that are running in the cluster, as follows:

```
$ kubectl get services
```

To start with, we will only see one service, named `kubernetes`. This service is the core API server for the cluster.

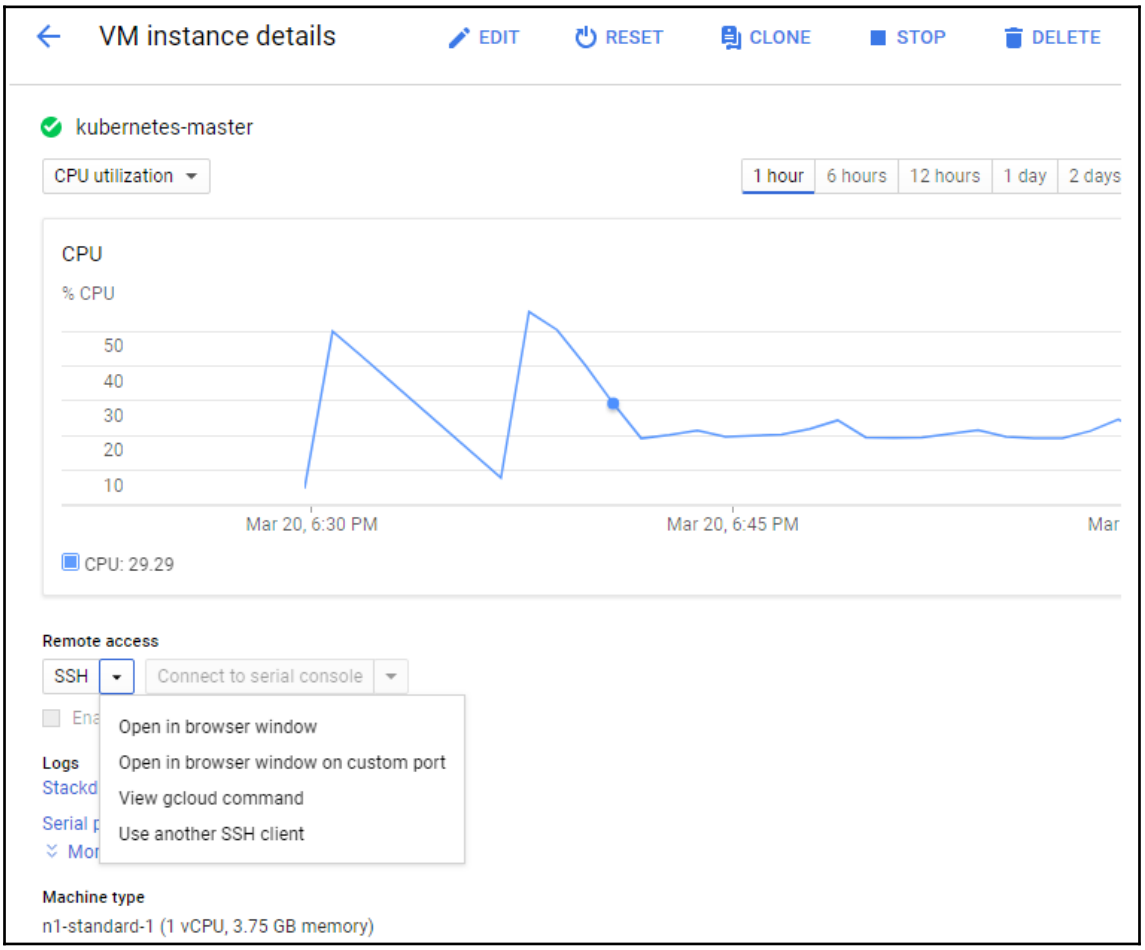
For any of the preceding commands, you can always add a `-h` flag on the end to understand the intended usage.

Services running on the master

Let's dig a little bit deeper into our new cluster and its core services. By default, machines are named with the `kubernetes-` prefix. We can modify this using `$KUBE_GCE_INSTANCE_PREFIX` before a cluster is spun up. For the cluster we just started, the master should be named `kubernetes-master`. We can use the `gcloud` command-line utility to SSH into the machine. The following command will start an SSH session with the master node. Be sure to substitute your project ID and zone to match your environment:

```
$ gcloud compute ssh --zone "<your gce zone>" "kubernetes-master"

$ gcloud compute ssh --zone "us-central1-b" "kubernetes-master"
Warning: Permanently added 'compute.5419404412212490753' (RSA) to the list
of known hosts.

Welcome to Kubernetes v1.9.4!

You can find documentation for Kubernetes at:
  http://docs.kubernetes.io/

The source for this release can be found at:
  /home/kubernetes/kubernetes-src.tar.gz
Or you can download it at:
https://storage.googleapis.com/kubernetes-release/release/v1.9.4/kubernetes
-src.tar.gz

It is based on the Kubernetes source at:
  https://github.com/kubernetes/kubernetes/tree/v1.9.4

For Kubernetes copyright and licensing information, see:
  /home/kubernetes/LICENSES

jesse@kubernetes-master ~ $
```

If you have trouble with SSH via the Google Cloud CLI, you can use the console, which has a built-in SSH client. Simply go to the **VM instances details** page and you'll see an **SSH** option as a column in the `kubernetes-master` listing. Alternatively, the VM instance details page has the SSH option at the top.



Once we are logged in, we should get a standard shell prompt. Let's run the `docker` command that filters for `Image` and `Status`:

```
$ docker container ls --format 'table {{.Image}}\t{{.Status}}'
```

```
IMAGE                                                                     STATUS
gcr.io/google_containers/node-problem-detector:v0.1                       Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/fluentd-gcp:1.21                                 Up 13 hours
gcr.io/google_containers/kube-apiserver:fa481b6112db7dcce46bfc8cfbf149a2  Up 13 hours
gcr.io/google_containers/etcd:2.2.1                                       Up 13 hours
gcr.io/google_containers/etcd:2.2.1                                       Up 13 hours
gcr.io/google_containers/rescheduler:v0.2.1                               Up 13 hours
gcr.io/google_containers/glbc:0.8.0                                       Up 13 hours
gcr.io/google-containers/kube-addon-manager:v5.1                          Up 13 hours
gcr.io/google_containers/etcd-empty-dir-cleanup:0.0.1                     Up 13 hours
gcr.io/google_containers/kube-controller-manager:9b1fc8f7afac597ccb49e34778214c49  Up 13 hours
gcr.io/google_containers/kube-scheduler:67b73a442b6a6f362a086ea4ab8dc1cd  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                                  Up 13 hours
```

Even though we have not deployed any applications on Kubernetes yet, we can note that there are several containers already running. The following is a brief description of each container:

- `fluentd-gcp`: This container collects and sends the cluster logs file to the Google Cloud Logging service.
- `node-problem-detector`: This container is a daemon that runs on every node and currently detects issues at the hardware and kernel layer.
- `rescheduler`: This is another add-on container that makes sure critical components are always running. In cases of low resource availability, it may even remove less critical pods to make room.
- `glbc`: This is another Kubernetes add-on container that provides Google Cloud Layer 7 load balancing using the new Ingress capability.
- `kube-addon-manager`: This component is core to the extension of Kubernetes through various add-ons. It also periodically applies any changes to the `/etc/kubernetes/addons` directory.
- `etcd-empty-dir-cleanup`: A utility to clean up empty keys in `etcd`.

- `kube-controller-manager`: This is a controller manager that controls a variety of cluster functions, ensuring accurate and up-to-date replication is one of its vital roles. Additionally, it monitors, manages, and discovers new nodes. Finally, it manages and updates service endpoints.
- `kube-apiserver`: This container runs the API server. As we explored in the Swagger interface, this RESTful API allows us to create, query, update, and remove various components of our Kubernetes cluster.
- `kube-scheduler`: This scheduler takes unscheduled pods and binds them to nodes based on the current scheduling algorithm.
- `etcd`: This runs the `etcd` software built by CoreOS, and it is a distributed and consistent key-value store. This is where the Kubernetes cluster state is stored, updated, and retrieved by various components of K8s.
- `pause`: This container is often referred to as the pod infrastructure container and is used to set up and hold the networking namespace and resource limits for each pod.

I omitted the `amd64` for many of these names to make this more generic. The purpose of the pods remains the same.

To exit the SSH session, simply type `exit` at the prompt.

In the next chapter, we will also show how a few of these services work together in the first image, *Kubernetes core architecture*.

Services running on the minions

We could SSH to one of the minions, but since Kubernetes schedules workloads across the cluster, we would not see all the containers on a single minion. However, we can look at the pods running on all the minions using the `kubectl` command:

```
$ kubectl get pods
No resources found.
```

Since we have not started any applications on the cluster yet, we don't see any pods. However, there are actually several system pods running pieces of the Kubernetes infrastructure. We can see these pods by specifying the `kube-system` namespace. We will explore namespaces and their significance later, but for now, the `--namespace=kube-system` command can be used to look at these K8s system resources, as follows:

```
$ kubectl get pods --namespace=kube-system
jesse@kubernetes-master ~ $ kubectl get pods --namespace=kube-system
NAME READY STATUS RESTARTS AGE
etcd-server-events-kubernetes-master 1/1 Running 0 50m
etcd-server-kubernetes-master 1/1 Running 0 50m
event-exporter-v0.1.7-64464bff45-rg88v 1/1 Running 0 51m
fluentd-gcp-v2.0.10-c4ptt 1/1 Running 0 50m
fluentd-gcp-v2.0.10-d9c5z 1/1 Running 0 50m
fluentd-gcp-v2.0.10-ztdzs 1/1 Running 0 51m
fluentd-gcp-v2.0.10-zxx6k 1/1 Running 0 50m
heapster-v1.5.0-584689c78d-z9blq 4/4 Running 0 50m
kube-addon-manager-kubernetes-master 1/1 Running 0 50m
kube-apiserver-kubernetes-master 1/1 Running 0 50m
kube-controller-manager-kubernetes-master 1/1 Running 0 50m
kube-dns-774d5484cc-gcgdx 3/3 Running 0 51m
kube-dns-774d5484cc-hgm9r 3/3 Running 0 50m
kube-dns-autoscaler-69c5cbdcdd-8hj5j 1/1 Running 0 51m
kube-proxy-kubernetes-minion-group-012f 1/1 Running 0 50m
kube-proxy-kubernetes-minion-group-699m 1/1 Running 0 50m
kube-proxy-kubernetes-minion-group-sj9r 1/1 Running 0 50m
kube-scheduler-kubernetes-master 1/1 Running 0 50m
kubernetes-dashboard-74f855c8c6-v4f6x 1/1 Running 0 51m
l7-default-backend-57856c5f55-2lz6w 1/1 Running 0 51m
l7-lb-controller-v0.9.7-kubernetes-master 1/1 Running 0 50m
metrics-server-v0.2.1-7f8dd98c8f-v9b4c 2/2 Running 0 50m
monitoring-influxdb-grafana-v4-554f5d97-l7q4k 2/2 Running 0 51m
rescheduler-v0.3.1-kubernetes-master 1/1 Running 0 50m
```

The first six lines should look familiar. Some of these are the services we saw running on the master, and we will see pieces of these on the nodes. There are a few additional services we have not seen yet. The `kube-dns` option provides the DNS and service discovery plumbing, `kubernetes-dashboard-xxxx` is the user interface for Kubernetes, `l7-default-backend-xxxx` provides the default load balancing backend for the new layer-7 load balancing capability, and `heapster-v1.2.0-xxxx` and `monitoring-influx-grafana` provide the Heapster database and user interface to monitor resource usage across the cluster.

Finally, `kube-proxy-kubernetes-minion-group-xxxx` is the proxy, which directs traffic to the proper backing services and pods running on our cluster. The `kube-apiserver` validates and configures data for the API objects, which include services, replication controllers, pods, and other Kubernetes objects. The `rescheduler` guarantees the scheduling of critical system add-ons, given that the cluster has enough available resources.

If we did SSH into a random minion, we would see several containers that run across a few of these pods. A sample might look like the following:

```
IMAGE                                                                  STATUS
gcr.io/google_containers/exechealthz-amd64:1.2                         Up 13 hours
gcr.io/google_containers/kube-dnsmasq-amd64:1.4                        Up 13 hours
gcr.io/google_containers/heapster_grafana:v3.1.1                       Up 13 hours
gcr.io/google_containers/kubedns-amd64:1.8                             Up 13 hours
gcr.io/google_containers/heapster_influxdb:v0.7                        Up 13 hours
gcr.io/google_containers/defaultbackend:1.0                            Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                               Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                               Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                               Up 13 hours
gcr.io/google_containers/fluentd-gcp:1.25                              Up 13 hours
gcr.io/google_containers/node-problem-detector:v0.1                    Up 13 hours
gcr.io/google_containers/kube-proxy:b87ffd2bf726a72a00bbc021970cb855   Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                               Up 13 hours
gcr.io/google_containers/pause-amd64:3.0                               Up 13 hours
gcr.io/google_containers/pause-amd64:3_0                               Up 13 hours
```

Again, we saw a similar lineup of services on the master. The services we did not see on the master include the following:

- `kubedns`: This container monitors the service and endpoint resources in Kubernetes and synchronizes any changes to DNS lookups.
- `kube-dnsmasq`: This is another container that provides DNS caching.
- `dnsmasq-metrics`: This provides metric reporting for DNS services in cluster.
- `l7-defaultbackend`: This is the default backend for handling the GCE L7 load balancer and Ingress.
- `kube-proxy`: This is the network and service proxy for your cluster. This component makes sure that service traffic is directed to wherever your workloads are running on the cluster. We will explore this in more depth later in this book.
- `heapster`: This container is for monitoring and analytics.
- `addon-resizer`: This cluster utility is for scaling containers.

- `heapster_grafana`: This tracks resource usage and monitoring.
- `heapster_influxdb`: This time series database is for Heapster data.
- `cluster-proportional-autoscaler`: This cluster utility is for scaling containers in proportion to the cluster size.
- `exechealthz`: This performs health checks on the pods.

Again, I have omitted the `amd64` for many of these names to make this more generic. The purpose of the pods remains the same.

Tearing down a cluster

Alright, this is our first cluster on GCE, but let's explore some other providers. To keep things simple, we need to remove the one we just created on GCE. We can tear down the cluster with one simple command:

```
$ cluster/kube-down.sh
```

Working with other providers

By default, Kubernetes uses the GCE provider for Google Cloud. In order to use other cloud providers, we can explore a rapidly expanding tool set of different options. Let's use AWS for this example, where we have two main options: kops (`https://github.com/kubernetes/kops`) and kube-aws (`https://github.com/kubernetes-incubator/kube-aws`). For reference, the following `KUBERNETES_PROVIDER` are listed in this table:

Provider	**KUBERNETES_PROVIDER value**	**Type**
Google Compute Engine	`gce`	Public cloud
Google Container Engine	`gke`	Public cloud
Amazon Web Services	`aws`	Public cloud
Microsoft Azure	`azure`	Public cloud
Hashicorp vagrant	`vagrant`	Virtual development environment
VMware vSphere	`vsphere`	Private cloud/on-premise virtualization
`libvirt` running CoreOS	`libvirt-coreos`	Virtualization management tool
Canonical Juju (folks behind Ubuntu)	`juju`	OS service orchestration tool

CLI setup

Let's try setting up the cluster on AWS. As a prerequisite, we need to have the AWS CLI installed and configured for our account. The AWS CLI installation and configuration documentation can be found at the following links:

- Installation documentation: `http://docs.aws.amazon.com/cli/latest/userguide/installing.html#install-bundle-other-os`
- Configuration documentation: `http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-started.html`

You'll also need to configure your credentials as recommended by AWS (refer to `https://docs.aws.amazon.com/sdk-for-go/v1/developer-guide/configuring-sdk.html#specifying-credentials`) in order to use kops. To get started, you'll need to first install the CLI tool (refer to `https://github.com/kubernetes/kops/blob/master/docs/install.md`). If you're running on Linux, you can install the tools as follows:

```
curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag_name | cut -d '"' -f 4)/kops-darwin-amd64
chmod +x ./kops
sudo mv ./kops /usr/local/bin/
```

If you're installing this for macOS, you can use `brew update && brew install kops` from the command-line Terminal. As a reminder, you'll need `kubectl` installed if you haven't already! Check the instructions in the preceding links to confirm the installation.

IAM setup

In order for us to use kops, we'll need an IAM role created in AWS with the following permissions:

```
AmazonEC2FullAccess
AmazonRoute53FullAccess
AmazonS3FullAccess
IAMFullAccess
AmazonVPCFullAccess
```

Once you've created those pieces manually in the AWS GUI, you can run the following commands from your PC to set up permissions with the correct access:

```
aws iam create-group --group-name kops

aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonEC2FullAccess --group-name kops
aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonRoute53FullAccess --group-name kops
aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonS3FullAccess --group-name kops
aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/IAMFullAccess --group-name kops
aws iam attach-group-policy --policy-arn
arn:aws:iam::aws:policy/AmazonVPCFullAccess --group-name kops

aws iam create-user --user-name kops

aws iam add-user-to-group --user-name kops --group-name kops

aws iam create-access-key --user-name kops
```

In order to use this newly created kops user to interact with the kops tool, you need to copy down the `SecretAccessKey` and `AccessKeyID` from the output JSON, and then configure the AWS CLI as follows:

```
# configure the aws client to use your new IAM user
aws configure # Use your new access and secret key here
aws iam list-users # you should see a list of all your IAM users here
# Because "aws configure" doesn't export these vars for kops to use, we
export them now
export AWS_ACCESS_KEY_ID=$(aws configure get aws_access_key_id)
export AWS_SECRET_ACCESS_KEY=$(aws configure get aws_secret_access_key)
```

We're going to use a gossip-based cluster to bypass a kops configuration requirement of public DNS zones. This requires kops 1.6.2 or later, and allows you to create a locally registered cluster that requires a name ending in `.k8s.local`. More on that in a bit.

If you'd like to explore how to purchase and set up publicly routable DNS through a provider, you can review the available scenarios in the kops documentation here: `https://github.com/kubernetes/kops/blob/master/docs/aws.md#configure-dns`.

Cluster state storage

Since we're building resources in the cloud using configuration management, we're going to need to store the representation of our cluster in a dedicated S3 bucket. This source of truth will allow us to maintain a single location for the configuration and state of our Kubernetes cluster. Please prepend your bucket name with a unique value.

You'll need to have `kubectl`, `kops`, the `aws cli`, and IAM credentials set up for yourself at this point!

Be sure to create your bucket in the `us-east-1` region for now, as kops is currently opinionated as to where the bucket belongs:

```
aws s3api create-bucket \
 --bucket gsw-k8s-3-state-store \
 --region us-east-1
```

Let's go ahead and set up versioning as well, so you can roll your cluster back to previous states in case anything goes wrong. Behold the power of Infrastructure as Code!

```
aws s3api put-bucket-versioning --bucket gsw-k8s-3-state-store --
versioning-configuration Status=Enabled
```

Creating your cluster

We'll go ahead and use the `.k8s.local` settings mentioned previously to simplify the DNS setup of the cluster. If you'd prefer, you can also use the name and state flags available within kops to avoid using environment variables. Let's prepare the local environment first:

```
$ export NAME=gswk8s3.k8s.local
$ export KOPS_STATE_STORE=s3://gsw-k8s-3-state-store
$ aws s3api create-bucket --bucket gsw-k8s-3-state-store --region us-east-1
{
  "Location": "/gsw-k8s-3-state-store"
}
$
```

Let's spin up our cluster in Ohio, and verify that we can see that region first:

```
$ aws ec2 describe-availability-zones --region us-east-2
{
  "AvailabilityZones": [
  {
  "State": "available",
  "ZoneName": "us-east-2a",
  "Messages": [],
  "RegionName": "us-east-2"
  },
  {
  "State": "available",
  "ZoneName": "us-east-2b",
  "Messages": [],
  "RegionName": "us-east-2"
  },
  {
  "State": "available",
  "ZoneName": "us-east-2c",
  "Messages": [],
  "RegionName": "us-east-2"
  }
  ]
}
```

Great! Let's make some Kubernetes. We're going to use the most basic kops cluster command available, though there are much more complex examples available in the documentation (`https://github.com/kubernetes/kops/blob/master/docs/high_availability.md`):

```
kops create cluster --zones us-east-2a ${NAME}
```

With kops and generally with Kubernetes, everything is going to be created within **Auto Scaling groups** (**ASGs**).

Read more about AWS autoscaling groups here—they're essential: `https://docs.aws.amazon.com/autoscaling/ec2/userguide/AutoScalingGroup.html`.

Once you run this command, you'll get a whole lot of configuration output in what we call a dry run format. This is similar to the Terraform idea of a Terraform plan, which lets you see what you're about to build in AWS and lets you edit the output accordingly.

At the end of the output, you'll see the following text, which gives you some basic suggestions on the next steps:

```
Must specify --yes to apply changes
Cluster configuration has been created.

Suggestions:
* list clusters with: kops get cluster
* edit this cluster with: kops edit cluster gwsk8s3.k8s.local
* edit your node instance group: kops edit ig --name=gwsk8s3.k8s.local
nodes
* edit your master instance group: kops edit ig --name=gwsk8s3.k8s.local
master-us-east-2a

Finally configure your cluster with: kops update cluster gwsk8s3.k8s.local
--yes
```

If you don't have an SSH keypair in your `~/.ssh` directory, you'll need to create one. This article will lead you through the steps: `https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/`.

Once you've confirmed that you like the look of the output, you can create the cluster:

```
kops update cluster gwsk8s3.k8s.local --yes
```

This will give you a lot of output about cluster creation that you can follow along with:

```
I0320 21:37:34.761784 29197 apply_cluster.go:450] Gossip DNS: skipping DNS
validation
I0320 21:37:35.172971 29197 executor.go:91] Tasks: 0 done / 77 total; 30
can run
I0320 21:37:36.045260 29197 vfs_castore.go:435] Issuing new certificate:
"apiserver-aggregator-ca"
I0320 21:37:36.070047 29197 vfs_castore.go:435] Issuing new certificate:
"ca"
I0320 21:37:36.727579 29197 executor.go:91] Tasks: 30 done / 77 total; 24
can run
I0320 21:37:37.740018 29197 vfs_castore.go:435] Issuing new certificate:
"apiserver-proxy-client"
I0320 21:37:37.758789 29197 vfs_castore.go:435] Issuing new certificate:
"kubecfg"
I0320 21:37:37.830861 29197 vfs_castore.go:435] Issuing new certificate:
"kube-controller-manager"
I0320 21:37:37.928930 29197 vfs_castore.go:435] Issuing new certificate:
"kubelet"
I0320 21:37:37.940619 29197 vfs_castore.go:435] Issuing new certificate:
"kops"
```

```
I0320 21:37:38.095516 29197 vfs_castore.go:435] Issuing new certificate:
"kubelet-api"
I0320 21:37:38.124966 29197 vfs_castore.go:435] Issuing new certificate:
"kube-proxy"
I0320 21:37:38.274664 29197 vfs_castore.go:435] Issuing new certificate:
"kube-scheduler"
I0320 21:37:38.344367 29197 vfs_castore.go:435] Issuing new certificate:
"apiserver-aggregator"
I0320 21:37:38.784822 29197 executor.go:91] Tasks: 54 done / 77 total; 19
can run
I0320 21:37:40.663441 29197 launchconfiguration.go:333] waiting for IAM
instance profile "nodes.gswk8s3.k8s.local" to be ready
I0320 21:37:40.889286 29197 launchconfiguration.go:333] waiting for IAM
instance profile "masters.gswk8s3.k8s.local" to be ready
I0320 21:37:51.302353 29197 executor.go:91] Tasks: 73 done / 77 total; 3
can run
I0320 21:37:52.464204 29197 vfs_castore.go:435] Issuing new certificate:
"master"
I0320 21:37:52.644756 29197 executor.go:91] Tasks: 76 done / 77 total; 1
can run
I0320 21:37:52.916042 29197 executor.go:91] Tasks: 77 done / 77 total; 0
can run
I0320 21:37:53.360796 29197 update_cluster.go:248] Exporting kubecfg for
cluster
kops has set your kubectl context to gswk8s3.k8s.local
```

As with GCE, the setup activity will take a few minutes. It will stage files in **S3** and create the appropriate instances, **Virtual Private Cloud** (**VPC**), security groups, and so on in our AWS account. Then, the Kubernetes cluster will be set up and started. Once everything is finished and started, we should see some options on what comes next:

```
Cluster is starting. It should be ready in a few minutes.

Suggestions:
 * validate cluster: kops validate cluster
 * list nodes: kubectl get nodes --show-labels
 * ssh to the master: ssh -i ~/.ssh/id_rsa admin@api.gswk8s3.k8s.local
The admin user is specific to Debian. If not using Debian please use the
appropriate user based on your OS.
 * read about installing addons:
https://github.com/kubernetes/kops/blob/master/docs/addons.md
```

You'll be able to see instances and security groups, and a VPC will be created for your cluster. The `kubectl` context will also be pointed at your new AWS cluster so that you can interact with it:

Once again, we will SSH into master. This time, we can use the native SSH client and the admin user as the AMI for Kubernetes in kops is Debian. We'll find the key files in `/home/<username>/.ssh`:

```
$ ssh -v -i /home/<username>/.ssh/<your_id_rsa_file> admin@<Your master IP>
```

If you have trouble with your SSH key, you can set it manually on the cluster by creating a secret, adding it to the cluster, and checking if the cluster requires a rolling update:

```
$ kops create secret --name gswk8s3.k8s.local sshpublickey admin -i
~/.ssh/id_rsa.pub
$ kops update cluster --yes
Using cluster from kubectl context: gswk8s3.k8s.local
I0320 22:03:42.823049 31465 apply_cluster.go:450] Gossip DNS: skipping DNS
validation
I0320 22:03:43.220675 31465 executor.go:91] Tasks: 0 done / 77 total; 30
can run
I0320 22:03:43.919989 31465 executor.go:91] Tasks: 30 done / 77 total; 24
can run
I0320 22:03:44.343478 31465 executor.go:91] Tasks: 54 done / 77 total; 19
can run
I0320 22:03:44.905293 31465 executor.go:91] Tasks: 73 done / 77 total; 3
can run
I0320 22:03:45.385288 31465 executor.go:91] Tasks: 76 done / 77 total; 1
can run
I0320 22:03:45.463711 31465 executor.go:91] Tasks: 77 done / 77 total; 0
can run
I0320 22:03:45.675720 31465 update_cluster.go:248] Exporting kubecfg for
cluster
kops has set your kubectl context to gswk8s3.k8s.local

Cluster changes have been applied to the cloud.

Changes may require instances to restart: kops rolling-update cluster

$ kops rolling-update cluster --name gswk8s3.k8s.local
```

```
NAME STATUS NEEDUPDATE READY MIN MAX NODES
master-us-east-2a Ready 0 1 1 1 1
nodes Ready 0 2 2 2 2

No rolling-update required.
$
```

Once you've gotten into the cluster master, we can look at the containers. We'll use `sudo docker ps --format 'table {{.Image}}t{{.Status}}'` to explore the running containers. We should see the following:

```
admin@ip-172-20-47-159:~$ sudo docker container ls --format 'table
{{.Image}}\t{{.Status}}'
IMAGE STATUS
kope/dns-
controller@sha256:97f80ad43ff833b254907a0341c7fe34748e007515004cf0da09727c5
442f53b Up 29 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 29 minutes
gcr.io/google_containers/kube-
apiserver@sha256:71273b57d811654620dc7a0d22fd893d9852b6637616f8e7e3f4507c60
ea7357 Up 30 minutes
gcr.io/google_containers/etcd@sha256:19544a655157fb089b62d4dac02bbd095f82ca
245dd5e31dd1684d175b109947 Up 30 minutes
gcr.io/google_containers/kube-
proxy@sha256:cc94b481f168bf96bd21cb576cfaa06c55807fcba8a6620b51850e1e30febe
b4 Up 30 minutes
gcr.io/google_containers/kube-controller-
manager@sha256:5ca59252abaf231681f96d07c939e57a05799d1cf876447fe6c2e1469d58
2bde Up 30 minutes
gcr.io/google_containers/etcd@sha256:19544a655157fb089b62d4dac02bbd095f82ca
245dd5e31dd1684d175b109947 Up 30 minutes
gcr.io/google_containers/kube-
scheduler@sha256:46d215410a407b9b5a3500bf8b421778790f5123ff2f4364f99b352a2b
a62940 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
gcr.io/google_containers/pause-amd64:3.0 Up 30 minutes
protokube:1.8.1
```

We can see some of the same containers as our GCE cluster had. However, there are several missing. We can see the core Kubernetes components, but the `fluentd-gcp` service is missing, as well as some of the newer utilities such as `node-problem-detector`, `rescheduler`, `glbc`, `kube-addon-manager`, and `etcd-empty-dir-cleanup`. This reflects some of the subtle differences in the `kube-up` script between various public cloud providers. This is ultimately decided by the efforts of the large Kubernetes open-source community, but GCP often has many of the latest features first.

You also have a command that allows you to check on the state of the cluster in `kops validate cluster`, which allows you to make sure that the cluster is working as expected. There's also a lot of handy modes that kops provides that allow you to do various things with the output, provisioners, and configuration of the cluster.

Other modes

There are various other modes to take into consideration, including the following:

- **Build a terraform model**: `--target=terraform`. The terraform model will be built in `out/terraform`.
- **Build a cloudformation model**: `--target=cloudformation`. The Cloudformation JSON file will be built in `out/cloudformation`.
- **Specify the K8s build to run**: `--kubernetes-version=1.2.2`.
- **Run nodes in multiple zones**: `--zones=us-east-1b,us-east-1c,us-east-1d`.
- **Run with a HA master**: `--master-zones=us-east-1b,us-east-1c,us-east-1d`.
- **Specify the number of nodes**: `--node-count=4`.
- **Specify the node size**: `--node-size=m4.large`.
- **Specify the master size**: `--master-size=m4.large`.
- **Override the default DNS zone**: `--dns-zone=<my.hosted.zone>`.

The full list of CLI documentation can be found here: `https://github.com/kubernetes/kops/tree/master/docs/cli`.

Another tool for diagnosing the cluster status is the `componentstatuses` command, which will inform you of state of the major Kubernetes moving pieces:

```
$ kubectl get componentstatuses
NAME STATUS MESSAGE ERROR
scheduler Healthy ok
controller-manager Healthy ok
etcd-0 Healthy {"health": "true"}
```

Resetting the cluster

You just had a little taste of running the cluster on AWS. For the remainder of this book, I will be basing my examples on a GCE cluster. For the best experience following along, you can get back to a GCE cluster easily.

Simply tear down the AWS cluster, as follows:

```
$ kops delete cluster --name ${NAME} --yes
```

If you omit the `--yes` flag, you'll get a similar dry run output that you can confirm. Then, create a GCE cluster again using the following, and in doing so making sure that you're back in the directory where you installed the Kubernetes code:

```
$ cd ~/<kubernetes_install_dir>
$ kube-up.sh
```

Investigating other deployment automation

If you'd like to learn more about other tools for cluster automation, we recommend that you visit the kube-deploy repository, which has references to community maintained Kubernetes cluster deployment tools.

Visit `https://github.com/kubernetes/kube-deploy` to learn more.

Local alternatives

The `kube-up.sh` script and `kops` are pretty handy ways to get started using Kubernetes on your platform of choice. However, they're not without flaws and can sometimes run aground when conditions are not just so.

Luckily, since K8's inception, a number of alternative methods for creating clusters have emerged. We'd recommend checking out Minikube in particular, as it's an extremely simple and local development environment that you can use to test out your Kubernetes configuration.

This project can be found here: `https://github.com/kubernetes/minikube`.

It's important to mention that you're going to need a hypervisor on your machine to run Minikube. For Linux, you can use kvm/kvm2, or VirtualBox, and on macOS you can run native xhyve or VirtualBox. For Windows, Hyper-V is the default hypervisor.

The main limitation for this project is that it only runs a single node, which limits our exploration of certain advanced topics that require multiple machines. Minikube is a great resource for simple or local development however, and can be installed very simply on your Linux VM with the following:

```
$ curl -Lo minikube
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd6
4 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

Or install it on macOS with the following:

```
$ brew cask install minikube
```

We'll cover how to get started with Minikube with the following commands:

```
$ minikube start
Starting local Kubernetes v1.7.5 cluster...
Starting VM...
SSH-ing files into VM...
Setting up certs...
Starting cluster components...
Connecting to cluster...
Setting up kubeconfig...
Kubectl is now configured to use the cluster.
```

You can create a sample deployment quite simply:

```
$ kubectl run hello-minikube --image=k8s.gcr.io/echoserver:1.4 --port=8080
deployment "hello-minikube" created
$ kubectl expose deployment hello-minikube --type=NodePort
service "hello-minikube" exposed
```

Once you have your cluster and service up and running, you can interact with it simply by using the `kubectl` tool and the `context` command. You can get to the Minikube dashboard with `minikube dashboard`.

Minikube is powered by localkube (`https://github.com/kubernetes/minikube/tree/master/pkg/localkube`) and libmachine (`https://github.com/docker/machine/tree/master/libmachine`). Check them out!

Additionally, we've already referenced a number of managed services, including GKE, EKS, and Microsoft **Azure Container Service** (**ACS**), which provide an automated installation and some managed cluster operations. We will look at a demos of these in `Chapter 14`, *Hardening Kubernetes*.

Starting from scratch

Finally, there is the option to start from scratch. Luckily, starting in 1.4, the Kubernetes team has put a major focus on simplifying the cluster setup process. To that end, they have introduced kubeadm for Ubuntu 16.04, CentOS 7, and HypriotOS v1.0.1+.

Let's take a quick look at spinning up a cluster on AWS from scratch using the kubeadm tool.

Cluster setup

We will need to provision our cluster master and nodes beforehand. For the moment, we are limited to the operating systems and version listed earlier. Additionally, it is recommended that you have at least 1 GB of RAM. All the nodes must have network connectivity to one another.

For this walkthrough, we will need one t2.medium (master node) and three t2.mirco (nodes) sized instances on AWS. These instance have burstable CPU and come with the minimum 1 GB of RAM that's required. We will need to create one master and three worker nodes.

We will also need to create some security groups for the cluster. The following ports are needed for the master:

Type	**Protocol**	**Port range**	**Source**
All traffic	All	All	{This SG ID (Master SG)}
All traffic	All	All	{Node SG ID}
SSH	TCP	`22`	{Your Local Machine's IP}
HTTPS	TCP	`443`	{Range allowed to access K8s API and UI}

The following table shows the port's node security groups:

Type	**Protocol**	**Port range**	**Source**
All traffic	All	All	{Master SG ID}
All traffic	All	All	{This SG ID (Node SG)}
SSH	TCP	`22`	{Your Local Machine's IP}

Once you have these SGs, go ahead and spin up four instances (one t2.medium and three t2.mircos) using Ubuntu 16.04. If you are new to AWS, refer to the documentation on spinning up EC2 instances at the following URL: `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/LaunchingAndUsingInstances.html`.

Be sure to identify the t2.medium instance as the master and associate the master security group. Name the other three as nodes and associate the node security group with those.

These steps are adapted from the walk-through in the manual. For more information or to work with an alternative to Ubuntu, refer to `https://kubernetes.io/docs/getting-started-guides/kubeadm/`.

Installing Kubernetes components (kubelet and kubeadm)

Next, we will need to SSH into all four of the instances and install the Kubernetes components.

As the root user, perform the following steps on all four instances:

1. Update the packages and install the `apt-transport-https` package so that we can download from sources that use HTTPS:

```
$ apt-get update
$ apt-get install -y apt-transport-https
```

2. Install the Google Cloud public key:

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.gpg |
apt-key add -
```

3. Next, let's set up the repository:

```
cat <<EOF >/etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
apt-get update
apt-get install -y kubelet kubeadm kubectl docker.io kubernetes-cni
```

You'll need to make sure that the `cgroup` driver used by the `kubelet` on the master node is configured correctly to work with Docker. Make sure you're on the master node, then run the following:

```
docker info | grep -i cgroup
cat /etc/systemd/system/kubelet.service.d/10-kubeadm.conf
```

If these items don't match, you're going to need to change the kubelet configuration to match the Docker driver. Running `sed -i "s/cgroup-driver=systemd/cgroup-driver=cgroupfs/g" /etc/systemd/system/kubelet.service.d/10-kubeadm.conf` should fix the settings, or you can manually open the `systemd` file and add the correct flag to the appropriate environment. After that's complete, restart the service:

```
$ systemctl daemon-reload
$ systemctl restart kubelet
```

Setting up a master

On the instance you have previously chosen as master, we will run master initialization. Again, as the root, run the following command, and you should see the following output:

```
$ kubeadm init
[init] using Kubernetes version: v1.11.3
[preflight] running pre-flight checks
I1015 02:49:42.378355 5250 kernel_validator.go:81] Validating kernel
version
I1015 02:49:42.378609 5250 kernel_validator.go:96] Validating kernel config
[preflight/images] Pulling images required for setting up a Kubernetes
cluster
[preflight/images] This might take a minute or two, depending on the speed
of your internet connection
[preflight/images] You can also perform this action in beforehand using
'kubeadm config images pull'
[kubelet] Writing kubelet environment file with flags to file
"/var/lib/kubelet/kubeadm-flags.env"
[kubelet] Writing kubelet configuration to file
"/var/lib/kubelet/config.yaml"
[preflight] Activating the kubelet service
[certificates] Generated ca certificate and key.
[certificates] Generated apiserver certificate and key.
[certificates] apiserver serving cert is signed for DNS names [master
kubernetes kubernetes.default kubernetes.default.svc
kubernetes.default.svc.cluster.local] and IPs [10.96.0.1 172.17.0.71]
[certificates] Generated apiserver-kubelet-client certificate and key.
[certificates] Generated sa key and public key.
[certificates] Generated front-proxy-ca certificate and key.
[certificates] Generated front-proxy-client certificate and key.
[certificates] Generated etcd/ca certificate and key.
[certificates] Generated etcd/server certificate and key.
[certificates] etcd/server serving cert is signed for DNS names [master
localhost] and IPs [127.0.0.1 ::1]
[certificates] Generated etcd/peer certificate and key.
[certificates] etcd/peer serving cert is signed for DNS names [master
localhost] and IPs [172.17.0.71 127.0.0.1 ::1]
[certificates] Generated etcd/healthcheck-client certificate and key.
[certificates] Generated apiserver-etcd-client certificate and key.
[certificates] valid certificates and keys now exist in
"/etc/kubernetes/pki"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/admin.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/kubelet.conf"
[kubeconfig] Wrote KubeConfig file to disk: "/etc/kubernetes/controller-
manager.conf"
[kubeconfig] Wrote KubeConfig file to disk:
"/etc/kubernetes/scheduler.conf"
```

```
[controlplane] wrote Static Pod manifest for component kube-apiserver to
"/etc/kubernetes/manifests/kube-apiserver.yaml"
[controlplane] wrote Static Pod manifest for component kube-controller-
manager to "/etc/kubernetes/manifests/kube-controller-manager.yaml"
[controlplane] wrote Static Pod manifest for component kube-scheduler to
"/etc/kubernetes/manifests/kube-scheduler.yaml"
[etcd] Wrote Static Pod manifest for a local etcd instance to
"/etc/kubernetes/manifests/etcd.yaml"
[init] waiting for the kubelet to boot up the control plane as Static Pods
from directory "/etc/kubernetes/manifests"
[init] this might take a minute or longer if the control plane images have
to be pulled
[apiclient] All control plane components are healthy after 43.001889
seconds
[uploadconfig] storing the configuration used in ConfigMap "kubeadm-config"
in the "kube-system" Namespace
[kubelet] Creating a ConfigMap "kubelet-config-1.11" in namespace kube-
system with the configuration for the kubelets in the cluster
[markmaster] Marking the node master as master by adding the label "node-
role.kubernetes.io/master=''"
[markmaster] Marking the node master as master by adding the taints [node-
role.kubernetes.io/master:NoSchedule]
[patchnode] Uploading the CRI Socket information "/var/run/dockershim.sock"
to the Node API object "master" as an annotation
[bootstraptoken] using token: o760dk.q4l5au0jyx4vg6hr
[bootstraptoken] configured RBAC rules to allow Node Bootstrap tokens to
post CSRs in order for nodes to get long term certificate credentials
[bootstraptoken] configured RBAC rules to allow the csrapprover controller
automatically approve CSRs from a Node Bootstrap Token
[bootstraptoken] configured RBAC rules to allow certificate rotation for
all node client certificates in the cluster
[bootstraptoken] creating the "cluster-info" ConfigMap in the "kube-public"
namespace
[addons] Applied essential addon: CoreDNS
[addons] Applied essential addon: kube-proxy

Your Kubernetes master has initialized successfully!

To start using your cluster, you need to run the following as a regular
user:

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

You should now deploy a pod network to the cluster.
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:
  https://kubernetes.io/docs/concepts/cluster-administration/addons/
```

```
You can now join any number of machines by running the following on each
node
as root:

  kubeadm join 172.17.0.71:6443 --token o760dk.q4l5au0jyx4vg6hr --
discovery-token-ca-cert-hash
sha256:453e2964eb9cc0cecfdb167194f60c6f7bd8894dc3913e0034bf0b33af4f40f5
```

To start using your cluster, you need to run as a regular user:

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

You should now deploy a pod network to the cluster. Run `kubectl apply -f [podnetwork].yaml` with one of the options listed at `https://kubernetes.io/docs/concepts/cluster-administration/addons/`.

You can now join any number of machines by running the following on each node as root:

```
kubeadm join --token <token> <master-ip>:<master-port> --discovery-token-
ca-cert-hash sha256:<hash>
```

Note that initialization can only be run once, so if you run into problems, you'll need to use `kubeadm reset`.

Joining nodes

After a successful initialization, you will get a `join` command that can be used by the nodes. Copy this down for the join process later on. It should look similar to this:

```
$ kubeadm join --token=<some token> <master ip address>
```

The token is used to authenticate cluster nodes, so make sure to store it somewhere securely for future use.

Networking

Our cluster will need a networking layer for the pods to communicate on. Note that kubeadm requires a CNI compatible network fabric. The list of plugins currently available can be found here: `http://kubernetes.io/docs/admin/addons/`.

For our example, we will use calico. We will need to create the calico components on our cluster using the following `yaml`. For convenience, you can download it here: `http://docs.projectcalico.org/v1.6/getting-started/kubernetes/installation/hosted/kubeadm/calico.yaml`.

Once you have this file on your master, create the components with the following command:

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

Give this a minute to run setup and then list the `kube-system` nodes in order to check this:

```
$ kubectl get pods --namespace=kube-system
```

You should get a listing similar to the following one with three new calico pods and one completed job that is not shown:

```
NAME                                    READY     STATUS    RESTARTS   AGE
calico-etcd-7ckip                       1/1       Running   0          43s
calico-node-em917                       2/2       Running   0          43s
calico-policy-controller-i43ct          1/1       Running   0          43s
dummy-2088944543-efrgw                  1/1       Running   0          2m
etcd-ip-172-30-0-26                     1/1       Running   0          1m
kube-apiserver-ip-172-30-0-26           1/1       Running   0          2m
kube-controller-manager-ip-172-30-0-26  1/1       Running   0          2m
kube-discovery-1150918428-lkntn         1/1       Running   0          2m
kube-dns-654381707-6u52r                2/3       Running   0          1m
kube-proxy-00wu7                        1/1       Running   0          1m
kube-scheduler-ip-172-30-0-26           1/1       Running   0          1m
  info: 1 completed object(s) was(were) not shown in pods list. Pass --show-all
to see all objects.
```

Calico setup

Joining the cluster

Now, we need to run the `join` command we copied earlier, on each of our node instances:

```
$ kubeadm join --token=<some token> <master ip address>
```

Once you've finished that, you should be able to see all nodes from the master by running the following command:

```
$ kubectl get nodes
```

If all went well, this will show three nodes and one master, as shown here:

```
NAME              STATUS         AGE
ip-172-30-0-22    Ready          6m
ip-172-30-0-26    Ready,master   8m
ip-172-30-0-28    Ready          6m
ip-172-30-0-8     Ready          6m
```

Summary

We took a very brief look at how containers work and how they lend themselves to the new architecture patterns in microservices. You should now have a better understanding of how these two forces will require a variety of operations and management tasks, and how Kubernetes offers strong features to address these challenges. We created two different clusters on both GCE and AWS, and explored the startup script as well as some of the built-in features of Kubernetes. Finally, we looked at the alternatives to the `kube-up` script in kops, and tried our hand at manual cluster configuration with the kubeadm tool on AWS with Ubuntu 16.04.

In the next chapter, we will explore the core concept and abstractions K8s provides to manage containers and full application stacks. We will also look at basic scheduling, service discovery, and health checking.

Questions

1. Name three places where you can easily deploy a Kubernetes cluster.
2. What are other types of pre-existing virtualization technologies that predate containers?
3. Name as many cgroup controls as you can!
4. What are some of the reasons why enabling CI/CD with containers is so important to organizations?
5. What prerequisites are required to get a Kubernetes cluster up and running on AWS or GCE?
6. Name four services running on the Kubernetes master nodes. Hint: these are containers.
7. What are some alternatives to the `kube-up.sh` script?
8. What's the tool used for building a Kubernetes cluster from scratch?

Further reading

Want more information on DevOps practices on Kubernetes? Check out *DevOps with Kubernetes*: `https://www.packtpub.com/virtualization-and-cloud/devops-kubernetes`.

You can also read about different applications and automation approaches with the *Kubernetes Cookbook*: `https://www.packtpub.com/virtualization-and-cloud/kubernetes-cookbook`.

2
Building a Foundation with Core Kubernetes Constructs

This chapter will cover the core Kubernetes constructs, namely pods, services, replication controllers, replica sets, and labels. We will describe Kubernetes components, dimensions of the API, and Kubernetes objects. We will also dig into the major Kubernetes cluster components. A few simple application examples will be included to demonstrate each construct. This chapter will also cover basic operations for your cluster. Finally, health checks and scheduling will be introduced with a few examples.

The following topics will be covered in this chapter:

- Kubernetes' overall architecture
- The context of Kubernetes architecture within system theory
- Introduction to core Kubernetes constructs, architecture, and components
- How labels can simplify the management of a Kubernetes cluster
- Monitoring services and container health
- Setting up scheduling constraints based on available cluster resources

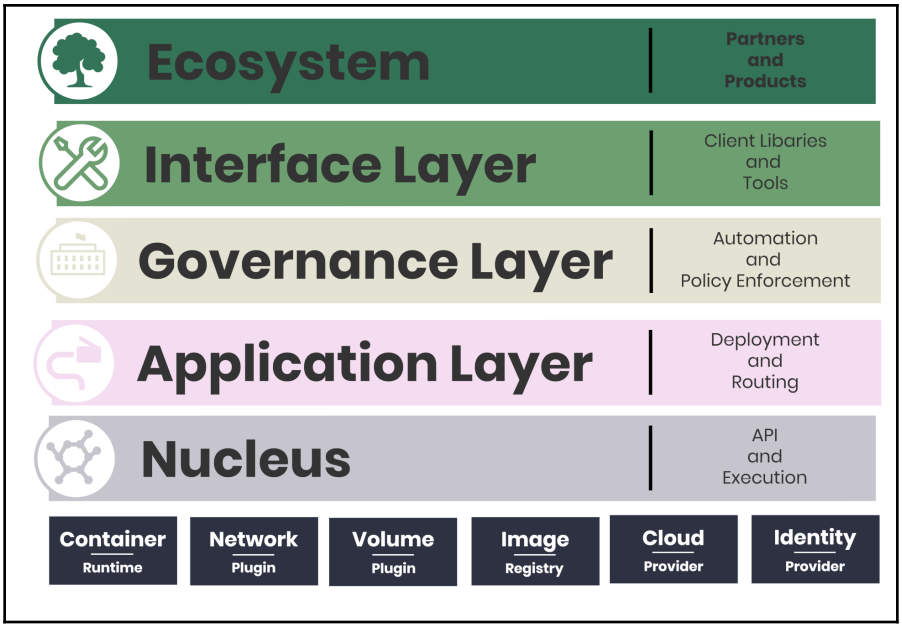
Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes over the web: `https://labs.play-with-k8s.com/`.

Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter02`.

The Kubernetes system

To understand the complex architecture and components of Kubernetes, we should take a step back and look at the landscape of the overall system in order to understand the context and place of each moving piece. This book focuses mainly on the technical pieces and processes of the Kubernetes software, but let's examine the system from a top-down perspective. In the following diagram, you can see the major parts of the Kubernetes system, which is a great way to think about the classification of the parts we'll describe and utilize in this book:



Let's take a look at each piece, starting from the bottom.

Nucleus

The nucleus of the Kubernetes system is devoted to providing a standard API and manner in which operators and/or software can execute work on the cluster. The nucleus is the bare minimum set of functionality that should be considered absolutely stable in order to build up the layers above. Each piece of this layer is clearly documented, and these pieces are required to build higher-order concepts at other layers of the system. You can consider the APIs here to make up the core bits of the Kubernetes control plane.

The cluster control plane is the first half of the Kubernetes nucleus, and it provides the RESTful APIs that allow operators to utilized the mostly CRUD-based operations of the cluster. It is important to note that the Kubernetes nucleus and consequently the cluster control plane was built with multi-tenancy in mind, so the layer must be flexible enough to provide logical separation of teams or workloads within a single cluster. The cluster control plane follows API conventions that allow it to take advantage of shared services such as identity and auditing, and has access to the namespaces and events of the cluster.

The second half of the nucleus is execution. While there are a number of controllers in Kubernetes, such as the replication controller, replica set, and deployments, the kubelet is the most important controller and it forms the basis of the node and pod APIs that allow us to interact with the container execution layer. Kubernetes builds upon the kubelet with the concept of pods, which allow us to manage many containers and their constituent storage as a core capability of the system. We'll dig more into pods later.

Below the nucleus, we can see the various pieces that the kubelet depends on in order to manage the container, network, container storage, image storage, cloud provider, and identity. We've left these intentionally vague as there are several options for each box, and you can pick and choose from standard and popular implementations or experiment with emerging tech. To give you an idea of how many options there are in the base layer, we'll outline container runtime and network plugin options here.

Container Runtime options: You'll use the Kubernetes **Container Runtime Interface** (**CRI**) to interact with the two main container runtimes:

- containerd
- rkt

You're still able to run Docker containers on Kubernetes at this point, and as containerd is the default runtime, it's going to be transparent to the operator at this point due to the defaults. You'll be able to run all of the same `docker <action>` commands on the cluster to introspect and gather information about your clusters.

There are also several competing, emerging formats:

- `cri-containerd`: `https://github.com/containerd/cri-containerd`
- `runv` and `clear` containers, which are hypervisor-based solutions: `https://github.com/hyperhq/runv` and `https://github.com/clearcontainers/runtime`
- `kata` containers, which are a combination of `runv` and clear containers: `https://katacontainers.io/`
- `frakti` containers, which combine `runv` and Docker: `https://github.com/kubernetes/frakti`

You can read more about the CRI here: `http://blog.kubernetes.io/2016/12/container-runtime-interface-cri-in-kubernetes.html`.

Network plugin: You can use the CNI to leverage any of the following plugins or the simple Kubenet networking implementation if you're going to rely on a cloud provider's network segmentation, or if you're going to be running a single node cluster:

- Cilium
- Contiv
- Contrail
- Flannel
- Kube-router
- Multus
- Calico
- Romana
- Weave net

Application layer

The application layer, often referred to as the service fabric or orchestration layer, does all of the fun things we've come to value so highly in Kubernetes: basic deployment and routing, service discovery, load balancing, and self-healing. In order for a cluster operator to manage the life cycle of the cluster, these primitives must be present and functional in this layer. Most containerized applications will depend on the full functionality of this layer, and will interact with these functions in order to provide "orchestration" of the application across multiple cluster hosts. When an application scales up or changes a configuration setting, the application layer will be managed by this layer. The application layer cares about the desired state of the cluster, the application composition, service discovery, load balancing, and routing, and utilizes all of these pieces to keep data flowing from the correct point A to the correct point B.

Governance layer

The governance layer consists of high-level automation and policy enforcement. This layer can be thought of as an opinionated version of the application management layer, as it provides the ability to enforce tenancy, gather metrics, and do intelligent provisioning and autoscaling of containers. The APIs at this layer should be considered options for running containerized applications.

The governance layer allows operators to control methods used for authorization, as well as quotas and control around network and storage. At this layer, functionality should be applicable to scenarios that large enterprises care about, such as operations, security, and compliance scenarios.

Interface layer

The interface layer is made up of commonly used tools, systems, user interfaces, and libraries that other custom Kubernetes distributions might use. The `kubectl` library is a great example of the interface layer, and importantly it's not seen as a privileged part of the Kubernetes system; it's considered a client tool in order to provide maximum flexibility for the Kubernetes API. If you run `$ kubectl -h`, you will get a clear picture of the functionality exposed to the interface layer.

Other pieces at this layer include cluster federation tools, dashboards, Helm, and client libraries such as `client-node`, `KubernetesClient`, and `python`. These tools provide common tasks for you, so you don't have to worry about writing code for authentication, for example. These libraries use the Kubernetes Service Account to authenticate to the cluster.

Ecosystem

The last layer of the Kubernetes system is the ecosystem, and it's by far the busiest and most hectic part of the picture. Kubernetes approach to container orchestration and management is to present the user with the options of a complementary choice; there are plug-in and general purpose APIs available for external systems to utilize. You can consider three types of ecosystem pieces in the Kubernetes system:

- **Above Kubernetes:** All of the glue software and infrastructure that's needed to "make things go" sits at this level, and includes operational ethos such as ChatOps and DevOps, logging and monitoring, Continuous Integration and Delivery, big data systems, and Functions as a Service.

- **Inside Kubernetes:** In short, what's inside a container is outside of Kubernetes. Kubernetes, or **K8s**, cares not at all what you run inside of a container.
- **Below Kubernetes**: These are the gray squares detailed at the bottom of the diagram. You'll need a technology for each piece of foundational technology to make Kubernetes function, and the ecosystem is where you get them. The cluster state store is probably the most famous example of an ecosystem component: `etcd`. Cluster bootstrapping tools such as `minikube`, `bootkube`, `kops`, `kube-aws`, and `kubernetes-anywhere` are other examples of community-provided ecosystem tools.

Let's move on to the architecture of the Kubernetes system, now that we understand the larger context.

The architecture

Although containers bring a helpful layer of abstraction and tooling for application management, Kubernetes brings additional to schedule and orchestrate containers at scale, while managing the full application life cycle.

K8s moves up the stack, giving us constructs to deal with management at the application- or service- level. This gives us automation and tooling to ensure high availability, application stack, and service-wide portability. K8s also allows finer control of resource usage, such as CPU, memory, and disk space across our infrastructure.

Kubernetes architecture is comprised of three main pieces:

- The cluster control plane (the **master**)
- The cluster state (a distributed storage system called etcd)
- Cluster nodes (individual servers running agents called **kubelets**)

The Master

The **cluster control plane**, otherwise known as the **Master**, makes global decisions based on the current and desired state of the cluster, detecting and responding to events as they propagate across the cluster. This includes starting and stopping pods if the replication factor of a replication controller is unsatisfied or running a scheduled cron job.

The overarching goal of the control plane is to report on and work towards a desired state. The API that the master runs depends on the persistent state store, `etcd`, and utilizes the `watch` strategy for minimizing change latency while enabling decentralized component coordination.

Components of the Master can be realistically run on any machine in the cluster, but best practices and production-ready systems dictate that master components should be co-located on a single machine (or a multi-master high availability setup). Running all of the Master components on a single machine allows operators to exclude running user containers on those machines, which is recommended for more reliable control plane operations. The less you have running on your Master node, the better!

We'll dig into the Master components, including `kube-apiserver`, etcd, `kube-scheduler`, `kube-controller-manager`, and `cloud-controller-manager` when we get into more detail on the Master node. It is important to note that the Kubernetes goal with these components is to provide a RESTful API against mostly persistent storage resources and a CRUD (Create, Read, Update, and Delete) strategy. We'll explore the basic primitives around container-specific orchestration and scheduling later in this chapter when we read about services, ingress, pods, deployments, StatefulSet, CronJobs, and ReplicaSets.

Cluster state

The second major piece of the Kubernetes architecture, the cluster state, is the `etcd` key value store. `etcd` is consistent and highly available, and is designed to quickly and reliably provide Kubernetes access to critical cluster current and desired state. etcd is able to provide this distributed coordination of data through such core concepts as leader election and distributed locks. The Kubernetes API, via its API server, is in charge of updating objects in etcd that correspond to the RESTful operations of the cluster. This is very important to remember: the API server is responsible for managing what's stuck into Kubernetes' picture of the world. Other components in this ecosystem watch etcd for changes in order to modify themselves and enter into the desired state.

This is of particular important because every component we've described in the Kubernetes Master and those that we'll investigate in the nodes below are stateless, which means their state is stored elsewhere, and that elsewhere is etcd.

Kubernetes doesn't take specific action to make things happen on the cluster; the Kubernetes API, via the API server, writes into etcd what should be true, and then the various pieces of Kubernetes make it so. etcd provides this interface via a simple HTTP/JSON API, which makes interacting with it quite simple.

etcd is also important in considerations of the Kubernetes security model due to it existing at a very low layer of the Kubernetes system, which means that any component that can write data to etcd has `root` to the cluster. Later on, we'll look into how the Kubernetes system is divided into layers in order to minimize this exposure. You can consider etcd to underlay Kubernetes with other parts of the ecosystem such as the container runtime, an image registry, a file storage, a cloud provider interface, and other dependencies that Kubernetes manages but does not have an opinionated perspective on.

In non-production Kubernetes clusters, you'll see single-node instantiations of etcd to save money on compute, simplify operations, or otherwise reduce complexity. It is essential to note however that a multi-master strategy of *2n+1* nodes is essential for production-ready clusters, in order to replicate data effectively across masters and ensure fault tolerance. It is recommended that you check the etcd documentation for more information.

Check out the etcd documentation here: `https://github.com/coreos/etcd/blob/master/Documentation/docs.md`.

If you're in front of your cluster, you can check to see the status of etcd by checking `componentstatuses` or `cs`:

```
[node3 /]$ kubectl get componentstatuses
NAME                 STATUS MESSAGE          ERROR
scheduler            Healthy ok
controller-manager   Healthy ok
etcd-0               Healthy {"health": "true"}
```

Due to a bug in the AKS ecosystem, this will currently not work on Azure. You can track this issue here to see when it is resolved:

`https://github.com/Azure/AKS/issues/173`: `kubectl get componentstatus fails for scheduler and controller-manager #173`

If you were to see an unhealthy `etcd` service, it'd look something like so:

```
[node3 /]$ kubectl get cs

NAME                 STATUS      MESSAGE      ERROR
etcd-0               Unhealthy                Get
http://127.0.0.1:2379/health: dial tcp 127.0.0.1:2379: getsockopt:
connection refused
controller-manager   Healthy     ok
scheduler            Healthy     ok
```

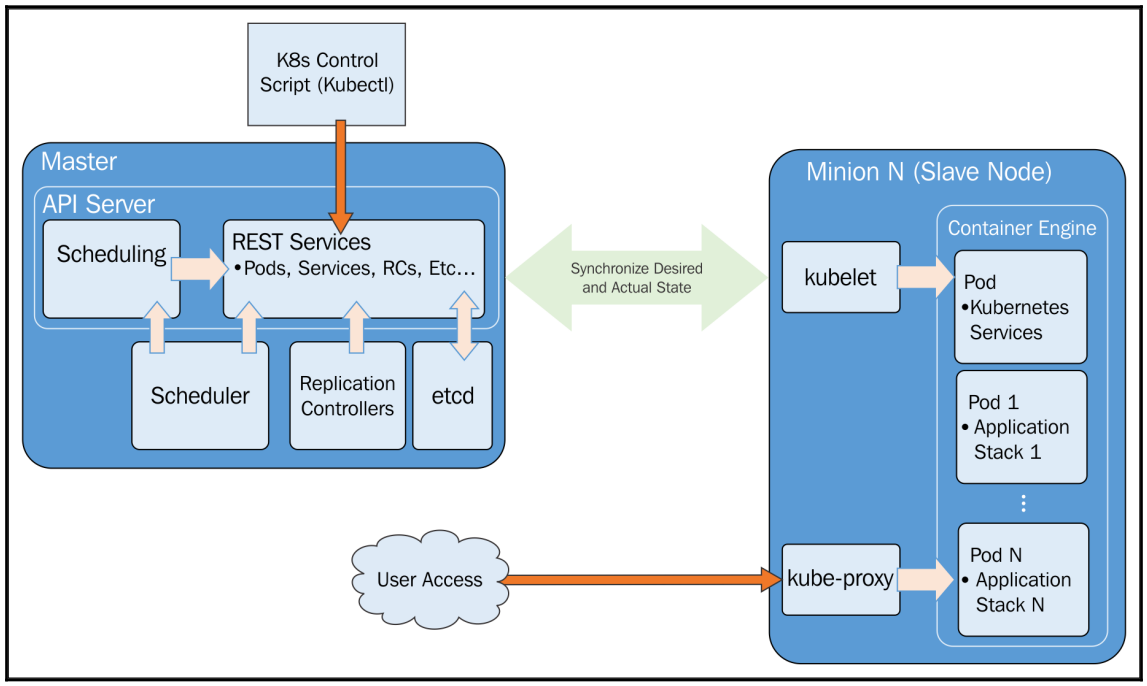
Cluster nodes

The third and final major Kubernetes component are the cluster nodes. While the master node components only run on a subset of the Kubernetes cluster, the node components run everywhere; they manage the maintenance of running pods, containers, and other primitives and provide the runtime environment. There are three node components:

- Kubelet
- Kube-proxy
- Container runtime

We'll dig into the specifics of these components later, but it's important to note several things about node componentry first. The kubelet can be considered the primary controller within Kubernetes, and providers the pod/node APIs that are used by the container runtime to execute container functionality. This functionality is grouped by container and their corresponding storage volumes into the concept of pods. The concept of a pod gives application developers a straightforward packaging paradigm from which to design their application, and allows us to take maximum advantage of the portability of containers, while realizing the power of orchestration and scheduling across many instances of a cluster.

It's interesting to note that a number of Kubernetes components run on Kubernetes itself (in other words, powered by the kubelets), including DNS, ingress, the Dashboard, and the resource monitoring of Heapster:



Kubernetes core architecture

In the preceding diagram, we see the core architecture of Kubernetes. Most administrative interactions are done via the `kubectl` script and/or RESTful service calls to the API.

As mentioned, note the ideas of the desired state and actual state carefully. This is the key to how Kubernetes manages the cluster and its workloads. All the pieces of K8s are constantly working to monitor the current actual state and synchronize it with the desired state defined by the administrators via the API server or `kubectl` script. There will be times when these states do not match up, but the system is always working to reconcile the two.

Let's dig into more detail on the Master and node instances.

Master

We know now that the **Master** is the brain of our cluster. We have the core API server, which maintains RESTful web services for querying and defining our desired cluster and workload state. It's important to note that the control pane only accesses the master to initiate changes and not the nodes directly.

Additionally, the master includes the **scheduler.** The replication controller/replica set works with the API server to ensure that the correct number of pod replicas are running at any given time. This is exemplary of the desired state concept. If our replication controller/replica set is defining three replicas and our actual state is two copies of the pod running, then the scheduler will be invoked to add a third pod somewhere in our cluster. The same is true if there are too many pods running in the cluster at any given time. In this way, K8s is always pushing toward that desired state.

As discussed previously, we'll look more closely into each of the Master components. `kube-apiserver` has the job of providing the API for the cluster as the front end of the control plane that the Master is providing. In fact, the apiserver is exposed through a service specifically called `kubernetes`, and we install the API server using the kubelet. This service is configured via the `kube-apiserver.yaml` file, which lives in `/etc/kubernetes/manifests/` on every manage node within your cluster.

`kube-apiserver` is a key portion of high availability in Kubernetes and, as such, it's designed to scale horizontally. We'll discuss how to construct highly available clusters later in this book, but suffice to say that you'll need to spread the `kube-apiserver` container across several Master nodes and provide a load balancer in the front.

Since we've gone into some detail about the cluster state store, it will suffice to say that an `etcd` agent is running on all of the Master nodes.

The next piece of the puzzle is `kube-scheduler`, which makes sure that all pods are associated and assigned to a node for operation. The schedulers works with the API server to schedule workloads in the form of pods on the actual minion nodes. These pods include the various containers that make up our application stacks. By default, the basic Kubernetes scheduler spreads pods across the cluster and uses different nodes for matching pod replicas. Kubernetes also allows specifying necessary resources, hardware and software policy constraints, affinity or anti-affinity as required, and data volume locality for each container, so scheduling can be altered by these additional factors.

The last two main pieces of the Master nodes are `kube-controller-manager` and `cloud-controller-manager`. As you might have guessed based on their names, while both of these services play an important part in container orchestration and scheduling, `kube-controller-manager` helps to orchestrate core internal components of Kubernetes, while `cloud-controller-manager` interacts with different vendors and their cloud provider APIs.

`kube-controller-manager` is actually a Kubernetes daemon that embeds the core control loops, otherwise known as controllers, that are included with Kubernetes:

- The **Node** controller, which manages pod availability and manages pods when they go down
- The **Replication** controller, which ensures that each replication controller object in the system has the correct number of pods
- The **Endpoints** controller, which controls endpoint records in the API, thereby managing DNS resolution of a pod or set of pods backing a service that defines selectors

In order to reduce the complexity of the controller components, they're all packed and shipped within this single daemon as `kube-controller-manager`.

`cloud-controller-manager`, on the other hand, pays attention to external components, and runs controller loops that are specific to the cloud provider that your cluster is using. The original intent of this design was to decouple the internal development of Kubernetes from cloud-specific vendor code. This was accomplished through the use of plugins, which prevents Kubernetes from relying on code that is not inherent to its value proposition. We can expect over time that future releases of Kubernetes will move vendor-specific code completely out of the Kubernetes code base, and that vendor-specific code will be maintained by the vendor themselves, and then called on by the Kubernetes `cloud-controller-manager`. This design prevents the need for several pieces of Kubernetes to communicate with the cloud provider, namely the kubelet, Kubernetes controller manager, and the API server.

Nodes (formerly minions)

In each node, we have several components as mentioned already. Let's look at each of them in detail.

The `kubelet` interacts with the API server to update the state and to start new workloads that have been invoked by the scheduler. As previously mentioned, this agent runs on every node of the cluster. The primary interface of the kubelet is one or more PodSpecs, which ensure that the containers and configurations are healthy.

The `kube-proxy` provides basic load balancing and directs the traffic destined for specific services to the proper pod on the backend. It maintains these network rules to enable the service abstraction through connection forwarding.

The last major component of the node is the container runtime, which is responsible for initiating, running, and stopping containers. The Kubernetes ecosystem has introduced the OCI runtime specification to democratize the container scheduler/orchestrator interface. While Docker, rkt, and runc are the current major implementations, the OCI aims to provide a common interface so you can bring your own runtime. At this point, Docker is the overwhelmingly dominant runtime.

Read more about the OCI runtime specifications here: `https://github.com/opencontainers/runtime-spec`.

In your cluster, the nodes may be virtual machines or bare metal hardware. Compared to other items such as controllers and pods, the node is not an abstraction that is created by Kubernetes. Rather, Kubernetes leverages `cloud-controller-manager` to interact with the cloud provider API, which owns the life cycle of the nodes. That means that when we instantiate a node in Kubernetes, we're simply creating an object that represents a machine in your given infrastructure. It's up to Kubernetes to determine if the node has converged with the object definition. Kubernetes validates the node's availability through its IP address, which is gathered via the `metadata.name` field. The status of these nodes can be discovered through the following status keys.

The addresses are where we'll find information such as the hostname and private and public IPs. This will be specific to your cloud provider's implementation. The condition field will give you a view into the state of your node's status in terms of disk, memory, network, and basic configuration.

Here's a table with the available node conditions:

Node Condition	Description
`OutOfDisk`	`True` if there is insufficient free space on the node for adding new pods, otherwise `False`
`Ready`	`True` if the node is healthy and ready to accept pods, `False` if the node is not healthy and is not accepting pods, and `Unknown` if the node controller has not heard from the node in the last 40 seconds
`MemoryPressure`	`True` if pressure exists on the node memory – that is, if the node memory is low; otherwise `False`
`DiskPressure`	`True` if pressure exists on the disk size – that is, if the disk capacity is low; otherwise `False`
`NetworkUnavailable`	`True` if the network for the node is not correctly configured, otherwise `False`
`ConfigOK`	`True` if the kubelet is correctly configured, otherwise `False`

A healthy node will have a status that looks similar to the following if you run it, you'll see the following output in the code:

```
$ kubectl get nodes -o json

"conditions": [
  {
    "type": "Ready",
    "status": "True"
  }
]
```

Capacity is simple: it's the available CPU, memory, and resulting number of pods that can be run on a given node. Nodes self-report their capacity and leave the responsibility for scheduling the appropriate number of resources to Kubernetes. The `Info` key is similarly straightforward and provides version information for Docker, OS, and Kubernetes.

It's important to note that the major component of the Kubernetes and node relationship is the **node controller**, which we called out previously as one of the core system controllers. There are three strategic pieces to this relationship:

- **Node health**: When you run large clusters in private, public, or hybrid cloud scenarios, you're bound to lose machines from time to time. Even within the data center, given a large enough cluster, you're bound to see regular failures at scale. The node controller is responsible for updating the node's `NodeStatus` to either `NodeReady` or `ConditionUnknown`, depending on the instance's availability. This management is key as Kubernetes will need to migrate pods (and therefore containers) to available nodes if `ConditionUnknown` occurs. You can set the health check interval for nodes in your cluster with `--node-monitor-period`.
- **IP assignment**: Every node needs some IP addresses, so it can distribute IPs to services and or containers.
- **Node list**: In order to manage pods across a number of machines, we need to keep an up-to-date list of available machines. Based on the aforementioned `NodeStatus`, the node controller will keep this list current.

We'll look into node controller specifics when investigating highly available clusters that span **Availability Zones** (**AZs**), which requires the spreading of nodes across AZs in order to provide availability.

Finally, we have some default pods, which run various infrastructure services for the node. As we explored briefly in the previous chapter, the pods include services for the **Domain Name System** (**DNS**), logging, and pod health checks. The default pod will run alongside our scheduled pods on every node.

In v1.0, minion was renamed to node, but there are still remnants of the term minion in some of the machine naming scripts and documentation that exists on the web. For clarity, I've added the term minion in addition to node in a few places throughout this book.

Core constructs

Now, let's dive a little deeper and explore some of the core abstractions Kubernetes provides. These abstractions will make it easier to think about our applications and ease the burden of life cycle management, high availability, and scheduling.

Pods

Pods allow you to keep related containers close in terms of the network and hardware infrastructure. Data can live near the application, so processing can be done without incurring a high latency from network traversal. Similarly, common data can be stored on volumes that are shared between a number of containers. Pods essentially allow you to logically group containers and pieces of our application stacks together.

While pods may run one or more containers inside, the pod itself may be one of many that is running on a Kubernetes node (minion). As we'll see, pods give us a logical group of containers across which we can then replicate, schedule, and balance service endpoints.

Pod example

Let's take a quick look at a pod in action. We'll spin up a Node.js application on the cluster. You'll need a GCE cluster running for this; if you don't already have one started, refer to the *Our first cluster* section in `Chapter 1`, *Introduction to Kubernetes*.

Now, let's make a directory for our definitions. In this example, I'll create a folder in the `/book-examples` subfolder under our home directory:

```
$ mkdir book-examples
$ cd book-examples
$ mkdir 02_example
$ cd 02_example
```

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

Use your favorite editor to create the following file and name it as `nodejs-pod.yaml`:

```
apiVersion: v1
kind: Pod
metadata:
  name: node-js-pod
spec:
  containers:
  - name: node-js-pod
    image: bitnami/apache:latest
    ports:
    - containerPort: 80
```

This file creates a pod named `node-js-pod` with the latest `bitnami/apache` container running on port `80`. We can check this using the following command:

```
$ kubectl create -f nodejs-pod.yaml
pod "node-js-pod" created
```

This gives us a pod running the specified container. We can see more information on the pod by running the following command:

```
$ kubectl describe pods/node-js-pod
```

You'll see a good deal of information, such as the pod's status, IP address, and even relevant log events. You'll note the pod IP address is a private IP address, so we cannot access it directly from our local machine. Not to worry, as the `kubectl exec` command mirrors Docker's `exec` functionality. You can get the pod IP address in a number of ways. A simple `get` of the pod will show you the IP where we use a template output that looks up the IP address in the status output:

```
$ kubectl get pod node-js-pod --template={{.status.podIP}}
```

You can use that IP address directly, or execute that command within backticks to `exec` into the pod. Once the pod shows it's in a running state, we can use this feature to run a command inside a pod:

```
$ kubectl exec node-js-pod -- curl <private ip address>

--or--

$ kubectl exec node-js-pod -- curl `kubectl get pod node-js-pod --
template={{.status.podIP}}`
```

By default, this runs a command in the first container it finds, but you can select a specific one using the `-c` argument.

After running the command, you should see some HTML code. We'll have a prettier view later in this chapter, but for now, we can see that our pod is indeed running as expected.

If you have experience with containers, you've probably also exec 'd into a container. You can do something very similar with Kubernetes:

```
master $ kubectl exec -it node-js-pod -- /bin/bash
root@node-js-pod:/opt/bitnami/apache/htdocs# exit
master $
```

You can also run other command directly into the container with the `exec` command. Note that you'll need to use two dashes to separate your command's argument in case it has the same in `kubectl`:

```
$ kubectl exec node-js-pod ls /
$ kubectl exec node-js-pod ps aux
$ kubectl exec node-js-pod -- uname -a
```

Labels

Labels give us another level of categorization, which becomes very helpful in terms of everyday operations and management. Similar to tags, labels can be used as the basis of service discovery as well as a useful grouping tool for day-to-day operations and management tasks. Labels are attached to Kubernetes objects and are simple key-value pairs. You will see them on pods, replication controllers, replica sets, services, and so on. Labels themselves and the keys/values inside of them are based on a constrained set of variables, so that queries against them can be evaluated efficiently using optimized algorithms and data structures.

The label indicates to Kubernetes which resources to work with for a variety of operations. Think of it as a filtering option. It is important to note that labels are meant to be meaningful and usable to the operators and application developers, but do not imply any semantic definitions to the cluster. Labels are used for organization and selection of subsets of objects, and can be added to objects at creation time and/or modified at any time during cluster operations. Labels are leveraged for management purposes, an example of which is when you want to know all of the backing containers for a particular service, you can normally get them via the labels on the container which correspond to the service at hand. With this type of management, you often end up with multiple labels on an object.

Kubernetes cluster management is often a cross-cutting operation, involving scaling up of different resources and services, management of multiple storage devices and dozens of nodes and is therefore a highly multi-dimensional operation.

Labels allow horizontal, vertical, and diagonal encapsulation of Kubernetes objects. You'll often see labels such as the following:

- `environment: dev`, `environment: integration`, `environment: staging`, `environment: UAT`, `environment: production`
- `tier: web`, `tier: stateless`, `tier: stateful`, `tier: protected`
- `tenancy: org1`, `tenancy: org2`

Once you've mastered labels, you can use selectors to identify a novel group of objects based on a particular set of label combination. There are currently equality-based and set-based selectors. Equality-based selectors allow operators to filter by keys/value pairs, and in order to `select(or)` an object, it must match all specified constraints. This kind of selector is often used to choose a particular node, perhaps to run against particularly speedy storage. Set-based selectors are more complex, and allow the operator to filter keys according to a specific value. This kind of selector is often used to determine where a object belongs, such as a tier, tenancy zone, or environment.

In short, an object may have many labels attached to it, but a selector can provide uniqueness to an object or set of objects.

We will take a look at labels in more depth later in this chapter, but first we will explore the remaining three constructs: services, replication controllers, and replica sets.

The container's afterlife

As Werner Vogels, CTO of AWS, famously said, *everything fails all the time;* containers and pods can and will crash, become corrupted, or maybe even just get accidentally shut off by a clumsy administrator poking around on one of the nodes. Strong policy and security practices such as enforcing least privilege curtail some of these incidents, but involuntary workload slaughter happens and is simply a fact of operations.

Luckily, Kubernetes provides two very valuable constructs to keep this somber affair all tidied up behind the curtains. Services and replication controllers/replica sets give us the ability to keep our applications running with little interruption and graceful recovery.

Services

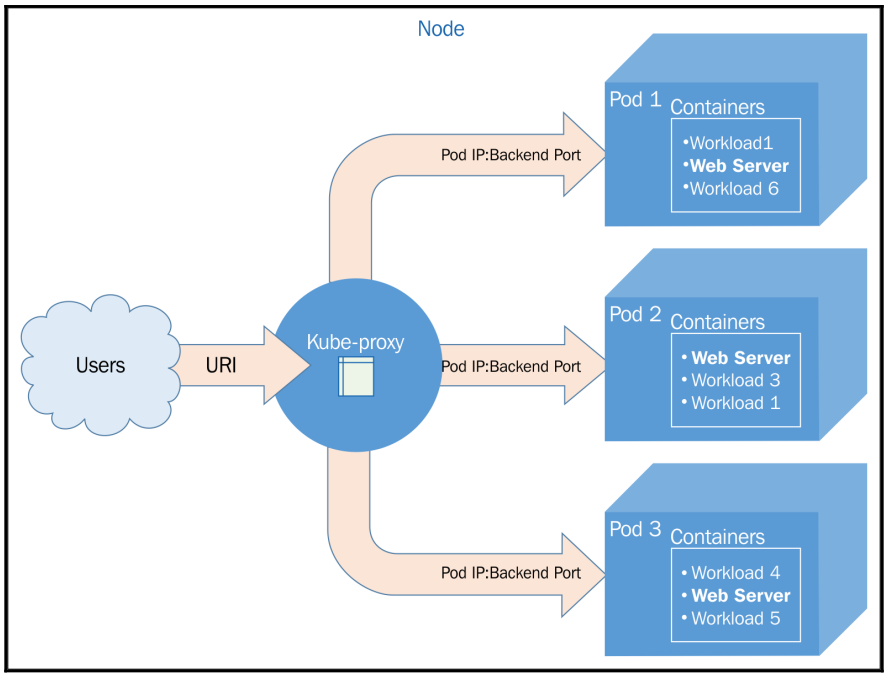
Services allow us to abstract access away from the consumers of our applications. Using a reliable endpoint, users and other programs can access pods running on your cluster seamlessly. This is in direct contradiction to one of our core Kubernetes constructs: pods.

Pods by definition are ephemeral and when they die they are not resurrected. If we trust that replication controllers will do their job to create and destroy pods as necessary, we'll need another construct to create a logical separation and policy for access.

Here we have services, which use a label selector to target a group of ever-changing pods. Services are important because we want frontend services that don't care about the specifics of backend services, and vice versa. While the pods that compose those tiers are fungible, the service via `ReplicationControllers` manages the relationships between objects and therefore decouples different types of applications.

For applications that require an IP address, there's a **Virtual IP** (VIP) available which can send round robin traffic to a backend pod. With cloud-native applications or microservices, Kubernetes provides the Endpoints API for simple communication between services.

K8s achieves this by making sure that every node in the cluster runs a proxy named `kube-proxy`. As the name suggests, the job of `kube-proxy` is to proxy communication from a service endpoint back to the corresponding pod that is running the actual application:



The kube-proxy architecture

Membership of the service load balancing pool is determined by the use of selectors and labels. Pods with matching labels are added to the list of candidates where the service forwards traffic. A virtual IP address and port are used as the entry points for the service, and the traffic is then forwarded to a random pod on a target port defined by either K8s or your definition file.

Updates to service definitions are monitored and coordinated from the K8s cluster Master and propagated to the `kube-proxy daemons` running on each node.

At the moment, `kube-proxy` is running on the node host itself. There are plans to containerize this and the kubelet by default in the future.

A service is a RESTful object, which relies on a `POST` transaction to the apiserver to create a new instance of the Kubernetes object. Here's an example of a simple service named `service-example.yaml`:

```
kind: Service
apiVersion: v1
metadata:
  name: gsw-k8s-3-service
spec:
  selector:
    app: gswk8sApp
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
```

This creates a service named `gsw-k8s-3-service`, which opens up a target port of `8080` with the key/value label of `app:gswk8sApp`. While the selector is continuously evaluated by a controller, the results of the IP address assignment (also called a cluster IP) will be posted to the endpoints object of `gsw-k8s-3-service`. The `kind` field is required, as is `ports`, while `selector` and `type` are optional.

Kube-proxy runs a number of other forms of virtual IP for services aside from the strategy outlined previously. There are three different types of proxy modes that we'll mention here, but will investigate in later chapters:

- Userspace
- Iptables
- Ipvs

Replication controllers and replica sets

Replication controllers have been deprecated in favor of using Deployments, which configure ReplicaSets. This method is a more robust manner of application replication, and has been developed as a response to the feedback of the container running community. We'll explore Deployments, Jobs, ReplicaSets, DaemonSets, and StatefulSets further in `Chapter 4`, *Implementing Reliable Container-Native Applications.* The following information is left here for reference.

Replication controllers (**RCs**), as the name suggests, manage the number of nodes that a pod and included container images run on. They ensure that an instance of an image is being run with the specific number of copies. RCs ensure that a pod or many same pods are always up and available to serve application traffic.

As you start to operationalize your containers and pods, you'll need a way to roll out updates, scale the number of copies running (both up and down), or simply ensure that at least one instance of your stack is always running. RCs create a high-level mechanism to make sure that things are operating correctly across the entire application and cluster. Pods created by RCs are replaced if they fail, and are deleted when terminated. RCs are recommended for use even if you only have a single pod in your application.

RCs are simply charged with ensuring that you have the desired scale for your application. You define the number of pod replicas you want running and give it a template for how to create new pods. Just like services, we'll use selectors and labels to define a pod's membership in an RC.

Kubernetes doesn't require the strict behavior of the replication controller, which is ideal for long-running processes. In fact, job controllers can be used for short-lived workloads, which allow jobs to be run to a completion state and are well suited for batch work.

Replica sets are a new type, currently in beta, that represent an improved version of replication controllers. Currently, the main difference consists of being able to use the new set-based label selectors, as we will see in the following examples.

Our first Kubernetes application

Before we move on, let's take a look at these three concepts in action. Kubernetes ships with a number of examples installed, but we'll create a new example from scratch to illustrate some of the concepts.

We already created a pod definition file but, as you learned, there are many advantages to running our pods via replication controllers. Again, using the `book-examples/02_example` folder we made earlier, we'll create some definition files and start a cluster of Node.js servers using a replication controller approach. Additionally, we'll add a public face to it with a load-balanced service.

Use your favorite editor to create the following file and name it as `nodejs-controller.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js
  labels:
    name: node-js
spec:
  replicas: 3
  selector:
    name: node-js
  template:
    metadata:
      labels:
        name: node-js
    spec:
      containers:
      - name: node-js
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
```

This is the first resource definition file for our cluster, so let's take a closer look. You'll note that it has four first-level elements (`kind`, `apiVersion`, `metadata`, and `spec`). These are common among all top-level Kubernetes resource definitions:

- `Kind`: This tells K8s the type of resource we are creating. In this case, the type is `ReplicationController`. The `kubectl` script uses a single `create` command for all types of resources. The benefit here is that you can easily create a number of resources of various types without the need for specifying individual parameters for each type. However, it requires that the definition files can identify what it is they are specifying.
- `apiVersion`: This simply tells Kubernetes which version of the schema we are using.

- `Metadata`: This is where we will give the resource a name and also specify labels that will be used to search and select resources for a given operation. The metadata element also allows you to create annotations, which are for the non-identifying information that might be useful for client tools and libraries.
- Finally, we have `spec`, which will vary based on the `kind` or `type` of resource we are creating. In this case, it's `ReplicationController`, which ensures the desired number of pods are running. The `replicas` element defines the desired number of pods, the `selector` element tells the controller which pods to watch, and finally, the `template` element defines a template to launch a new pod. The `template` section contains the same pieces we saw in our pod definition earlier. An important thing to note is that the `selector` values need to match the `labels` values specified in the pod template. Remember that this matching is used to select the pods being managed.

Now, let's take a look at the service definition named `nodejs-rc-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js
  labels:
    name: node-js
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    name: node-js
```

If you are using the free trial for Google Cloud Platform, you may have issues with the `LoadBalancer` type services. This type creates an external IP addresses, but trial accounts are limited to only one static address.

For this example, you won't be able to access the example from the external IP address using Minikube. In Kubernetes versions above 1.5, you can use Ingress to expose services but that is outside of the scope of this chapter.

The YAML here is similar to `ReplicationController`. The main difference is seen in the service `spec` element. Here, we define the `Service` type, listening `port`, and `selector`, which tell the `Service` proxy which pods can answer the service.

Kubernetes supports both YAML and JSON formats for definition files.

Create the Node.js express replication controller:

```
$ kubectl create -f nodejs-controller.yaml
```

The output is as follows:

```
replicationcontroller "node-js" created
```

This gives us a replication controller that ensures that three copies of the container are always running:

```
$ kubectl create -f nodejs-rc-service.yaml
```

The output is as follows:

```
service "node-js" created
```

On GCE, this will create an external load balancer and forwarding rules, but you may need to add additional firewall rules. In my case, the firewall was already open for port `80`. However, you may need to open this port, especially if you deploy a service with ports other than `80` and `443`.

OK, now we have a running service, which means that we can access the Node.js servers from a reliable URL. Let's take a look at our running services:

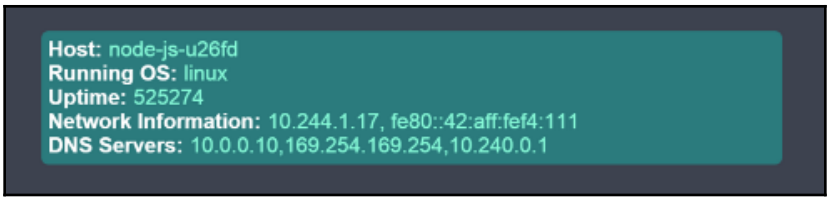
```
$ kubectl get services
```

The following screenshot is the result of the preceding command:

```
NAME         CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
kubernetes   10.0.0.1        <none>           443/TCP          11m
node-js      10.0.200.192    35.184.181.18    80:30874/TCP     4m
```

Services listing

In the preceding screenshot (services listing), we should note that the `node-js` service is running, and in the `IP(S)` column, we should have both a private and a public (`130.211.186.84` in the screenshot) IP address. If you don't see the external IP, you may need to wait a minute for the IP to be allocated from GCE. Let's see if we can connect by opening up the public address in a browser:



Container information application

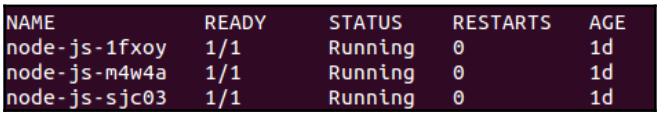
You should see something like the previous screenshot. If we visit multiple times, you should note that the container name changes. Essentially, the service load balancer is rotating between available pods on the backend.

Browsers usually cache web pages, so to really see the container name change, you may need to clear your cache or use a proxy like this one: `https://hide.me/en/proxy`.

Let's try playing chaos monkey a bit and kill off a few containers to see what Kubernetes does. In order to do this, we need to see where the pods are actually running. First, let's list our pods:

```
$ kubectl get pods
```

The following screenshot is the result of the preceding command:



NAME	READY	STATUS	RESTARTS	AGE
node-js-1fxoy	1/1	Running	0	1d
node-js-m4w4a	1/1	Running	0	1d
node-js-sjc03	1/1	Running	0	1d

Currently running pods

Now, let's get some more details on one of the pods running a `node-js` container. You can do this with the `describe` command and one of the pod names listed in the last command:

```
$ kubectl describe pod/node-js-sjc03
```

The following screenshot is the result of the preceding command:

```
Name:                           node-js-sjc03
Namespace:                      default
Image(s):                       petegoo/node-express-sample:latest
Node:                           kubernetes-minion-aqdf/10.240.142.178
Labels:                         name=node-js
Status:                         Running
Reason:
Message:
IP:                             10.244.0.10
Replication Controllers:        node-js (3/3 replicas created)
Containers:
  node-js:
    Image:      petegoo/node-express-sample:latest
    Limits:
      cpu:              100m
    State:              Running
      Started:          Tue, 28 Jul 2015 16:57:33 -0400
    Ready:              True
    Restart Count:      0
Conditions:
  Type          Status
  Ready         True
No events.
```

Pod description

You should see the preceding output. The information we need is the `Node:` section. Let's use the node name to **SSH** (short for **Secure Shell**) into the node (minion) running this workload:

```
$ gcloud compute --project "<Your project ID>" ssh --zone "<your gce zone>"
"<Node from
pod describe>"
```

Once SSHed into the node, if we run the `sudo docker ps` command, we should see at least two containers: one running the `pause` image and one running the actual `node-express-info` image. You may see more if K8s scheduled more than one replica on this node. Let's grab the container ID of the `jonbaier/node-express-info` image (not `gcr.io/google_containers/pause`) and kill it off to see what happens. Save this container ID somewhere for later:

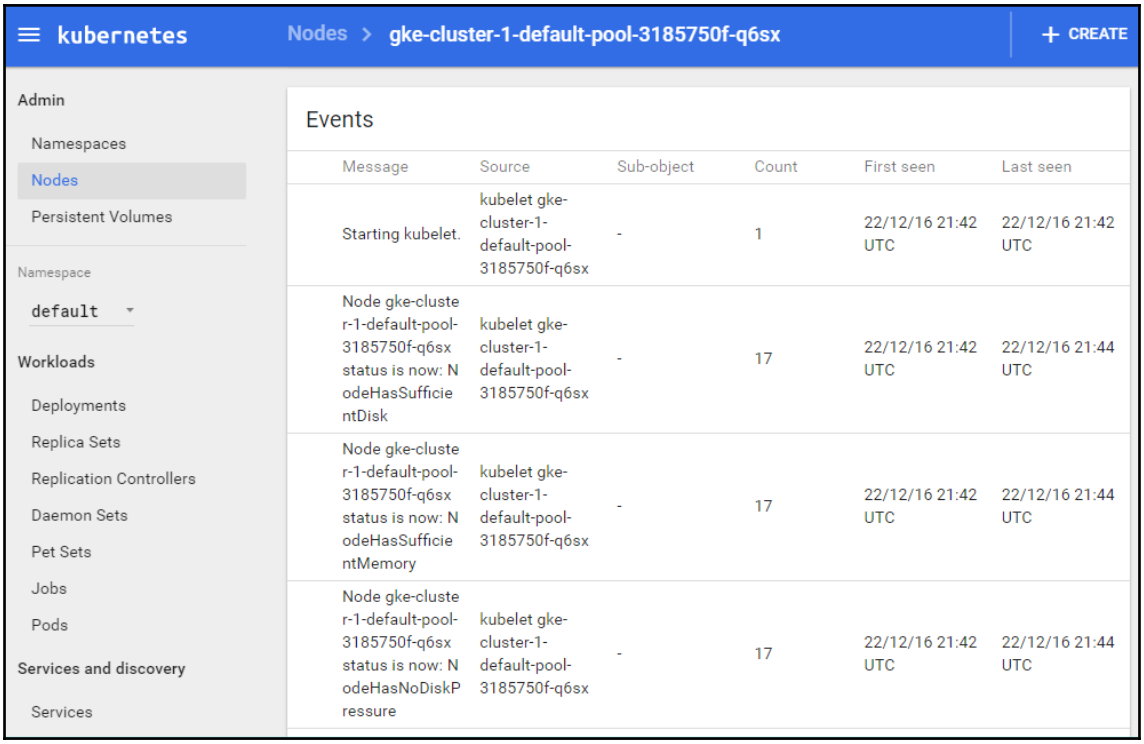
```
$ sudo docker ps --filter="name=node-js"
$ sudo docker stop <node-express container id>
$ sudo docker rm <container id>
$ sudo docker ps --filter="name=node-js"
```

Unless you are really quick, you'll probably note that there is still a `node-express-info` container running, but look closely and you'll note that `container id` is different and the creation timestamp shows only a few seconds ago. If you go back to the service URL, it is functioning as normal. Go ahead and exit the SSH session for now.

Here, we are already seeing Kubernetes playing the role of on-call operations, ensuring that our application is always running.

Let's see if we can find any evidence of the outage. Go to the **Events** page in the Kubernetes UI. You can find it by navigating to the **Nodes** page on the main K8s dashboard. Select a node from the list (the same one that we SSHed into) and scroll down to **Events** on the node details page.

You'll see a screen similar to the following screenshot:



Kubernetes UI event page

You should see three recent events. First, Kubernetes pulls the image. Second, it creates a new container with the pulled image. Finally, it starts that container again. You'll note that, from the timestamps, this all happens in less than a second. Time taken may vary based on the cluster size and image pulls, but the recovery is very quick.

More on labels

As mentioned previously, labels are just simple key-value pairs. They are available on pods, replication controllers, replica sets, services, and more. If you recall our service YAML `nodejs-rc-service.yaml`, there was a `selector` attribute. The `selector` attribute tells Kubernetes which labels to use in finding pods to forward traffic for that service.

K8s allows users to work with labels directly on replication controllers, replica sets, and services. Let's modify our replicas and services to include a few more labels. Once again, use your favorite editor to create these two files and name it as `nodejs-labels-controller.yaml` and `nodejs-labels-service.yaml`, as follows:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-labels
  labels:
    name: node-js-labels
    app: node-js-express
    deployment: test
spec:
  replicas: 3
  selector:
    name: node-js-labels
    app: node-js-express
    deployment: test
  template:
    metadata:
      labels:
        name: node-js-labels
        app: node-js-express
        deployment: test
    spec:
      containers:
      - name: node-js-labels
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-labels
  labels:
    name: node-js-labels
    app: node-js-express
    deployment: test
spec:
  type: LoadBalancer
  ports:
  - port: 80
  selector:
    name: node-js-labels
    app: node-js-express
    deployment: test
```

Create the replication controller and service as follows:

```
$ kubectl create -f nodejs-labels-controller.yaml
$ kubectl create -f nodejs-labels-service.yaml
```

Let's take a look at how we can use labels in everyday management. The following table shows us the options to select labels:

Operators	Description	Example
= or ==	You can use either style to select keys with values equal to the string on the right	`name = apache`
`!=`	Select keys with values that do not equal the string on the right	`Environment != test`
`in`	Select resources whose labels have keys with values in this set	`tier in (web, app)`
`notin`	Select resources whose labels have keys with values not in this set	`tier notin (lb, app)`
`<Key name>`	Use a key name only to select resources whose labels contain this key	`tier`

Label selectors

Let's try looking for replicas with `test` deployments:

```
$ kubectl get rc -l deployment=test
```

The following screenshot is the result of the preceding command:

```
NAME              DESIRED   CURRENT   READY     AGE
node-js-labels    3         3         3         46s
```

Replication controller listing

You'll notice that it only returns the replication controller we just started. How about services with a label named `component`? Use the following command:

```
$ kubectl get services -l component
```

The following screenshot is the result of the preceding command:

```
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   10.0.0.1     <none>        443/TCP   5d
```

Listing of services with a label named component

Here, we see the core Kubernetes service only. Finally, let's just get the `node-js` servers we started in this chapter. See the following command:

```
$ kubectl get services -l "name in (node-js,node-js-labels)"
```

The following screenshot is the result of the preceding command:

```
NAME             CLUSTER-IP     EXTERNAL-IP       PORT(S)          AGE
node-js          10.0.13.62     104.197.124.230   80:30798/TCP     14h
node-js-labels   10.0.207.25    104.154.54.104    80:31315/TCP     1m
```

Listing of services with a label name and a value of node-js or node-js-labels

Additionally, we can perform management tasks across a number of pods and services. For example, we can kill all replication controllers that are part of the `demo` deployment (if we had any running), as follows:

```
$ kubectl delete rc -l deployment=demo
```

Otherwise, kill all services that are part of a `production` or `test` deployment (again, if we have any running), as follows:

```
$ kubectl delete service -l "deployment in (test, production)"
```

It's important to note that, while label selection is quite helpful in day-to-day management tasks, it does require proper deployment hygiene on our part. We need to make sure that we have a tagging standard and that it is actively followed in the resource definition files for everything we run on Kubernetes.

While we used service definition YAML files to create our services so far, you can actually create them using a `kubectl` command only. To try this out, first run the `get pods` command and get one of the `node-js` pod names. Next, use the following `expose` command to create a service endpoint for just that pod:

```
$ kubectl expose pods node-js-gxkix --port=80 --
name=testing-vip --type=LoadBalancer
```

This will create a service named `testing-vip` and also a public `vip` (load balancer IP) that can be used to access this pod over port `80`.

There are number of other optional parameters that can be used. These can be found with the following command: `kubectl expose --help`.

Replica sets

As discussed earlier, replica sets are the new and improved version of replication controllers. Here's a basic example of their functionality, which we'll expand further in `Chapter 4`, *Implementing Reliable Container-Native Applications*, with advanced concepts. Here is an example of `ReplicaSet` based on and similar to the `ReplicationController`. Name this file as `nodejs-labels-replicaset.yaml`:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: node-js-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      app: node-js-express
      deployment: test
    matchExpressions:
      - {key: name, operator: In, values: [node-js-rs]}
  template:
```

```
    metadata:
      labels:
        name: node-js-rs
        app: node-js-express
        deployment: test
    spec:
      containers:
      - name: node-js-rs
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
```

Health checks

Kubernetes provides three layers of health checking. First, in the form of HTTP or TCP checks, K8s can attempt to connect to a particular endpoint and give a status of healthy on a successful connection. Second, application-specific health checks can be performed using command-line scripts. We can also use the `exec` container to run a health check from within your container. Anything that exits with a `0` status will be considered healthy.

Let's take a look at a few health checks in action. First, we'll create a new controller named `nodejs-health-controller.yaml` with a health check:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js
  labels:
    name: node-js
spec:
  replicas: 3
  selector:
    name: node-js
  template:
    metadata:
      labels:
        name: node-js
    spec:
      containers:
      - name: node-js
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
        livenessProbe:
          # An HTTP health check
```

```
    httpGet:
      path: /status/
      port: 80
    initialDelaySeconds: 30
    timeoutSeconds: 1
```

Note the addition of the `livenessprobe` element. This is our core health check element. From here, we can specify `httpGet`, `tcpScoket`, or `exec`. In this example, we use `httpGet` to perform a simple check for a URI on our container. The probe will check the path and port specified and restart the pod if it doesn't successfully return.

Status codes between `200` and `399` are all considered healthy by the probe.

Finally, `initialDelaySeconds` gives us the flexibility to delay health checks until the pod has finished initializing. The `timeoutSeconds` value is simply the timeout value for the probe.

Let's use our new health check-enabled controller to replace the old `node-js` RC. We can do this using the `replace` command, which will replace the replication controller definition:

```
$ kubectl replace -f nodejs-health-controller.yaml
```

Replacing the RC on its own won't replace our containers because it still has three healthy pods from our first run. Let's kill off those pods and let the updated `ReplicationController` replace them with containers that have health checks:

```
$ kubectl delete pods -l name=node-js
```

Now, after waiting a minute or two, we can list the pods in an RC and grab one of the pod IDs to inspect it a bit deeper with the `describe` command:

```
$ kubectl describe rc/node-js
```

The following screenshot is the result of the preceding command:

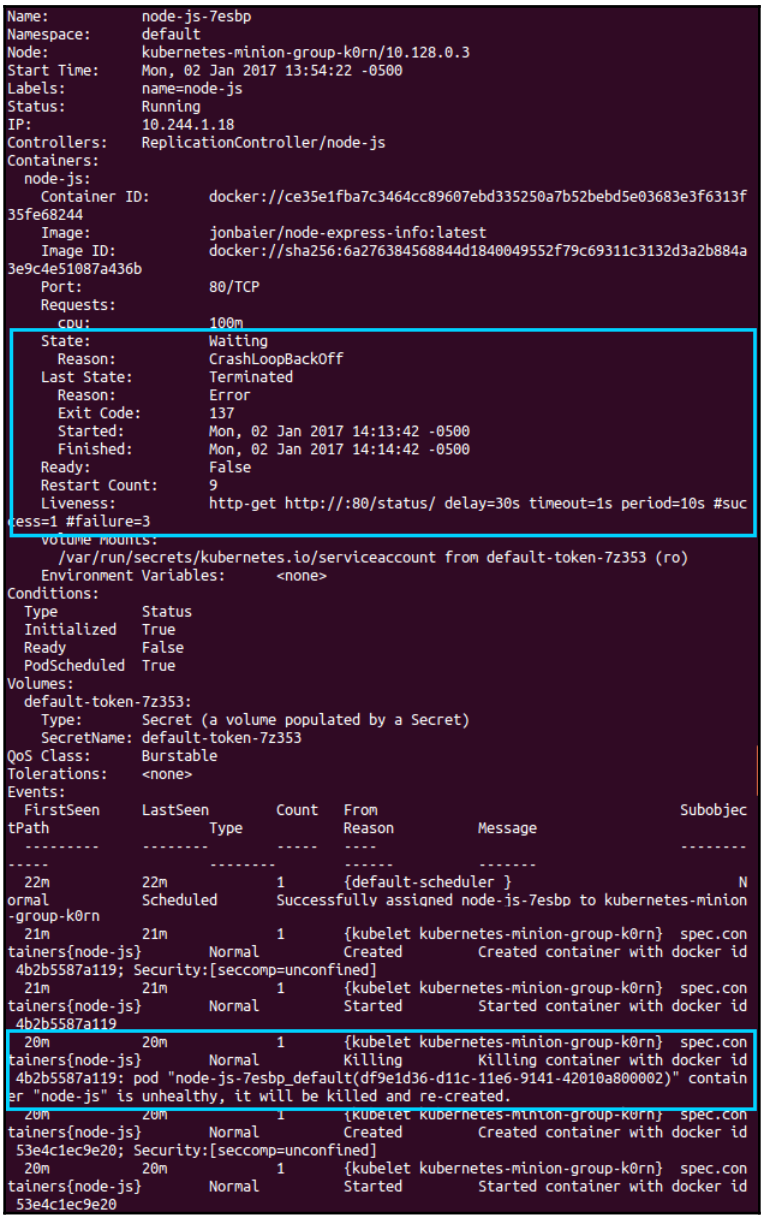
```
Name:           node-js
Namespace:      default
Image(s):       jonbaier/node-express-info:latest
Selector:       name=node-js
Labels:         name=node-js
Replicas:       3 current / 3 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen        Count   From                        SubobjectPath   Type
 Reason                 Message
  ---------     --------        -----   ----                        -------------   ------
--      ------                  -------
  42s           42s             1       {replication-controller }                   Normal
SuccessfulCreate        Created pod: node-js-7esbp
  42s           42s             1       {replication-controller }                   Normal
SuccessfulCreate        Created pod: node-js-istu0
  42s           42s             1       {replication-controller }                   Normal
SuccessfulCreate        Created pod: node-js-im7jw
```

Description of node-js replication controller

Now, use the following command for one of the pods:

```
$ kubectl describe pods/node-js-7esbp
```

The following screenshot is the result of the preceding command:



```
Name:           node-js-7esbp
Namespace:      default
Node:           kubernetes-minion-group-k0rn/10.128.0.3
Start Time:     Mon, 02 Jan 2017 13:54:22 -0500
Labels:         name=node-js
Status:         Running
IP:             10.244.1.18
Controllers:    ReplicationController/node-js
Containers:
  node-js:
    Container ID:       docker://ce35e1fba7c3464cc89607ebd335250a7b52bebd5e03683e3f6313f
35fe68244
    Image:              jonbaier/node-express-info:latest
    Image ID:           docker://sha256:6a276384568844d1840049552f79c69311c3132d3a2b884a
3e9c4e51087a436b
    Port:               80/TCP
    Requests:
      cpu:              100m
    State:              Waiting
      Reason:           CrashLoopBackOff
    Last State:         Terminated
      Reason:           Error
      Exit Code:        137
      Started:          Mon, 02 Jan 2017 14:13:42 -0500
      Finished:         Mon, 02 Jan 2017 14:14:42 -0500
    Ready:              False
    Restart Count:      9
    Liveness:           http-get http://:80/status/ delay=30s timeout=1s period=10s #suc
cess=1 #failure=3
    Volume Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7z353 (ro)
    Environment Variables:      <none>
Conditions:
  Type          Status
  Initialized   True
  Ready         False
  PodScheduled  True
Volumes:
  default-token-7z353:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-7z353
QoS Class:      Burstable
Tolerations:    <none>
Events:
  FirstSeen     LastSeen        Count   From                                    Subobjec
tPath                   Type            Reason          Message
  ---------     --------        -----   ----                                    --------
-----                   --------        ------          -------
  22m           22m             1       {default-scheduler }                    N
ormal           Scheduled       Successfully assigned node-js-7esbp to kubernetes-minion
-group-k0rn
  21m           21m             1       {kubelet kubernetes-minion-group-k0rn}  spec.con
tainers{node-js}        Normal          Created         Created container with docker id
 4b2b5587a119; Security:[seccomp=unconfined]
  21m           21m             1       {kubelet kubernetes-minion-group-k0rn}  spec.con
tainers{node-js}        Normal          Started         Started container with docker id
 4b2b5587a119
  20m           20m             1       {kubelet kubernetes-minion-group-k0rn}  spec.con
tainers{node-js}        Normal          Killing         Killing container with docker id
 4b2b5587a119: pod "node-js-7esbp_default(df9e1d36-d11c-11e6-9141-42010a800002)" contain
er "node-js" is unhealthy, it will be killed and re-created.
  20m           20m             1       {kubelet kubernetes-minion-group-k0rn}  spec.con
tainers{node-js}        Normal          Created         Created container with docker id
 53e4c1ec9e20; Security:[seccomp=unconfined]
  20m           20m             1       {kubelet kubernetes-minion-group-k0rn}  spec.con
tainers{node-js}        Normal          Started         Started container with docker id
 53e4c1ec9e20
```

Description of node-js-1m3cs pod

At the top, we'll see the overall pod details. Depending on your timing, under `State`, it will either show `Running` or `Waiting` with a `CrashLoopBackOff` reason and some error information. A bit below that, we can see information on our `Liveness` probe and we will likely see a failure count above `0`. Further down, we have the pod events. Again, depending on your timing, you are likely to have a number of events for the pod. Within a minute or two, you'll note a pattern of killing, started, and created events repeating over and over again. You should also see a note in the `Killing` entry that the container is unhealthy. This is our health check failing because we don't have a page responding at `/status`.

You may note that if you open a browser to the service load balancer address, it still responds with a page. You can find the load balancer IP with a `kubectl get services` command.

This is happening for a number of reasons. First, the health check is simply failing because `/status` doesn't exist, but the page where the service is pointed is still functioning normally between restarts. Second, the `livenessProbe` is only charged with restarting the container on a health check fail. There is a separate `readinessProbe` that will remove a container from the pool of pods answering service endpoints.

Let's modify the health check for a page that does exist in our container, so we have a proper health check. We'll also add a readiness check and point it to the nonexistent status page. Open the `nodejs-health-controller.yaml` file and modify the `spec` section to match the following listing and save it as `nodejs-health-controller-2.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js
  labels:
    name: node-js
spec:
  replicas: 3
  selector:
    name: node-js
  template:
    metadata:
      labels:
        name: node-js
    spec:
      containers:
      - name: node-js
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
        livenessProbe:
```

```
    # An HTTP health check
    httpGet:
      path: /
      port: 80
    initialDelaySeconds: 30
    timeoutSeconds: 1
  readinessProbe:
    # An HTTP health check
    httpGet:
      path: /status/
      port: 80
    initialDelaySeconds: 30
    timeoutSeconds: 1
```

This time, we'll delete the old RC, which will kill the pods with it, and create a new RC with our updated YAML file:

```
$ kubectl delete rc -l name=node-js-health
$ kubectl create -f nodejs-health-controller-2.yaml
```

Now, when we describe one of the pods, we only see the creation of the pod and the container. However, you'll note that the service load balancer IP no longer works. If we run the `describe` command on one of the new nodes, we'll note a `Readiness probe failed` error message, but the pod itself continues running. If we change the readiness probe path to `path: /`, we'll again be able to fulfill requests from the main service. Open up `nodejs-health-controller-2.yaml` in an editor and make that update now. Then, once again remove and recreate the replication controller:

```
$ kubectl delete rc -l name=node-js
$ kubectl create -f nodejs-health-controller-2.yaml
```

Now the load balancer IP should work once again. Keep these pods around as we will use them again in `Chapter 3`, *Networking, Load Balancers, and Ingress*.

TCP checks

Kubernetes also supports health checks via simple TCP socket checks and also with custom command-line scripts.

The following snippets are examples of what both use cases look like in the YAML file.

Health check using command-line script:

```
livenessProbe:
  exec:
    command:
    -/usr/bin/health/checkHttpServce.sh
  initialDelaySeconds:90
  timeoutSeconds: 1
```

Health check using simple TCP Socket connection:

```
livenessProbe:
  tcpSocket:
    port: 80
  initialDelaySeconds: 15
  timeoutSeconds: 1
```

Life cycle hooks or graceful shutdown

As you run into failures in real-life scenarios, you may find that you want to take additional action before containers are shut down or right after they are started. Kubernetes actually provides life cycle hooks for just this kind of use case.

The following example controller definition, `apache-hooks-controller.yaml`, defines both a `postStart` action and a `preStop` action to take place before Kubernetes moves the container into the next stage of its life cycle:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: apache-hook
  labels:
    name: apache-hook
spec:
  replicas: 3
  selector:
    name: apache-hook
  template:
    metadata:
      labels:
        name: apache-hook
    spec:
      containers:
      - name: apache-hook
```

```
image: bitnami/apache:latest
ports:
- containerPort: 80
lifecycle:
  postStart:
    httpGet:
      path: http://my.registration-server.com/register/
      port: 80
  preStop:
    exec:
      command: ["/usr/local/bin/apachectl","-k","graceful-
      stop"]
```

You'll note that, for the `postStart` hook, we define an `httpGet` action, but for the `preStop` hook, we define an `exec` action. Just as with our health checks, the `httpGet` action attempts to make an HTTP call to the specific endpoint and port combination, while the `exec` action runs a local command in the container.

The `httpGet` and `exec` actions are both supported for the `postStart` and `preStop` hooks. In the case of `preStop`, a parameter named `reason` will be sent to the handler as a parameter. See the following table for valid values:

Reason parameter	Failure description
Delete	Delete command issued via `kubectl` or the API
Health	Health check fails
Dependency	Dependency failure such as a disk mount failure or a default infrastructure pod crash

Valid preStop reasons

Check out the references section here: `https://github.com/kubernetes/kubernetes/blob/release-1.0/docs/user-guide/container-environment.md#container-hooks`.

It's important to note that hook calls are delivered at least once. Therefore, any logic in the action should gracefully handle multiple calls. Another important note is that `postStart` runs before a pod enters its ready state. If the hook itself fails, the pod will be considered unhealthy.

Application scheduling

Now that we understand how to run containers in pods and even recover from failure, it may be useful to understand how new containers are scheduled on our cluster nodes.

As mentioned earlier, the default behavior for the Kubernetes scheduler is to spread container replicas across the nodes in our cluster. In the absence of all other constraints, the scheduler will place new pods on nodes with the least number of other pods belonging to matching services or replication controllers.

Additionally, the scheduler provides the ability to add constraints based on resources available to the node. Today, this includes minimum CPU and memory allocations. In terms of Docker, these use the CPU-shares and memory limit flags under the covers.

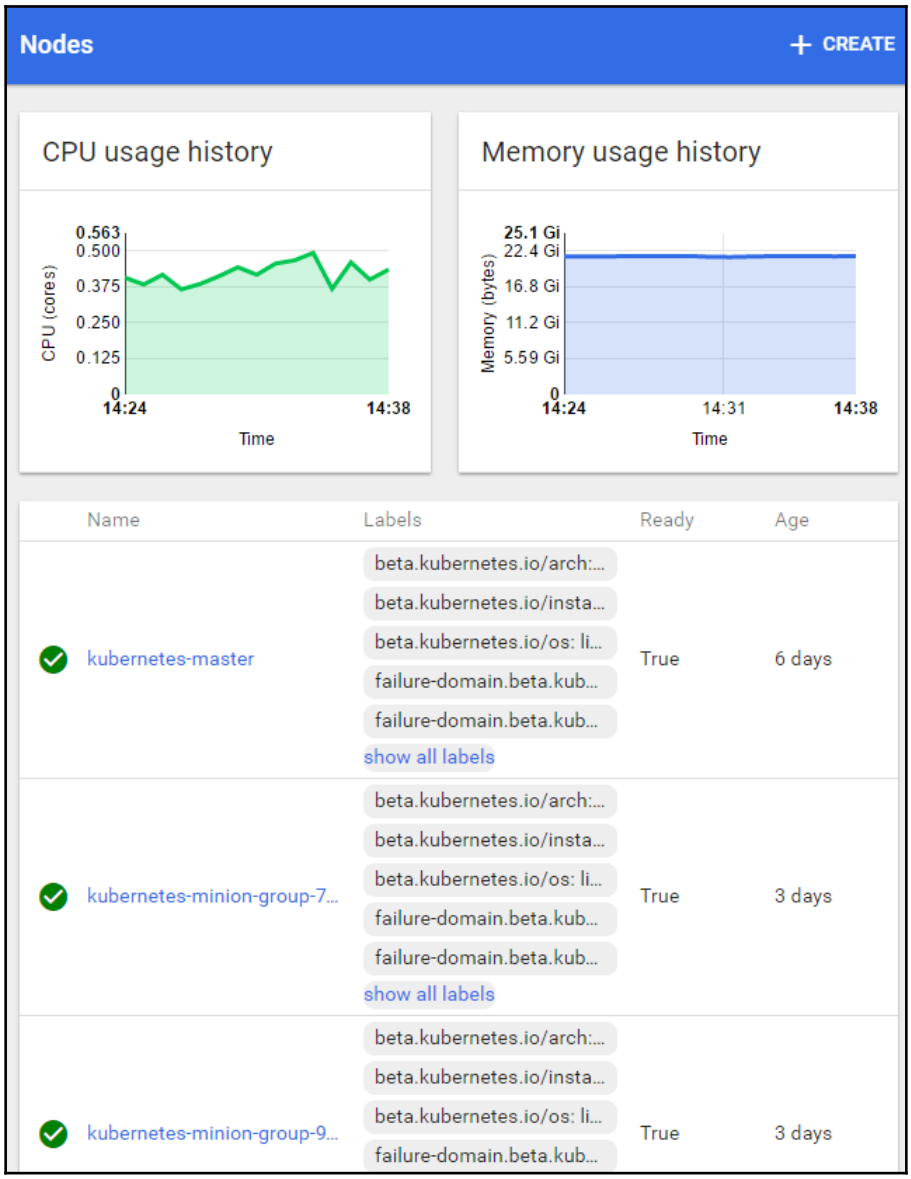
When additional constraints are defined, Kubernetes will check a node for available resources. If a node does not meet all the constraints, it will move to the next. If no nodes can be found that meet the criteria, then we will see a scheduling error in the logs.

The Kubernetes road map also has plans to support networking and storage. Because scheduling is such an important piece of overall operations and management for containers, we should expect to see many additions in this area as the project grows.

Scheduling example

Let's take a look at a quick example of setting some resource limits. If we look at our K8s dashboard, we can get a quick snapshot of the current state of resource usage on our cluster using `https://<your master ip>/api/v1/proxy/namespaces/kube-system/services/kubernetes-dashboard` and clicking on **Nodes** on the left-hand side menu.

We'll see a dashboard, as shown in the following screenshot:



Kube node dashboard

This view shows the aggregate CPU and memory across the whole cluster, nodes, and Master. In this case, we have fairly low CPU utilization, but a decent chunk of memory in use.

Let's see what happens when I try to spin up a few more pods, but this time, we'll request `512 Mi` for memory and `1500 m` for the CPU. We'll use `1500 m` to specify 1.5 CPUs; since each node only has 1 CPU, this should result in failure. Here's an example of the RC definition. Save this file as `nodejs-constraints-controller.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-constraints
  labels:
    name: node-js-constraints
spec:
  replicas: 3
  selector:
    name: node-js-constraints
  template:
    metadata:
      labels:
        name: node-js-constraints
    spec:
      containers:
      - name: node-js-constraints
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: "512Mi"
            cpu: "1500m"
```

To open the preceding file, use the following command:

```
$ kubectl create -f nodejs-constraints-controller.yaml
```

The replication controller completes successfully, but if we run a `get pods` command, we'll note the `node-js-constraints` pods are stuck in a pending state. If we look a little closer with the `describe pods/<pod-id>` command, we'll note a scheduling error (for `pod-id` use one of the pod names from the first command):

```
$ kubectl get pods
$ kubectl describe pods/<pod-id>
```

The following screenshot is the result of the preceding command:

```
Name:           node-js-constraints-n9dlx
Namespace:      default
Node:           /
Labels:         name=node-js-constraints
Status:         Pending
IP:
Controllers:    ReplicationController/node-js-constraints
Containers:
  node-js-constraints:
    Image:      jonbaier/node-express-info:latest
    Port:       80/TCP
    Limits:
      cpu:      1500m
      memory:   512Mi
    Requests:
      cpu:      1500m
      memory:   512Mi
    Volume Mounts:
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-7z353 (ro)
    Environment Variables:      <none>
Conditions:
  Type          Status
  PodScheduled  False
Volumes:
  default-token-7z353:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-7z353
QoS Class:      Guaranteed
Tolerations:    <none>
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath   Type    R
eason                   Message
  ---------     --------        -----   ----                    -------------   --------
------                  -------
  1m            1m              3       {default-scheduler }                    WarningF
ailedScheduling pod (node-js-constraints-n9dlx) failed to fit in any node
fit failure on node (kubernetes-minion-group-9zf7): Insufficient cpu
fit failure on node (kubernetes-minion-group-k0rn): Insufficient cpu
fit failure on node (kubernetes-minion-group-7th4): Insufficient cpu

  1m    1m      3       {default-scheduler }            Warning FailedScheduling        p
od (node-js-constraints-n9dlx) failed to fit in any node
fit failure on node (kubernetes-minion-group-7th4): Insufficient cpu
fit failure on node (kubernetes-minion-group-9zf7): Insufficient cpu
fit failure on node (kubernetes-minion-group-k0rn): Insufficient cpu

  1m    41s     2       {default-scheduler }            Warning FailedScheduling        p
od (node-js-constraints-n9dlx) failed to fit in any node
fit failure on node (kubernetes-minion-group-k0rn): Insufficient cpu
fit failure on node (kubernetes-minion-group-7th4): Insufficient cpu
fit failure on node (kubernetes-minion-group-9zf7): Insufficient cpu
```

Pod description

Note, in the bottom events section, that the `WarningFailedScheduling pod` error listed in `Events` is accompanied by `fit failure on node....Insufficient cpu` after the error. As you can see, Kuberneftes could not find a fit in the cluster that met all the constraints we defined.

If we now modify our CPU constraint down to `500 m`, and then recreate our replication controller, we should have all three pods running within a few moments.

Summary

We took a look at the overall architecture for Kubernetes, as well as the core constructs provided to build your services and application stacks. You should have a better understanding of how these abstractions make it easier to manage the life cycle of your stack and/or services as a whole and not just the individual components. Additionally, we took a first-hand look at how to manage some simple day-to-day tasks using pods, services, and replication controllers. We also looked at how to use Kubernetes to automatically respond to outages via health checks. Finally, we explored the Kubernetes scheduler and some of the constraints users can specify to influence scheduling placement.

In the next chapter, we'll dive into the networking layer of Kubernetes. We'll see how networking is done and also look at the core Kubernetes proxy that is used for traffic routing. We'll also look at service discovery and logical namespace groupings.

Questions

1. What are the three types of health checks?
2. What is the replacement technology for Replication Controllers?
3. Name all five layers of the Kubernetes system
4. Name two network plugins for Kubernetes
5. What are two of the options for container runtimes available to Kubernetes?
6. What are the three main components of the Kubernetes architecture?
7. Which type of selector filters keys and values according to a specific value?

Further reading

- Check out DevOps with Kubernetes: `https://www.packtpub.com/virtualization-and-cloud/devops-kubernetes`
- Mastering Kubernetes: `https://www.packtpub.com/virtualization-and-cloud/mastering-kubernetes`
- More information on labels: `https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/`
- More information on Replication Controllers: `https://kubernetes.io/docs/concepts/workloads/controllers/replicationcontroller/`

3
Working with Networking, Load Balancers, and Ingress

In this chapter, we will discuss Kubernetes' approach to cluster networking and how it differs from other approaches. We will describe key requirements for Kubernetes networking solutions and explore why these are essential for simplifying cluster operations. We will investigate DNS in the Kubernetes cluster, dig into the **Container Network Interface** (**CNI**) and plugin ecosystems, and will take a deeper dive into services and how the Kubernetes proxy works on each node. Finishing up, we will look at a brief overview of some higher level isolation features for multitenancy.

In this chapter, we will cover the following topics:

- Kubernetes networking
- Advanced services concepts
- Service discovery
- DNS, CNI, and ingress
- Namespace limits and quotas

Technical requirements

You'll need a running Kubernetes cluster like the one we created in the previous chapters. You'll also need access to deploy the cluster through the `kubectl` command.

The GitHub repository for this chapter can be found at `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter03`.

Container networking

Networking is a vital concern for production-level operations. At a service level, we need a reliable way for our application components to find and communicate with each other. Introducing containers and clustering into the mix makes things more complex as we now have multiple networking namespaces to bear in mind. Communication and discovery now becomes a feat that must navigate container IP space, host networking, and sometimes even multiple data center network topologies.

Kubernetes benefits here from getting its ancestry from the clustering tools used by Google for the past decade. Networking is one area where Google has outpaced the competition with one of the largest networks on the planet. Earlier, Google built its own hardware switches and **Software-defined Networking** (**SDN**) to give them more control, redundancy, and efficiency in their day-to-day network operations. Many of the lessons learned from running and networking two billion containers per week have been distilled into Kubernetes, and informed how K8s networking is done.

The Docker approach

In order to understand the motivation behind the K8s networking model, let's review Docker's approach to container networking.

Docker default networks

The following are some of Docker's default networks:

- **Bridge network**: In a nonswarm scenario, Docker will use the bridge network driver (called `bridge`) to allow standalone containers to speak to each other. You can think of the bridge as a link layer device that forwards network traffic between segments. If containers are connected to the same bridge network, they can communicate; if they're not connected, they can't. The bridged network is the default choice unless otherwise specified. In this mode, the container has its own networking namespace and is then bridged via virtual interfaces to the host (or node, in the case of K8s) network. In the bridged network, two containers can use the same IP range because they are completely isolated. Therefore, service communication requires some additional port mapping through the host side of network interfaces.

- **Host based**: Docker also offers host-based networking for standalone containers, which creates a virtual bridge called `docker0` that allocates private IP address space for the containers using that bridge. Each container gets a virtual Ethernet (`veth`) device that you can see in the container as `eth0`. Performance is greatly benefited since it removes a level of network virtualization; however, you lose the security of having an isolated network namespace. Additionally, port usage must be managed more carefully since all containers share an IP.

There's also a none network, which creates a container with no external interface. Only a `loopback` device is shown if you inspect the network interfaces.

In all of these scenarios, we are still on a single machine, and outside of host mode, the container IP space is not available outside that machine. Connecting containers across two machines requires NAT and port mapping for communication.

Docker user-defined networks

In order to address the cross-machine communication issue and allow greater flexibility, Docker also supports user-defined networks via network plugins. These networks exist independent of the containers themselves. In this way, containers can join the same existing networks. Through the new plugin architecture, various drivers can be provided for different network use cases such as the following:

- **Swarm**: In a clustered situation with Swarm, the default behavior is an overlay network, which allows you to connect multiple Docker daemons running on multiple machines. In order to coordinate across multiple hosts, all containers and daemons must all agree on the available networks and their topologies. Overlay networking introduces a significant amount of complexity with dynamic port mapping that Kubernetes avoids.

You can read more about overlay networks here: `https://docs.docker.com/network/overlay/`.

- **Macvlan**: Docker also provides macvlan addressing, which is most similar to the networking model that Kubernetes provides, as it assigns each Docker container a MAC address that makes it appear as a physical device on your network. Macvlan offers a more efficient network virtualization and isolation as it bypasses the Linux bridge. It is important to note that as of this book's publishing, Macvlan isn't supported in most cloud providers.

As a result of these options, Docker must manage complex port allocation on a per-machine basis for each host IP, and that information must be maintained and propagated to all other machines in the cluster. Docker users a gossip protocol to manage the forwarding and proxying of ports to other containers.

The Kubernetes approach

Kubernetes' approach to networking differs from the Docker's, so let's see how. We can learn about Kubernetes while considering four major topics in cluster scheduling and orchestration:

- Decoupling container-to-container communication by providing pods, not containers, with an IP address space
- Pod-to-pod communication and service as the dominant communication paradigm within the Kubernetes networking model
- Pod-to-service and external-to-service communications, which are provided by the `services` object

These considerations are a meaningful simplification for the Kubernetes networking model, as there's no dynamic port mapping to track. Again, IP addressing is scoped at the pod level, which means that networking in Kubernetes requires that each pod has its own IP address. This means that all containers in a given pod share that IP address, and are considered to be in the same network namespace. We'll explore how to manage this shared IP resource when we discuss internal and external services later in this chapter. Kubernetes facilitates the pod-to-pod communication by not allowing the use of **network address translation** (**NAT**) for container-to-container or container-to-node (minion) traffic. Furthermore, the internal container IP address must match the IP address that is used to communicate with it. This underlines the Kubernetes assumption that all pods are able to communicate with all other pods regardless of the host they've landed on, and that communication then informs routing within pods to a local IP address space that is provided to containers. All containers within a given host can communicate with each other on their reserved ports via localhost. This unNATed, flat IP space simplifies networking changes when you begin scaling to thousands of pods.

These rules keep much of the complexity out of our networking stack and ease the design of the applications. Furthermore, they eliminate the need to redesign network communication in legacy applications that are migrated from existing infrastructure. In greenfield applications, they allow for a greater scale in handling hundreds, or even thousands of services and application communications.

Astute readers may have also noticed that this creates a model that's backwards compatible with VMs and physical hosts that have a similar IP architecture as pods, with a single address per VM or physical host. This means you don't have to change your approach to service discovery, loadbalancing, application configuration, and port management, and can port over your application management workflows when working with Kubernetes.

K8s achieves this pod-wide IP magic using a pod container placeholder. Remember that the pause container that we saw in `Chapter 1`, *Introduction to Kubernetes*, in the *Services running on the master* section, is often referred to as a pod infrastructure container, and it has the important job of reserving the network resources for our application containers that will be started later on. Essentially, the pause container holds the networking namespace and IP address for the entire pod, and can be used by all the containers running within. The pause container joins first and holds the namespace while the subsequent containers in the pod join it when they start up using Docker's `--net=container:%ID%` function.

If you'd like to look over the code in the pause container, it's right here: `https://github.com/kubernetes/kubernetes/blob/master/build/pause/pause.c`.

Kubernetes can achieve the preceding feature set using either CNI plugins for production workloads or kubenet networking for simplified cluster communication. Kubernetes can also be used when your cluster is going to rely on logical partioning provided by a cloud service provider's security groups or **network access control lists** (**NACLs**). Let's dig into the specific networking options now.

Networking options

There are two approaches to the networking model that we have suggested. First, you can use one of the CNI plugins that exist in the ecosystem. This involves solutions that work with native networking layers of AWS, GCP, and Azure. There are also overlay-friendly plugins, which we'll cover in the next section. CNI is meant to be a common plugin architecture for containers. It's currently supported by several orchestration tools such as Kubernetes, Mesos, and CloudFoundry.

Network plugins are considered in alpha and therefore their capabilities, content, and configuration will change rapidly.

If you're looking for a simpler alternative for testing and using smaller clusters, you can use the kubenet plugin, which uses `bridge` and `host-local` CNI plugs with a straightforward implementation of `cbr0`. This plugin is only available on Linux, and doesn't provide any advanced features. As it's often used with the supplementation of a cloud provider's networking stance, it does not handle policies or cross-node networking.

Just as with CPU, memory, and storage, Kubernetes takes advantage of network namespaces, each with their own iptables rules, interfaces, and route tables. Kubernetes uses iptables and NAT to manage multiple logical addresses that sit behind a single physical address, though you have the option to provide your cluster with multiple physical interfaces (NICs). Most people will find themselves generating multiple logical interfaces and using technologies such as multiplexing, virtual bridges, and hardware switching using SR-IOV in order to create multiple devices.

You can find out more information at `https://github.com/containernetworking/cni`.

Always refer to the Kubernetes documentation for the latest and full list of supported networking options.

Networking comparisons

To get a better understanding of networking in containers, it can be instructive to look at the popular choices for container networking. The following approaches do not make an exhaustive list, but should give a taste of the options available.

Weave

Weave provides an overlay network for Docker containers. It can be used as a plugin with the new Docker network plugin interface, and it is also compatible with Kubernetes through a CNI plugin. Like many overlay networks, many criticize the performance impact of the encapsulation overhead. Note that they have recently added a preview release with **Virtual Extensible LAN** (**VXLAN**) encapsulation support, which greatly improves performance. For more information, visit `http://blog.weave.works/2015/06/12/weave-fast-datapath/`.

Flannel

Flannel comes from CoreOS and is an etcd-backed overlay. Flannel gives a full subnet to each host/node, enabling a similar pattern to the Kubernetes practice of a routable IP per pod or group of containers. Flannel includes an in-kernel VXLAN encapsulation mode for better performance and has an experimental multi-network mode similar to the overlay Docker plugin. For more information, visit `https://github.com/coreos/flannel`.

Project Calico

Project Calico is a layer 3-based networking model that uses the built-in routing functions of the Linux kernel. Routes are propagated to virtual routers on each host via **Border Gateway Protocol** (**BGP**). Calico can be used for anything from small-scale deploys to large internet-scale installations. Because it works at a lower level on the network stack, there is no need for additional NAT, tunneling, or overlays. It can interact directly with the underlying network infrastructure. Additionally, it has a support for network-level ACLs to provide additional isolation and security. For more information, visit `http://www.projectcalico.org/`.

Canal

Canal merges both Calico for the network policy and Flannel for the overlay into one solution. It supports both Calico and Flannel type overlays and uses the Calico policy enforcement logic. Users can choose from overlay and non-overlay options with this setup as it combines the features of the preceding two projects. For more information, visit `https://github.com/tigera/canal`.

Kube-router

Kube-router option is a purpose-built networking solution that aims to provide high performance that's easy to use. It's based on the Linux LVS/IPVS kernel load balancing technologies as proxy. It also uses kernel-based networking and uses iptables as a network policy enforcer. Since it doesn't use an overlay technology, it's potentially a high-performance option for the future. For more information, visit the following URL: `https://github.com/cloudnativelabs/kube-router`.

Balanced design

It's important to point out the balance that Kubernetes is trying to achieve by placing the IP at the pod level. Using unique IP addresses at the host level is problematic as the number of containers grows. Ports must be used to expose services on specific containers and allow external communication. In addition to this, the complexity of running multiple services that may or may not know about each other (and their custom ports) and managing the port space becomes a big issue.

However, assigning an IP address to each container can be overkill. In cases of sizable scale, overlay networks and NATs are needed in order to address each container. Overlay networks add latency, and IP addresses would be taken up by backend services as well since they need to communicate with their frontend counterparts.

Here, we really see an advantage in the abstractions that Kubernetes provides at the application and service level. If I have a web server and a database, we can keep them on the same pod and use a single IP address. The web server and database can use the local interface and standard ports to communicate, and no custom setup is required. Furthermore, services on the backend are not needlessly exposed to other application stacks running elsewhere in the cluster (but possibly on the same host). Since the pod sees the same IP address that the applications running within it see, service discovery does not require any additional translation.

If you need the flexibility of an overlay network, you can still use an overlay at the pod level. Weave, Flannel, and Project Calico can be used with Kubernetes as well as a plethora of other plugins and overlays that are available.

This is also very helpful in the context of scheduling the workloads. It is key to have a simple and standard structure for the scheduler to match constraints and understand where space exists on the cluster's network at any given time. This is a dynamic environment with a variety of applications and tasks running, so any additional complexity here will have rippling effects.

There are also implications for service discovery. New services coming online must determine and register an IP address on which the rest of the world, or at least a cluster, can reach them. If NAT is used, the services will need an additional mechanism to learn their externally facing IP.

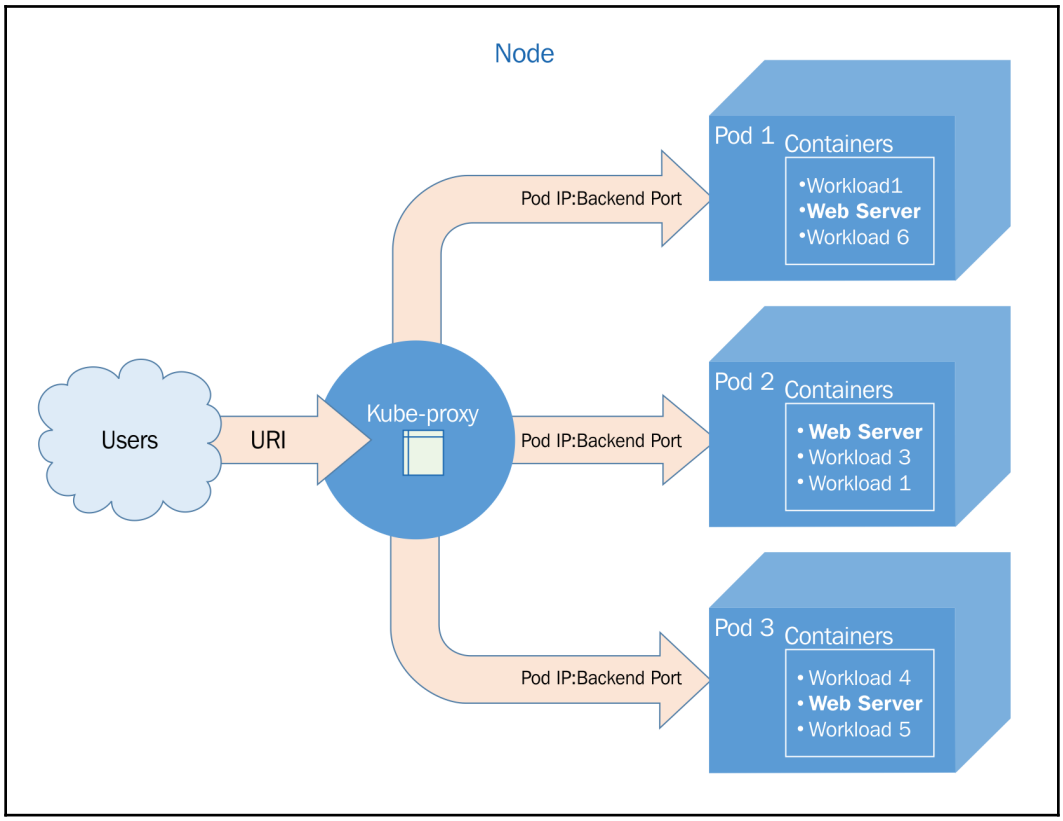
Advanced services

Let's explore the IP strategy as it relates to services and communication between containers. If you recall, in the *Services* section of `Chapter 2`, *Pods, Services, Replication Controllers, and Labels*, you learned that Kubernetes is using `kube-proxy` to determine the proper pod IP address and port serving each request. Behind the scenes, `kube-proxy` is actually using virtual IPs and iptables to make all this magic work.

`kube-proxy` now has two modes—*userspace* and *iptables*. As of now, 1.2 iptables is the default mode. In both modes, `kube-proxy` is running on every host. Its first duty is to monitor the API from the Kubernetes master. Any updates to services will trigger an update to iptables from `kube-proxy`. For example, when a new service is created, a virtual IP address is chosen and a rule in iptables is set, which will direct its traffic to `kube-proxy` via a random port. Thus, we now have a way to capture service-destined traffic on this node. Since `kube-proxy` is running on all nodes, we have cluster-wide resolution for the service **VIP** (short for **virtual IP**). Additionally, DNS records can point to this VIP as well.

In the userspace mode, we have a hook created in iptables, but the proxying of traffic is still handled by `kube-proxy`. The iptables rule is only sending traffic to the service entry in `kube-proxy` at this point. Once `kube-proxy` receives the traffic for a particular service, it must then forward it to a pod in the service's pool of candidates. It does this using a random port that was selected during service creation.

Refer to the following diagram for an overview of the flow:



Kube-proxy communication

It is also possible to always forward traffic from the same client IP to the same backend pod/container using the `sessionAffinity` element in your service definition.

In the iptables mode, the pods are coded directly in the iptable rules. This removes the dependency on `kube-proxy` for actually proxying the traffic. The request will go straight to iptables and then on to the pod. This is faster and removes a possible point of failure. Readiness probe, as we discussed in the *Health Check* section of `Chapter 2`, *Pods, Services, Replication Controllers, and Labels*, is your friend here as this mode also loses the ability to retry pods.

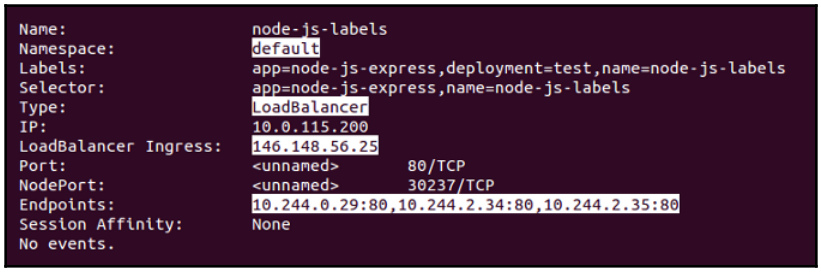
External services

In the previous chapter, we saw a few service examples. For testing and demonstration purposes, we wanted all the services to be externally accessible. This was configured by the `type: LoadBalancer` element in our service definition. The `LoadBalancer` type creates an external load balancer on the cloud provider. We should note that support for external load balancers varies by provider, as does the implementation. In our case, we are using GCE, so integration is pretty smooth. The only additional setup needed is to open firewall rules for the external service ports.

Let's dig a little deeper and do a `describe` command on one of the services from the *More on labels* section in `Chapter 2`, *Pods, Services, Replication Controllers, and Labels*:

```
$ kubectl describe service/node-js-labels
```

The following screenshot is the result of the preceding command:



Service description

In the output of the preceding screenshot, you'll note several key elements. Our `Namespace:` is set to `default`, the `Type:` is `LoadBalancer`, and we have the external IP listed under `LoadBalancer Ingress:`. Furthermore, we can see `Endpoints:`, which shows us the IPs of the pods that are available to answer service requests.

Internal services

Let's explore the other types of services that we can deploy. First, by default, services are only internally facing. You can specify a type of `clusterIP` to achieve this, but, if no type is defined, `clusterIP` is the assumed type. Let's take a look at an example, `nodejs-service-internal.yaml`; note the lack of the `type` element:

```
apiVersion: v1
kind: Service
```

```
  metadata:
    name: node-js-internal
    labels:
      name: node-js-internal
  spec:
    ports:
    - port: 80
    selector:
      name: node-js
```

Use this listing to create the service definition file. You'll need a healthy version of the `node-js` RC (Listing `nodejs-health-controller-2.yaml`). As you can see, the selector matches on the pods named `node-js` that our RC launched in the previous chapter. We will create the service and then list the currently running services with a filter as follows:

```
$ kubectl create -f nodejs-service-internal.yaml
$ kubectl get services -l name=node-js-internal
```

The following screenshot is the result of the preceding command:

```
NAME              LABELS                  SELECTOR      IP(S)        PORT(S)
node-js-internal  name=node-js-internal   name=node-js  10.0.5.134   80/TCP
```

Internal service listing

As you can see, we have a new service, but only one IP. Furthermore, the IP address is not externally accessible. We won't be able to test the service from a web browser this time. However, we can use the handy `kubectl exec` command and attempt to connect from one of the other pods. You will need `node-js-pod` (`nodejs-pod.yaml`) running. Then, you can execute the following command:

```
$ kubectl exec node-js-pod -- curl <node-js-internal IP>
```

This allows us to run a `docker exec` command as if we had a shell in the `node-js-pod` container. It then hits the internal service URL, which forwards to any pods with the `node-js` label.

If all is well, you should get the raw HTML output back. You have successfully created an internal-only service. This can be useful for backend services that you want to make available to other containers running in your cluster, but not open to the world at large.

Custom load balancing

A third type of service that K8s allows is the `NodePort` type. This type allows us to expose a service through the host or node (minion) on a specific port. In this way, we can use the IP address of any node (minion) and access our service on the assigned node port. Kubernetes will assign a node port by default in the range of `3000-32767`, but you can also specify your own custom port. In the example in the following listing `nodejs-service-nodeport.yaml`, we choose port `30001`, as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-nodeport
  labels:
    name: node-js-nodeport
spec:
  ports:
  - port: 80
    nodePort: 30001
  selector:
    name: node-js
  type: NodePort
```

Once again, create this YAML definition file and create your service, as follows:

```
$ kubectl create -f nodejs-service-nodeport.yaml
```

The output should have a message like this:

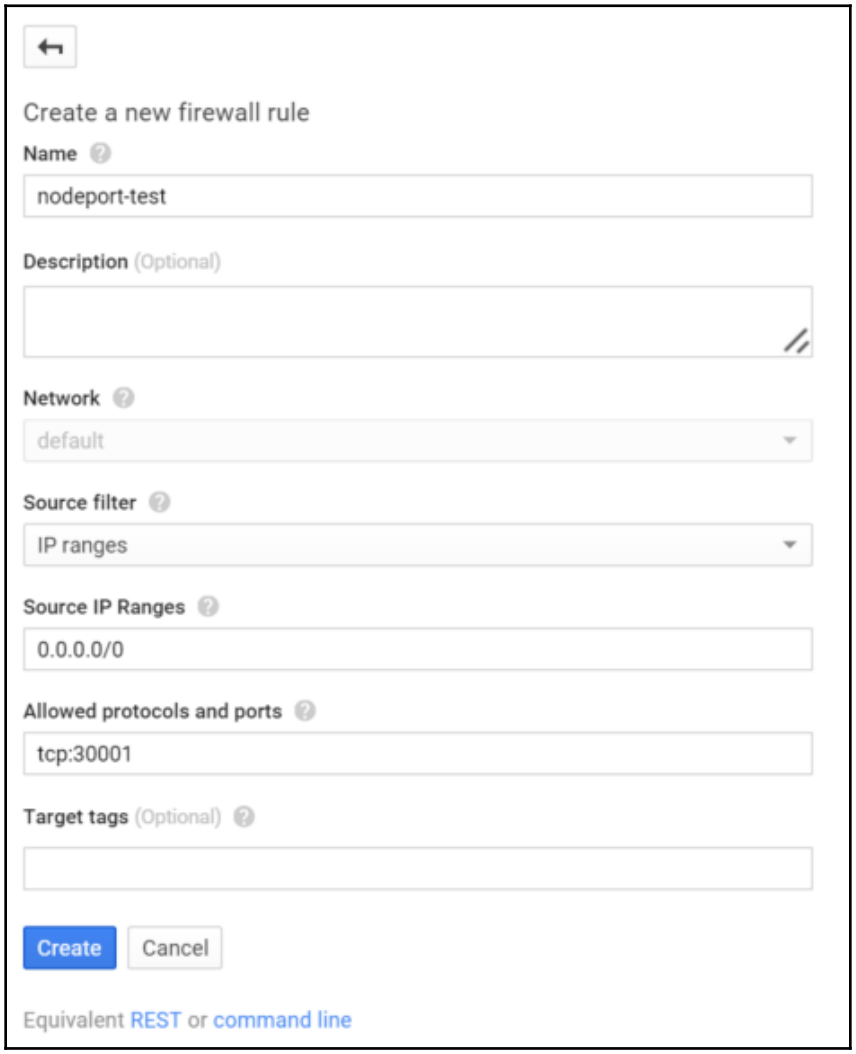
```
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet,
you may
need to set up firewall rules for the service port(s) (tcp:30001) to se
rve traffic.

See http://releases.k8s.io/HEAD/docs/user-guide/services-firewalls.md f
or more details.
services/node-js-nodeport
```

New GCP firewall rule

Note message about opening firewall ports. Similar to the external load balancer type, `NodePort` is exposing your service externally using ports on the nodes. This could be useful if, for example, you want to use your own load balancer in front of the nodes. Let's make sure that we open those ports on GCP before we test our new service.

From the GCE VM instance console, click on the details for any of your nodes (minions). Then, click on the network, which is usually the default unless otherwise specified during creation. In **Firewall rules**, we can add a rule by clicking on **Add firewall rule**.

Create a rule like the one shown in the following screenshot (`tcp:30001` on the `0.0.0.0/0` IP range):



Create a new firewall rule page

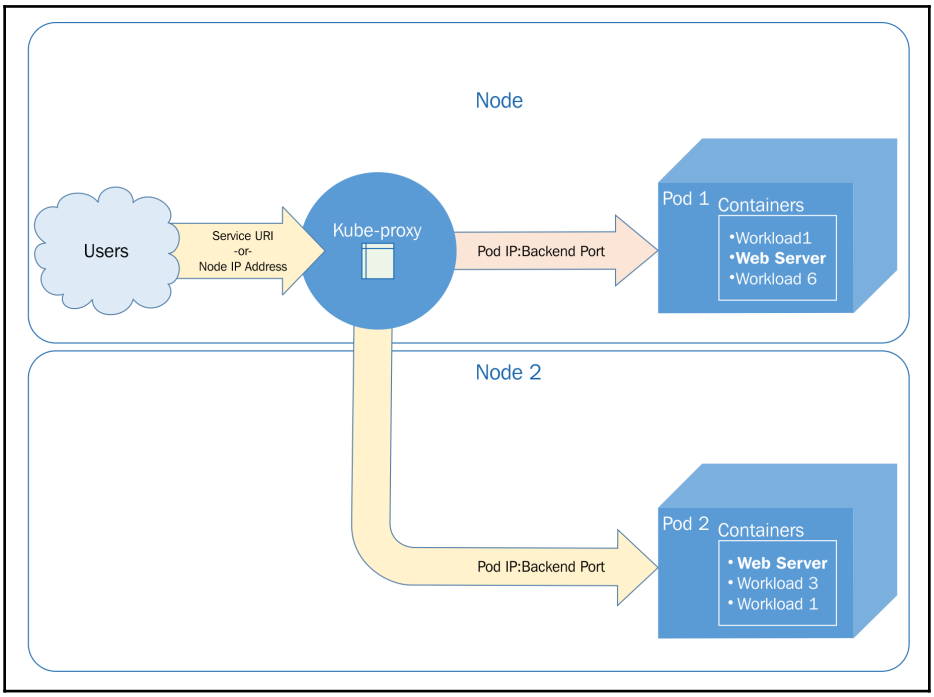
We can now test our new service by opening a browser and using an IP address of any node (minion) in your cluster. The format to test the new service is as follows:

```
http://<Minoion IP Address>:<NodePort>/
```

Finally, the latest version has added an `ExternalName` type, which maps a `CNAME` to the service.

Cross-node proxy

Remember that `kube-proxy` is running on all the nodes, so even if the pod is not running there, the traffic will be given a proxy to the appropriate host. Refer to the *Cross-node traffic* screenshot for a visual on how the traffic flows. A user makes a request to an external IP or URL. The request is serviced by **Node** in this case. However, the pod does not happen to run on this node. This is not a problem because the pod IP addresses are routable. So, `kube-proxy` or **iptables** simply passes traffic onto the pod IP for this service. The network routing then completes on **Node 2**, where the requested application lives:



Cross-node traffic

Custom ports

Services also allow you to map your traffic to different ports; then, the containers and pods expose themselves. We will create a service that exposes port `90` and forwards traffic to port `80` on the pods. We will call the `node-js-90` pod to reflect the custom port number. Create the following two definition files, `nodejs-customPort-controller.yaml` and `nodejs-customPort-service.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-90
  labels:
    name: node-js-90
spec:
  replicas: 3
  selector:
    name: node-js-90
  template:
    metadata:
      labels:
        name: node-js-90
    spec:
      containers:
      - name: node-js-90
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80

apiVersion: v1
kind: Service
metadata:
  name: node-js-90
  labels:
    name: node-js-90
spec:
  type: LoadBalancer
  ports:
  - port: 90
    targetPort: 80
  selector:
    name: node-js-90
```

If you are using the free trial for Google Cloud Platform, you may have issues with the `LoadBalancer` type services. This type creates multiple external IP addresses, but trial accounts are limited to only one static address.

You'll note that in the service definition, we have a `targetPort` element. This element tells the service the port to use for pods/containers in the pool. As we saw in previous examples, if you do not specify `targetPort`, it assumes that it's the same port as the service. This port is still used as the service port, but, in this case, we are going to expose the service on port `90` while the containers serve content on port `80`.

Create this RC and service and open the appropriate firewall rules, as we did in the last example. It may take a moment for the external load balancer IP to propagate to the `get service` command. Once it does, you should be able to open and see our familiar web application in a browser using the following format:

```
http://<external service IP>:90/
```

Multiple ports

Another custom port use case is that of multiple ports. Many applications expose multiple ports, such as HTTP on port `80` and port `8888` for web servers. The following example shows our app responding on both ports. Once again, we'll also need to add a firewall rule for this port, as we did for the list `nodejs-service-nodeport.yaml` previously. Save the listing as `nodejs-multi-controller.yaml` and `nodejs-multi-service.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-multi
  labels:
    name: node-js-multi
spec:
  replicas: 3
  selector:
    name: node-js-multi
  template:
    metadata:
      labels:
        name: node-js-multi
    spec:
```

```
      containers:
      - name: node-js-multi
        image: jonbaier/node-express-multi:latest
        ports:
        - containerPort: 80
        - containerPort: 8888

apiVersion: v1
kind: Service
metadata:
  name: node-js-multi
  labels:
    name: node-js-multi
spec:
  type: LoadBalancer
  ports:
  - name: http
    protocol: TCP
    port: 80
  - name: fake-admin-http
    protocol: TCP
    port: 8888
  selector:
    name: node-js-multi
```

The application and container itself must be listening on both ports for this to work. In this example, port `8888` is used to represent a fake admin interface. If, for example, you want to listen on port `443`, you would need a proper SSL socket listening on the server.

Ingress

We previously discussed how Kubernetes uses the service abstract as a means to proxy traffic to a backing pod that's distributed throughout our cluster. While this is helpful in both scaling and pod recovery, there are more advanced routing scenarios that are not addressed by this design.

To that end, Kubernetes has added an ingress resource, which allows for custom proxying and load balancing to a back service. Think of it as an extra layer or hop in the routing path before traffic hits our service. Just as an application has a service and backing pods, the ingress resource needs both an Ingress entry point and an ingress controller that perform the custom logic. The entry point defines the routes and the controller actually handles the routing. This is helpful for picking up traffic that would normally be dropped by an edge router or forwarded elsewhere outside of the cluster.

Ingress itself can be configured to offer externally addressable URLs for internal services, to terminate SSL, offer name-based virtual hosting as you'd see in a traditional web server, or load balance traffic. Ingress on its own cannot service requests, but requires an additional ingress controller to fulfill the capabilities outlined in the object. You'll see nginx and other load balancing or proxying technology involved as part of the controller framework. In the following examples, we'll be using GCE, but you'll need to deploy a controller yourself in order to take advantage of this feature. A popular option at the moment is the nginx-based ingress-nginx controller.

You can check it out here: `https://github.com/kubernetes/ingress-gce/blob/master/BETA_LIMITATIONS.md#glbc-beta-limitations`.

An ingress controller is deployed as a pod which runs a daemon. This pod watches the Kubernetes apiserver/ingresses endpoint for changes to the ingress resource. For our examples, we will use the default GCE backend.

Types of ingress

There are a couple different types of ingress, such as the following:

- **Single service ingress**: This strategy exposes a single service via creating an ingress with a default backend that has no rules. You can alternatively use `Service.Type=LoadBalancer` or `Service.Type=NodePort`, or a port proxy to accomplish something similar.
- **Fanout**: Given that od IP addressing is only available internally to the Kubernetes network, you'll need to use a simple fanout strategy in order to accommodate edge traffic and provide ingress to the correct endpoints in your cluster. This will resemble a load balancer in practice.
- **Name-based hosting:** This approach is similar to **service name indication** (**SNI**), which allows a web server to present multiple HTTPS websites with different certificates on the same TCP port and IP address.

Kubernetes uses host headers to route requests with this approach. The following example snippet `ingress-example.yaml` shows what name-based virtual hosting would look like:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: name-based-hosting
spec:
  rules:
  - host: example01.foo.com
    http:
      paths:
      - backend:
          serviceName: sevice01
          servicePort: 8080
  - host: example02.foo.com
    http:
      paths:
      - backend:
          serviceName: sevice02
          servicePort: 8080
```

As you may recall, in `Chapter 1`, *Introduction to Kubernetes*, we saw that a GCE cluster comes with a default back which provides Layer 7 load balancing capability. We can see this controller running if we look at the `kube-system` namespace:

```
$ kubectl get rc --namespace=kube-system
```

We should see an RC listed with the `l7-default-backend-v1.0` name, as shown here:

```
NAME                          DESIRED   CURRENT   READY     AGE
kube-dns-v20                  1         1         1         8d
kubernetes-dashboard-v1.4.0   1         1         1         8d
l7-default-backend-v1.0       1         1         1         8d
monitoring-influxdb-grafana-v4   1         1         1         8d
```

GCE Layer 7 Ingress controller

This provides the ingress controller piece that actually routes the traffic defined in our ingress entry points. Let's create some resources for an Ingress.

First, we will create a few new replication controllers with the `httpwhalesay` image. This is a remix of the original whalesay that was displayed in a browser. The following listing, `whale-rcs.yaml`, shows the YAML. Note the three dashes that let us combine several resources into one YAML file:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: whale-ingress-a
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: whale-ingress-a
    spec:
      containers:
      - name: sayhey
        image: jonbaier/httpwhalesay:0.1
        command: ["node", "index.js", "Whale Type A, Here."]
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: ReplicationController
metadata:
  name: whale-ingress-b
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: whale-ingress-b
    spec:
      containers:
      - name: sayhey
        image: jonbaier/httpwhalesay:0.1
        command: ["node", "index.js", "Hey man, It's Whale B, Just
        Chillin'."]
        ports:
        - containerPort: 80
```

Note that we are creating pods with the same container, but different startup parameters. Take note of these parameters for later. We will also create `Service` endpoints for each of these RCs as shown in the `whale-svcs.yaml` listing:

```
apiVersion: v1
kind: Service
metadata:
  name: whale-svc-a
  labels:
    app: whale-ingress-a
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30301
    protocol: TCP
    name: http
  selector:
    app: whale-ingress-a
---
apiVersion: v1
kind: Service
metadata:
  name: whale-svc-b
  labels:
    app: whale-ingress-b
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30284
    protocol: TCP
    name: http
  selector:
    app: whale-ingress-b
---
apiVersion: v1
kind: Service
metadata:
 name: whale-svc-default
 labels:
   app: whale-ingress-a
spec:
  type: NodePort
  ports:
  - port: 80
    nodePort: 30302
    protocol: TCP
```

```
    name: http
  selector:
    app: whale-ingress-a
```

Again, create these with the `kubectl create -f` command, as follows:

```
$ kubectl create -f whale-rcs.yaml
$ kubectl create -f whale-svcs.yaml
```

We should see messages about the successful creation of the RCs and Services. Next, we need to define the Ingress entry point. We will use `http://a.whale.hey` and `http://b.whale.hey` as our demo entry points as shown in the following listing `whale-ingress.yaml`:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: whale-ingress
spec:
  rules:
  - host: a.whale.hey
    http:
      paths:
      - path: /
        backend:
          serviceName: whale-svc-a
          servicePort: 80
  - host: b.whale.hey
    http:
      paths:
      - path: /
        backend:
          serviceName: whale-svc-b
          servicePort: 80
```

Again, use `kubectl create -f` to create this ingress. Once this is successfully created, we will need to wait a few moments for GCE to give the ingress a static IP address. Use the following command to watch the Ingress resource:

```
$ kubectl get ingress
```

Once the Ingress has an IP, we should see an entry in `ADDRESS`, like the one shown here:

```
NAME            HOSTS                   ADDRESS          PORTS     AGE
whale-ingress   a.whale.hey,b.whale.hey   130.211.24.177   80        3h
```

Ingress description

Since this is not a registered domain name, we will need to specify the resolution in the `curl` command, like this:

```
$ curl --resolve a.whale.hey:80:130.211.24.177 http://a.whale.hey/
```

This should display the following:

```
<html>
    <head>
        <title>HTTP Whalesay</title>
    </head>
    <body>
        <pre>
                <code>
      Whale Type A, Here.
            \
             \
                \
                                                ##         .
                                          ## ## ##        ==
                                      ## ## ## ##       ===
                                  /""""""""""""""""___/ ===
            ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
                                  \______ o          __/
                                    \    \        __/
                                      \____\______/
      </code>
      </pre>
    <body/>
</html>
```

Whalesay A

We can also try the second URL. Doing this, we will get our second RC:

```
$ curl --resolve b.whale.hey:80:130.211.24.177 http://b.whale.hey/
```

```
<html>
    <head>
        <title>HTTP Whalesay</title>
    </head>
    <body>
        <pre>
                <code>
      Hey man, It's Whale B, Just Chillin'.
             \
              \
                  \
                                                ##         .
                                          ## ## ##        ==
                                       ## ## ## ##       ===
                                   /""""""""""""""""___/ ===
           ~~~ {~~ ~~~~ ~~~ ~~~~ ~~ ~ /  ===- ~~~
                                   \_____ o           __/
                                      \    \         __/
                                         \____\_____/
      </code>
      </pre>
    <body/>
</html>
```

Whalesay B

Note that the images are almost the same, except that the words from each whale reflect the startup parameters from each RC we started earlier. Thus, our two Ingress points are directing traffic to different backends.

In this example, we used the default GCE backend for an Ingress controller. Kubernetes allows us to build our own, and nginx actually has a few versions available as well.

Migrations, multicluster, and more

As we've already seen so far, Kubernetes offers a high level of flexibility and customization to create a service abstraction around your containers running in the cluster. However, there may be times where you want to point to something outside your cluster.

An example of this would be working with legacy systems or even applications running on another cluster. In the case of the former, this is a perfectly good strategy in order to migrate to Kubernetes and containers in general. We can begin by managing the service endpoints in Kubernetes while stitching the stack together using the K8s orchestration concepts. Additionally, we can even start bringing over pieces of the stack, as the frontend, one at a time as the organization refactors applications for microservices and/or containerization.

To allow access to non pod-based applications, the services construct allows you to use endpoints that are outside the cluster. Kubernetes is actually creating an endpoint resource every time you create a service that uses selectors. The `endpoints` object keeps track of the pod IPs in the load balancing pool. You can see this by running the `get endpoints` command, as follows:

```
$ kubectl get endpoints
```

You should see something similar to the following:

```
NAME              ENDPOINTS
http-pd           10.244.2.29:80,10.244.2.30:80,10.244.3.16:80
kubernetes        10.240.0.2:443
node-js           10.244.0.12:80,10.244.2.24:80,10.244.3.13:80
```

You'll note the entry for all the services we currently have running on our cluster. For most services, the endpoints are just the IP of each pod running in an RC. As I mentioned previously, Kubernetes does this automatically based on the selector. As we scale the replicas in a controller with matching labels, Kubernetes will update the endpoints automatically.

If we want to create a service for something that is not a pod and therefore has no labels to select, we can easily do this with both a service definition `nodejs-custom-service.yaml` and endpoint definition `nodejs-custom-endpoint.yaml`, as follows:

```
apiVersion: v1
kind: Service
metadata:
  name: custom-service
spec:
  type: LoadBalancer
  ports:
  - name: http
    protocol: TCP
    port: 80
```

```
apiVersion: v1
kind: Endpoints
metadata:
  name: custom-service
subsets:
- addresses:
  - ip: <X.X.X.X>
  ports:
    - name: http
      port: 80
      protocol: TCP
```

In the preceding example, you'll need to replace `<X.X.X.X>` with a real IP address, where the new service can point to. In my case, I used the public load balancer IP from the `node-js-multi` service we created earlier in listing `ingress-example.yaml`. Go ahead and create these resources now.

If we now run a `get endpoints` command, we will see this IP address at port `80`, which is associated with the `custom-service` endpoint. Furthermore, if we look at the service details, we will see the IP listed in the `Endpoints` section:

```
$ kubectl describe service/custom-service
```

We can test out this new service by opening the `custom-service` external IP from a browser.

Custom addressing

Another option to customize services is with the `clusterIP` element. In our examples so far, we've not specified an IP address, which means that it chooses the internal address of the service for us. However, we can add this element and choose the IP address in advance with something like `clusterip: 10.0.125.105`.

There may be times when you don't want to load balance and would rather have DNS with *A* records for each pod. For example, software that needs to replicate data evenly to all nodes may rely on *A* records to distribute data. In this case, we can use an example like the following one and set `clusterip` to `None`.

Kubernetes will not assign an IP address and instead only assign *A* records in DNS for each of the pods. If you are using DNS, the service should be available at `node-js-none` or `node-js-none.default.cluster.local` from within the cluster. For this, we will use the following listing `nodejs-headless-service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-none
  labels:
    name: node-js-none
spec:
  clusterIP: None
  ports:
  - port: 80
  selector:
    name: node-js
```

Test it out after you create this service with the trusty `exec` command:

```
$ kubectl exec node-js-pod -- curl node-js-none
```

Service discovery

As we discussed earlier, the Kubernetes master keeps track of all service definitions and updates. Discovery can occur in one of three ways. The first two methods use Linux environment variables. There is support for the Docker link style of environment variables, but Kubernetes also has its own naming convention. Here is an example of what our `node-js` service example might look like using K8s environment variables (note that IPs will vary):

```
NODE_JS_PORT_80_TCP=tcp://10.0.103.215:80
NODE_JS_PORT=tcp://10.0.103.215:80
NODE_JS_PORT_80_TCP_PROTO=tcp
NODE_JS_PORT_80_TCP_PORT=80
NODE_JS_SERVICE_HOST=10.0.103.215
NODE_JS_PORT_80_TCP_ADDR=10.0.103.215
NODE_JS_SERVICE_PORT=80
```

Another option for discovery is through DNS. While environment variables can be useful when DNS is not available, it has drawbacks. The system only creates variables at creation time, so services that come online later will not be discovered or will require some additional tooling to update all the system environments.

DNS

DNS solves the issues seen with environment variables by allowing us to reference the services by their name. As services restart, scale out, or appear anew, the DNS entries will be updating and ensuring that the service name always points to the latest infrastructure. DNS is set up by default in most of the supported providers. You can add DNS support for your cluster via a cluster add on (`https://kubernetes.io/docs/concepts/cluster-administration/addons/`).

If DNS is supported by your provider, but is not set up, you can configure the following variables in your default provider config when you create your Kubernetes cluster:

```
ENABLE_CLUSTER_DNS="${KUBE_ENABLE_CLUSTER_DNS:-true}"
DNS_SERVER_IP="10.0.0.10"
DNS_DOMAIN="cluster.local"
DNS_REPLICAS=1.
```

With DNS active, services can be accessed in one of two forms—either the service name itself, `<service-name>`, or a fully qualified name that includes the namespace, `<service-name>.<namespace-name>.cluster.local`. In our examples, it would look similar to `node-js-90` or `node-js-90.default.cluster.local`.

The DNS server create DNS records based on new services that are created through the API. Pods in shared DNS namespaces will be able to see each other, and can use DNS SRV records to record ports as well.

Kubernetes DNS is comprised of a DNS pod and Service on the cluster which communicates directly with kubelets and containers in order to translate DNS names to IP. Services with clusterIPs are given `my-service.my-namespace.svc.cluster.local` addresses. If the service does not have a clusterIP (otherwise called headless) it gets the same address format, but this resolves in a round-robin fashion to a number of IPs that point to the pods of a service. There a number of DNS policies that can also be set.

One of the Kubernetes incubator projects, `CoreDNS` can also be used for service discovery. This replaces the native `kube-dns` DNS services and requires Kubernetes v1.9 or later. You'll need to leverage `kubeadm` during the initialization process in order to try CoreDNS out. You can install this on your cluster with the following command:

```
$ kubeadm init --feature-gates=CoreDNS=true
```

If you'd like more information on an example use case of CoreDNS, check out this blog post: `https://coredns.io/2017/05/08/custom-dns-entries-for-kubernetes/`.

Multitenancy

Kubernetes also has an additional construct for isolation at the cluster level. In most cases, you can run Kubernetes and never worry about namespaces; everything will run in the default namespace if not specified. However, in cases where you run multitenancy communities or want broad-scale segregation and isolation of the cluster resources, namespaces can be used to this end. True, end-to-end multitenancy is not yet feature complete in Kubernetes, but you can get very close using RBAC, container permissions, ingress rules, and clear network policing. If you're interested in enterprise-strength multitenancy right now, Red Hat's **Openshift Origin** (**OO**) would be a good place to learn.

You can check out OO at `https://github.com/openshift/origin`.

To start, Kubernetes has two namespaces—`default` and `kube-system`. The `kube-system` namespace is used for all the system-level containers we saw in `Chapter 1`, *Introduction to Kubernetes*, in the *Services running on the minions* section. UI, logging, DNS, and so on are all run in `kube-system`. Everything else the user creates runs in the default namespace. However, our resource definition files can optionally specify a custom namespace. For the sake of experimenting, let's take a look at how to build a new namespace.

First, we'll need to create a namespace definition file `test-ns.yaml` like the one in the following lines of code:

```
apiVersion: v1
kind: Namespace
metadata:
  name: test
```

We can go ahead and create this file with our handy `create` command:

```
$ kubectl create -f test-ns.yaml
```

Now, we can create resources that use the `test` namespace. The following listing, `ns-pod.yaml`, is an example of a pod using this new namespace:

```
apiVersion: v1
kind: Pod
metadata:
  name: utility
  namespace: test
spec:
  containers:
  - image: debian:latest
    command:
      - sleep
      - "3600"
    name: utility
```

While the pod can still access services in other namespaces, it will need to use the long DNS form of `<service-name>.<namespace-name>.cluster.local`. For example, if you were to run a command from inside the container in listing `ns-pod.yaml`, you could use `node-js.default.cluster.local` to access the Node.js example from `Chapter 2`, *Pods, Services, Replication Controllers, and Labels*.

Here is a note about resource utilization. At some point in this book, you may run out of space on your cluster to create new Kubernetes resources. The timing will vary based on cluster size, but it's good to keep this in mind and do some cleanup from time to time. Use the following commands to remove old examples:

```
$ kubectl delete pod <pod name>
$ kubectl delete svc <service name>
$ kubectl delete rc <replication controller name>
$ kubectl delete rs <replicaset name>.
```

Limits

Let's inspect our new namespace a bit more. Run the `describe` command as follows:

```
$ kubectl describe namespace/test
```

The following screenshot is the result of the preceding command:

```
Name:   test
Labels: <none>
Status: Active

No resource quota.

No resource limits.
```

The describe namespace

Kubernetes allows you to both limit the resources used by individual pods or containers and the resources used by the overall namespace using quotas. You'll note that there are no resource limits or quotas currently set on the `test` namespace.

Suppose we want to limit the footprint of this new namespace; we can set quotas as shown in the following listing `quota.yaml`:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: test-quotas
  namespace: test
spec:
  hard:
    pods: 3
    services: 1
    replicationcontrollers: 1
```

In reality, namespaces would be for larger application communities and would probably never have quotas this low. I am using this for ease of illustration of the capability in this example.

Here, we will create a quota of `3` pods, `1` RC, and `1` service for the test namespace. As you have probably guessed, this is executed once again by our trusty `create` command, as follows:

```
$ kubectl create –f quota.yaml
```

Now that we have that in place, let's use `describe` on the namespace, as follows:

```
$ kubectl describe namespace/test
```

The following screenshot is the result of the preceding command:

```
Name:   test
Labels: <none>
Status: Active

Resource Quotas
 Resource                Used    Hard
 ---                     ---     ---
 pods                    0       3
 replicationcontrollers  0       1
 services                0       1

No resource limits.
```

The describe namespace after the quota is set

You'll note that we now have some values listed in the quota section, and that the limits section is still blank. We also have a `Used` column, which lets us know how close to the limits we are at the moment. Let's try to spin up a few pods using the following definition `busybox-ns.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: busybox-ns
  namespace: test
  labels:
    name: busybox-ns
spec:
  replicas: 4
  selector:
    name: busybox-ns
```

```
  template:
    metadata:
      labels:
        name: busybox-ns
    spec:
      containers:
      - name: busybox-ns
        image: busybox
        command:
          - sleep
          - "3600"
```

You'll note that we are creating four replicas of this basic pod. After using `create` to build this RC, run the `describe` command on the `test` namespace once more. You'll notice that the `Used` values for pods and RCs are at their max. However, we asked for four replicas and can only see three pods in use.

Let's see what's happening with our RC. You might attempt to do that with the following command:

```
kubectl describe rc/busybox-ns
```

However, if you try, you'll be discouraged by being met with a `not found` message from the server. This is because we created this RC in a new namespace and `kubectl` assumes the default namespace if not specified. This means that we need to specify `--namepsace=test` with every command when we wish to access resources in the `test` namespace.

We can also set the current namespace by working with the context settings. First, we need to find our current context, which is found with the following command:

```
$ kubectl config view | grep current-context
```

Next, we can take that context and set the namespace variable like in the following code:

```
$ kubectl config set-context <Current Context> --
namespace=test
```

Now, you can run the `kubectl` command without the need to specify the namespace. Just remember to switch back when you want to look at the resources running in your default namespace.

Run the command with the namespace specified as shown in the following command. If you've set your current namespace as demonstrated in the tip box, you can leave off the `--namespace` argument:

```
$ kubectl describe rc/busybox-ns --namespace=test
```

The following screenshot is the result of the preceding command:

```
Name:           busybox-ns
Namespace:      test
Image(s):       busybox
Selector:       name=busybox-ns
Labels:         name=busybox-ns
Replicas:       3 current / 4 desired
Pods Status:    3 Running / 0 Waiting / 0 Succeeded / 0 Failed
Events:
  FirstSeen                             LastSeen                        Count   F
rom                             SubobjectPath   Reason                  Message
  Mon, 17 Aug 2015 16:29:43 -0400       Mon, 17 Aug 2015 16:29:43 -0400 1       {
replication-controller }                        successfulCreate        Created p
od: busybox-ns-spfrn
  Mon, 17 Aug 2015 16:29:43 -0400       Mon, 17 Aug 2015 16:29:43 -0400 1       {
replication-controller }                        successfulCreate        Created p
od: busybox-ns-xjf6q
  Mon, 17 Aug 2015 16:29:43 -0400       Mon, 17 Aug 2015 16:29:43 -0400 1       {
replication-controller }                        successfulCreate        Created p
od: busybox-ns-zeuuy
  Mon, 17 Aug 2015 16:29:44 -0400       Mon, 17 Aug 2015 16:33:01 -0400 18      {
replication-controller }                        failedCreate            Error cre
ating: Pod "busybox-ns-" is forbidden: Limited to 3 pods
```

Namespace quotas

As you can see in the preceding image, the first three pods were successfully created, but our final one fails with a `Limited to 3 pods` error.

This is an easy way to set limits for resources partitioned out at a community scale. It's worth noting that you can also set quotas for CPU, memory, persistent volumes, and secrets. Additionally, limits work in a similar way to quota, but they set the limit for each pod or container within the namespace.

A note on resource usage

As most of the examples in this book utilize GCP or AWS, it can be costly to keep everything running. It's also easy to run out of resources using the default cluster size, especially if you keep every example running. Therefore, you may want to delete older pods, replication controllers, replica sets, and services periodically. You can also destroy the cluster and recreate it using `Chapter 1`, *Introduction to Kubernetes*, as a way to lower your cloud provider bill.

Summary

In this chapter, we took a deeper look into networking and services in Kubernetes. You should now understand how networking communications are designed in K8s and feel comfortable accessing your services internally and externally. We saw how `kube-proxy` balances traffic both locally and across the cluster. Additionally, we explored the new Ingress resources that allow us finer control of incoming traffic. We also looked briefly at how DNS and service discovery is achieved in Kubernetes. We finished off with a quick look at namespaces and isolation for multitenancy.

Questions

1. Give two way in which the Docker networking approach is different than the Kubernetes networking approach.
2. What does NAT stand for?
3. What are the two major classes of Kubernetes networking models?
4. Name at least two of the third-party overlay networking options available to Kubernetes.
5. At what level (or alternatively, to what object) does Kubernetes assign IP addresses?
6. What are the available modes for `kube-proxy`?
7. What are the three types of services allowed by Kubernetes?
8. What elements are used to define container and service ports?
9. Name two or more types of ingress available to Kubernetes.
10. How can you provide multitenancy for your Kubernetes cluster?

Further reading

- Read more about CoreDNS's entry in the CNCF: `https://coredns.io/2018/03/12/coredns-1.1.0-release/`.
- More details on the current crop of Kubernetes network provider scan be found at `https://kubernetes.io/docs/concepts/cluster-administration/networking/#how-to-achieve-this`.
- You can compare nginx's implementation of an ingress controller (`https://github.com/nginxinc/kubernetes-ingress`) and the Kubernetes community approach (`https://github.com/kubernetes/ingress-nginx`) and their differences (`https://github.com/nginxinc/kubernetes-ingress/blob/master/docs/nginx-ingress-controllers.md`).
- You can read up about Google Compute Engine's layer 7 load balancer, GLBC at `https://github.com/kubernetes/ingress-gce/`.

4
Implementing Reliable Container-Native Applications

This chapter will cover the various types of workloads that Kubernetes supports. We will cover deployments for applications that are regularly updated and long-running. We will also revisit the topics of application updates and gradual rollouts using Deployments. In addition, we will look at jobs used for short-running tasks. We will look at DaemonSets, which allow programs to be run on every node in our Kubernetes cluster. In case you noticed, we won't look into StatefulSets yet in this chapter but we'll investigate them in the next, when we look at store and how K8s helps you manage storage and stateful applications on your cluster.

The following topics will be covered in this chapter:

- Deployments
- Application scaling with Deployments
- Application updates with Deployments
- Jobs
- DaemonSets

Technical requirements

You'll need a running Kubernetes cluster like the one we created in the previous chapters. You'll also need access to deploy to that cluster through the `kubectl` command.

Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter04`.

How Kubernetes manages state

As discussed previously, we know that Kubernetes makes an effort to enforce the desired state of the operator in a given cluster. Deployments give operators the ability to define an end state and the mechanisms to effect change at a controlled rate of stateless services, such as microservices. Since Kubernetes is a control and data plane that manages the metadata, current status, and specification of a set of objects, Deployments provide a deeper level of control for your applications. There are a few archetypal deployment patterns that are available: recreate, rolling update, blue/green via selector, canary via replicas, and A/B via HTTP headers.

Deployments

In the previous chapter, we explored some of the core concepts for application updates using the old rolling-update method. Starting with version 1.2, Kubernetes added the **Deployment** construct, which improves on the basic mechanisms of rolling-update and **ReplicationControllers**. As the name suggests, it gives us finer control over the code deployment itself. Deployments allow us to pause and resume application rollouts via declarative definitions and updates to pods and **ReplicaSets.** Additionally, they keep a history of past deployments and allow the user to easily roll back to previous versions.

It is no longer recommended to use ReplicationControllers. Instead, use a Deployment that configures a ReplicaSet in order to set up application availability for your stateless services or applications. Furthermore, do not directly manage the ReplicaSets that are created by your deployments; only do so through the Deployment API.

Deployment use cases

We'll explore a number of typical scenarios for deployments in more detail:

- Roll out a ReplicaSet
- Update the state of a set of Pods
- Roll back to an earlier version of a Deployment
- Scale up to accommodate cluster load
- Pause and use Deployment status in order to make changes or indicate a stuck deployment
- Clean up a deployment

In the following code of the `node-js-deploy.yaml` file, we can see that the definition is very similar to a ReplicationController. The main difference is that we now have an ability to make changes and updates to the deployment objects and let Kubernetes manage updating the underlying pods and replicas for us:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: node-js-deploy
  labels:
    name: node-js-deploy
spec:
  replicas: 1
  template:
    metadata:
      labels:
        name: node-js-deploy
    spec:
      containers:
      - name: node-js-deploy
        image: jonbaier/pod-scaling:0.1
        ports:
        - containerPort: 80
```

In this example, we've created a Deployment named `node-js-deploy` via the `name` field under `metadata`. We're creating a single pod that will be managed by the `selector` field, which is going to help the Deployment understand which pods to manage. The `spec` tells the pod to run the `jobbaier/pod-scaling` container and directs traffic through port `80` via the `containerPort`.

We can run the familiar `create` command with the optional `--record` flag so that the creation of the Deployment is recorded in the rollout history. Otherwise, we will only see subsequent changes in the rollout history using the `$ kubectl create -f node-js-deploy.yaml --record` command.

You may need to add `--validate=false` if this beta type is not enabled on your cluster.

We should see a message about the deployment being successfully created. After a few moments, it will finish creating our pod, which we can check for ourselves with a `get pods` command. We add the `-l` flag to only see the pods relevant to this deployment:

```
$ kubectl get pods -l name=node-js-deploy
```

If you'd like to get the state of the deployment, you can issue the following:

```
$ kubectl get deployments
```

You can also see the state of a rollout, which will be more useful in the future when we update our Deployments. You can use `kubectl rollout status deployment/node-js-deploy` to see what's going on.

We create a service just as we did with ReplicationControllers. The following is a `Service` definition for the Deployment we just created. Notice that it is almost identical to the `Services` we created in the past. Save the following code in `node-js-deploy-service.yaml` file:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-deploy
  labels:
    name: node-js-deploy
spec:
  type: LoadBalancer
  ports:
  - port: 80
  sessionAffinity: ClientIP
  selector:
    name: node-js-deploy
```

Once this service is created using `kubectl`, you'll be able to access the deployment pods through the service IP or the service name if you are inside a pod on this namespace.

Scaling

The `scale` command works the same way as it did in our ReplicationController. To scale up, we simply use the deployment name and specify the new number of replicas, as shown here:

```
$ kubectl scale deployment node-js-deploy --replicas 3
```

If all goes well, we'll simply see a message about the deployment being scaled in the output of our Terminal window. We can check the number of running pods using the `get pods` command from earlier. In the latest versions of Kubernetes, you're also able to set up pod scaling for your cluster, which allows you to do horizontal autoscaling so you can scale up pods based on the CPU utilization of your cluster. You'll need to set a maximum and minimum number of pods in order to get this going.

Here's what that command would look like with this example:

```
$ kubectl autoscale deployment node-js-deploy --min=25 --max=30 --cpu-
percent=75
deployment "node-js-deploy" autoscaled
```

Read more about horizontal pod scaling in this walkthrough: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale-walkthrough/`.

There's also a concept of proportional scaling, which allows you to run multiple version of your application at the same time. This implementation would be useful when incrementing a backward-compatible version of an API-based microservice, for example. When doing this type of deployment, you'll use `.spec.strategy.rollingUpdate.maxUnavailable` and `.spec.strategy.rollingUpdate.maxSurge` to limit the maximum number of pods that can be down during an update to the deployment, or the maximum number of pods that can be created that exceed the desired number of pods, respectively.

Updates and rollouts

Deployments allow for updating in a few different ways. First, there is the `kubectl set` command, which allows us to change the deployment configuration without redeploying manually. Currently, it only allows for updating the image, but as new versions of our application or container image are processed, we will need to do this quite often.

Let's take a look using our deployment from the previous section. We should have three replicas running right now. Verify this by running the `get pods` command with a filter for our deployment:

```
$ kubectl get pods -l name=node-js-deploy
```

We should see three pods similar to those listed in the following screenshot:

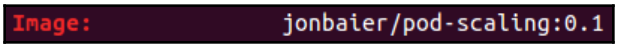
```
NAME                              READY     STATUS    RESTARTS   AGE
node-js-deploy-1713031517-itnwi   1/1       Running   0          6m
node-js-deploy-1713031517-nx8vs   1/1       Running   0          6m
node-js-deploy-1713031517-uge5y   1/1       Running   0          6m
```

Deployment pod listing

Take one of the pods listed on our setup, replace it in the following command where it says `{POD_NAME_FROM_YOUR_LISTING}`, and run this command:

```
$ kubectl describe pod/{POD_NAME_FROM_YOUR_LISTING} | grep Image:
```

We should see an output like the following screenshot with the current image version of `0.1`:



Current pod image

Now that we know what our current deployment is running, let's try to update to the next version. This can be achieved easily using the `kubectl set` command and specifying the new version, as shown here:

```
$ kubectl set image deployment/node-js-deploy node-js-deploy=jonbaier/pod-
scaling:0.2
$ deployment "node-js-deploy" image updated
```

If all goes well, we should see text that says `deployment "node-js-deploy" image updated` displayed on the screen.

We can double–check the status using the following `rollout status` command:

```
$ kubectl rollout status deployment/node-js-deploy
```

Alternatively, we can directly edit the deployment in an editor window with `kubectl edit deployment/node-js-deploy` and change `.spec.template.spec.containers[0].image` from `jonbaier/pod-scaling:0.1` to `jonbaier/pod-scaling:0.2`. Either of these methods will work to update your deployment, and as a reminder you can check the status of your update with the `kubectl status` command:

```
$ kubectl rollout status deployment/node-js-deployment
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
deployment "node-js-deployment" successfully rolled out
```

We should see some text saying that the deployment successfully rolled out. If you see any text about waiting for the rollout to finish, you may need to wait a moment for it to finish, or alternatively check the logs for issues.

Once it's finished, run the `get pods` command as earlier. This time, we will see new pods listed:

```
NAME                                READY     STATUS    RESTARTS   AGE
node-js-deploy-1794296158-5wivi     1/1       Running   0          5m
node-js-deploy-1794296158-b2any     1/1       Running   0          5m
node-js-deploy-1794296158-y2tx3     1/1       Running   0          5m
```

Deployment pod listing after update

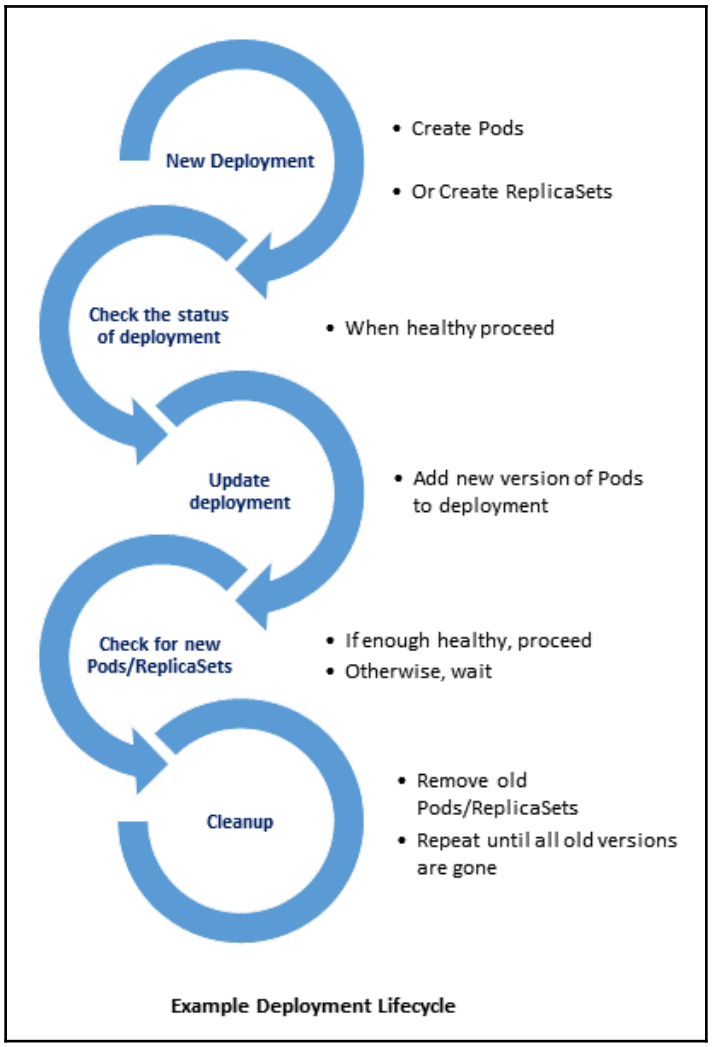
Once again, plug one of your pod names into the `describe` command we ran earlier. This time, we should see the image has been updated to 0.2.

What happened behind the scenes is that Kubernetes has *rolled out* a new version for us. It basically creates a new ReplicaSet with the new version. Once this pod is online and healthy, it kills one of the older versions. It continues this behavior, scaling out the new version and scaling down the old versions, until only the new pods are left. Another way to observe this behavior indirectly is to investigate the ReplicaSet that the Deployment object is using to update your desired application state.

Remember, you don't interact directly with ReplicaSet, but rather give Kubernetes directives in the form of Deployment elements and let Kubernetes make the required changes to the cluster object store and state. Take a look at the ReplicaSets quickly after running your `image update` command, and you'll see how multiple ReplicaSets are used to effect the image change without application downtime:

```
$ kubectl get rs
NAME                              DESIRED  CURRENT  READY  AGE
node-js-deploy-1556879905         3        3        3      46s
node-js-deploy-4657899444         0        0        0      85s
```

The following diagram describes the workflow for your reference:



Deployment life cycle

It's worth noting that the rollback definition allows us to control the pod replace method in our deployment definition. There is a `strategy.type` field that defaults to `RollingUpdate` and the preceding behavior. Optionally, we can also specify `Recreate` as the replacement strategy and it will kill all the old pods first before creating the new versions.

History and rollbacks

One of the useful features of the rollout API is the ability to track the deployment history. Let's do one more update before we check the history. Run the `kubectl set` command once more and specify version 0.3:

```
$ kubectl set image deployment/node-js-deploy node-js-deploy=jonbaier/pod-scaling:0.3
$ deployment "node-js-deploy" image updated
```

Once again, we'll see text that says `deployment "node-js-deploy" image updated` displayed on the screen. Now, run the `get pods` command once more:

```
$ kubectl get pods -l name=node-js-deploy
```

Let's also take a look at our deployment history. Run the `rollout history` command:

```
$ kubectl rollout history deployment/node-js-deploy
```

We should see an output similar to the following:

```
REVISION        CHANGE-CAUSE
1               kubectl scale deployment node-js-deploy --replicas 3
2               kubectl set image deployment/node-js-deploy node-js-deploy=jonbaier/pod-scaling:0.2
3               kubectl set image deployment/node-js-deploy node-js-deploy=jonbaier/pod-scaling:0.3
```

Rollout history

As we can see, the history shows us the initial deployment creation, our first update to 0.2, and then our final update to 0.3. In addition to status and history, the `rollout` command also supports the `pause`, `resume`, and `undo` sub-commands. The `rollout pause` command allows us to pause a command while the rollout is still in progress. This can be useful for troubleshooting and also helpful for canary-type launches, where we wish to do final testing of the new version before rolling out to the entire user base. When we are ready to continue the rollout, we can simply use the `rollout resume` command.

But what if something goes wrong? That is where the `rollout undo` command and the rollout history itself are really handy. Let's simulate this by trying to update to a version of our pod that is not yet available. We will set the image to version 42.0, which does not exist:

```
$ kubectl set image deployment/node-js-deploy node-js-deploy=jonbaier/pod-scaling:42.0
```

We should still see the text that says `deployment "node-js-deploy" image updated` displayed on the screen. But if we check the status, we will see that it is still waiting:

```
$ kubectl rollout status deployment/node-js-deploy
Waiting for rollout to finish: 2 out of 3 new replicas have been updated...
```

Here, we see that the deployment has been paused after updating two of the three pods, but Kubernetes knows enough to stop there in order to prevent the entire application from going offline due to the mistake in the container image name. We can press *Ctrl* + *C* to kill the `status` command and then run the `get pods` command once more:

```
$ kubectl get pods -l name=node-js-deploy
```

We should now see an `ErrImagePull`, as in the following screenshot:

```
NAME                              READY     STATUS         RESTARTS   AGE
node-js-deploy-1875560799-ehi0o   1/1       Running        0          40m
node-js-deploy-1875560799-tqset   1/1       Running        0          40m
node-js-deploy-1907673490-cadw2   0/1       ErrImagePull   0          1m
node-js-deploy-1907673490-qvc9w   0/1       ErrImagePull   0          1m
```

Image pull error

As we expected, it can't pull the 42.0 version of the image because it doesn't exist. This error refers to a container that's stuck in an image pull loop, which is noted as `ImagePullBackoff` in the latest versions of Kubernetes. We may also have issues with deployments if we run out of resources on the cluster or hit limits that are set for our namespace. Additionally, the deployment can fail for a number of application-related causes, such as health check failure, permission issues, and application bugs, of course.

It's entirely possible to create deployments that are wholly unavailable if you don't change `maxUnavailable` and `spec.replicas` to different numbers, as the default for each is `1`!

Whenever a failure to roll out happens, we can easily roll back to a previous version using the `rollout undo` command. This command will take our deployment back to the previous version:

```
$ kubectl rollout undo deployment/node-js-deploy
```

After that, we can run a `rollout status` command once more and we should see everything rolled out successfully. Run the `kubectl rollout history deployment/node-js-deploy` command again and we'll see both our attempt to roll out version 42.0 and revert to 0.3:

```
REVISION        CHANGE-CAUSE
1               kubectl scale deployment node-js-deploy --replicas 3
2               kubectl set image deployment/node-js-deploy node-js-de
ploy=jonbaier/pod-scaling:0.2
4               kubectl set image deployment/node-js-deploy node-js-de
ploy=jonbaier/pod-scaling:42.0
5               kubectl set image deployment/node-js-deploy node-js-de
ploy=jonbaier/pod-scaling:0.3
```

Rollout history after rollback

We can also specify the `--to-revision` flag when running an undo to roll back to a specific version. This can be handy for times when our rollout succeeds, but we discover logical errors down the road.

Autoscaling

As you can see, Deployments are a great improvement over ReplicationControllers, allowing us to seamlessly update our applications, while integrating with the other resources of Kubernetes in much the same way.

Another area that we saw in the previous chapter, and also supported for Deployments, is **Horizontal Pod Autoscalers** (**HPAs**). HPAs help you manage cluster utilization by scaling the number of pods based on CPU utilization. There are three objects that can scale using HPAs, DaemonSets not included:

- Deployment (the recommended method)
- ReplicaSet
- ReplicationController (not recommended)

The HPA is implemented as a control loop similar to other controllers that we've discussed, and you can adjust the sensitivity of the controller manager by adjusting its sync period via `--horizontal-pod-autoscaler-sync-period` (default 30 seconds).

We will walk through a quick remake of the HPAs from the previous chapter, this time using the Deployments we have created so far. Save the following code in `node-js-deploy-hpa.yaml` file:

```
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: node-js-deploy
spec:
  minReplicas: 3
  maxReplicas: 6
  scaleTargetRef:
    apiVersion: v1
    kind: Deployment
    name: node-js-deploy
  targetCPUUtilizationPercentage: 10
```

The API is changing quickly with these tools as they're in beta, so take careful note of the `apiVersion` element, which used to be `autoscaling/v1`, but is now `autoscalingv2beta1`.

We have lowered the CPU threshold to 10% and changed our minimum and maximum pods to `3` and `6`, respectively. Create the preceding HPA with our trusty `kubectl create -f` command. After this is completed, we can check that it's available with the `kubectl get hpa` command:

```
NAME             REFERENCE                                TARGET    CURRENT   MIN
PODS   MAXPODS   AGE
node-js-deploy   Deployment/node-js-deploy                10%       0%        3
       6         3h
node-js-scale    ReplicationController/node-js-scale      30%       0%        1
       3         10d
```

Horizontal pod autoscaler

We can also check that we have only `3` pods running with the `kubectl get deploy` command. Now, let's add some load to trigger the autoscaler:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: boomload-deploy
spec:
  replicas: 1
  template:
```

```
      metadata:
        labels:
          app: loadgenerator-deploy
      spec:
        containers:
        - image: williamyeh/boom
          name: boom-deploy
          command: ["/bin/sh","-c"]
          args: ["while true ; do boom http://node-js-deploy/ -c 100 -n 500 ;
sleep 1 ; done"]
```

Create `boomload-deploy.yaml` file as usual. Now, monitor the HPA with the alternating `kubectl get hpa` and `kubectl get deploy` commands. After a few moments, we should see the load jump above `10%`. After a few more moments, we should also see the number of pods increase all the way up to `6` replicas:

```
$ kubectl get hpa
NAME             REFERENCE                                TARGET    CURRENT   MIN
PODS   MAXPODS   AGE
node-js-deploy   Deployment/node-js-deploy                10%       20%       3
       6         8m
node-js-scale    ReplicationController/node-js-scale      30%       0%        1
       3         9d
$ kubectl get deploy
NAME             DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
boomload-deploy  1         1         1            1           4m
node-js-deploy   6         6         6            6           10d
```

HPA increase and pod scale up

Again, we can clean this up by removing our load generation pod and waiting a few moments:

```
$ kubectl delete deploy boomload-deploy
```

Again, if we watch the HPA, we'll start to see the CPU usage drop. After a few minutes, we will go back down to `0%` CPU load and then the Deployment will scale back to `3` replicas.

Jobs

Deployments and ReplicationControllers are a great way to ensure long-running applications are always up and able to tolerate a wide array of infrastructure failures. However, there are some use cases this does not address, specifically short-running, run once tasks, as well as regularly scheduled tasks. In both cases, we need the tasks to run until completion, but then terminate and start again at the next scheduled interval.

To address this type of workload, Kubernetes has added a `batch` API, which includes the `Job` type. This type will create 1 to n pods and ensure that they all run to completion with a successful exit. Based on `restartPolicy`, we can either allow pods to simply fail without retry (`restartPolicy: Never`) or retry when a pods exits without successful completion (`restartPolicy: OnFailure`). In this example, we will use the latter technique as shown in the listing `longtask.yaml`:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: long-task
spec:
  template:
    metadata:
      name: long-task
    spec:
      containers:
      - name: long-task
        image: docker/whalesay
        command: ["cowsay", "Finishing that task in a jiffy"]
      restartPolicy: OnFailure
```

Let's go ahead and run this with the following command:

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

If all goes well, you'll see `job "long-task" created` printed on the screen.

This tells us the job was created, but doesn't tell us if it completed successfully. To check that, we need to query the job status with the following command:

```
$ kubectl describe jobs/long-task
```

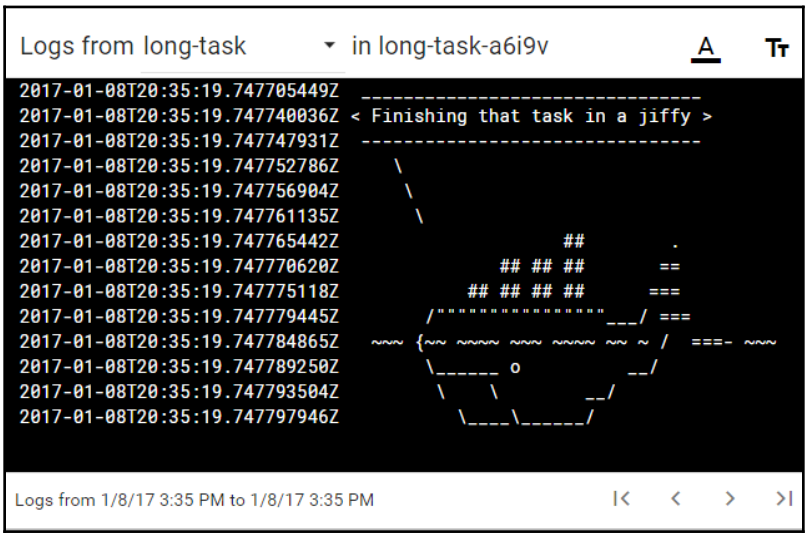
```
Name:           long-task
Namespace:      default
Image(s):       docker/whalesay
Selector:       controller-uid=eff2fcd2-d5e1-11e6-90ee-42010a800002
Parallelism:    1
Completions:    1
Start Time:     Sun, 08 Jan 2017 15:35:05 -0500
Labels:         <none>
Pods Statuses:  0 Running / 1 Succeeded / 0 Failed
No volumes.
Events:
  FirstSeen     LastSeen        Count   From                    SubobjectPath   Type
Reason                  Message
  ---------     --------        -----   ----                    -------------   ---------
-----                   -------
  4m            4m              1       {job-controller }                       Normal
SuccessfulCreate        Created pod: long-task-a6i9v
```

Job status

You should see that we had `1` task that succeeded, and in the `Events` logs, we have a `SuccessfulCreate` message. If we use the `kubectl get pods` command, we won't see our `long-task` pods in the list, but we may notice the message at the bottom in the listing states that there are completed jobs that are not shown. We will need to run the command again with the `-a` or `--show-all` flag to see the `long-task` pod and the completed job status.

Let's dig a little deeper to prove to ourselves the work was completed successfully. We could use the `logs` command to look at the pod logs. However, we can also use the UI for this task. Open a browser and go to the following UI URL: `https://<your master ip>/ui/`.

Click on **Jobs** and then **long-task** from the list, so we can see the details. Then, in the **Pods** section, click on the pod listed there. This will give us the **Pod details** page. At the bottom of the details, click on **View Logs** and we will see the log output:



Job log

As you can see in the preceding screenshot, the whalesay container is complete with the ASCII art and our custom message from the runtime parameters in the example.

Other types of jobs

While this example provides a basic introduction to short-running jobs, it only addresses the use case of once and done tasks. In reality, batch work is often done in parallel or as part of a regularly occurring task.

Parallel jobs

Using parallel jobs, we may be grabbing tasks from an ongoing queue or simply running a set number of tasks that are not dependent on each other. In the case of jobs pulling from a queue, our application must be aware of the dependencies and have the logic to decide how tasks are processed and what to work on next. Kubernetes is simply scheduling the jobs.

You can learn more about parallel jobs from the Kubernetes documentation and batch API reference.

Scheduled jobs

For tasks that need to run periodically, Kubernetes has also released a `CronJob` type in alpha. As we might expect, this type of job uses the underlying cron formatting to specify a schedule for the task we wish to run. By default, our cluster will not have the alpha batch features enabled, but we can look at an example `CronJob` listing to learn how these types of workloads will work going forward. Save the following code in `longtask-cron.yaml` file:

```
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: long-task-cron
spec:
  schedule: "15 10 * * 6"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: long-task-cron
            image: docker/whalesay
            command: ["cowsay", "Developers! Developers! Developers!
          \n\n Saturday task
            complete!"]
          restartPolicy: OnFailure
```

As you can see, the schedule portion reflects a crontab with the following format: *minute hour day-of-month month day-of-week*. In this example, 15 10 * * 6 creates a task that will run every Saturday at 10:15 am.

DaemonSets

While ReplicationControllers and Deployments are great at making sure that a specific number of application instances are running, they do so in the context of the best fit. This means that the scheduler looks for nodes that meet resource requirements (available CPU, particular storage volumes, and so on) and tries to spread across the nodes and zones.

This works well for creating highly available and fault tolerant applications, but what about cases where we need an agent to run on every single node in the cluster? While the default spread does attempt to use different nodes, it does not guarantee that every node will have a replica and, indeed, will only fill a number of nodes equivalent to the quantity specified in the ReplicationController or Deployment specification.

To ease this burden, Kubernetes introduced `DaemonSet`, which simply defines a pod to run on every single node in the cluster or a defined subset of those nodes. This can be very useful for a number of production–related activities, such as monitoring and logging agents, security agents, and filesystem daemons.

In Kubernetes version 1.6, `RollingUpdate` was added as an update strategy for the `DaemonSet` object. This functionality allows you to perform serial updates to your pods based on updates to `spec.template`. In the next version, 1.7, history was added so that operators could roll back an update based on a history of revisions to `spec.template`.

You would roll back a rollout with the following `kubectl` example command:

```
$ kubectl rollout history ds example-app --revision=2
```

In fact, Kubernetes already uses these capabilities for some of its core system components. If we recall from `Chapter 1`, *Introduction to Kubernetes*, we saw `node-problem-detector` running on the nodes. This pod is actually running on every node in the cluster as `DaemonSet`. We can see this by querying DaemonSets in the `kube-system` namespace:

```
$ kubectl get ds --namespace=kube-system
```

```
NAME                        DESIRED   CURRENT   NODE-SELECTOR   AGE
node-problem-detector-v0.1  4         4         <none>          13d
```

kube-system DaemonSets

You can find more information about `node-problem-detector`, as well as `yaml`, in the following `node-problem-detector definition` listing at `http://kubernetes.io/docs/admin/node-problem/#node-problem-detector`:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-problem-detector-v0.1
  namespace: kube-system
  labels:
    k8s-app: node-problem-detector
    version: v0.1
```

```
      kubernetes.io/cluster-service: "true"
  spec:
    template:
      metadata:
        labels:
          k8s-app: node-problem-detector
          version: v0.1
          kubernetes.io/cluster-service: "true"
      spec:
        hostNetwork: true
        containers:
        - name: node-problem-detector
          image: gcr.io/google_containers/node-problem-detector:v0.1
          securityContext:
            privileged: true
          resources:
            limits:
              cpu: "200m"
              memory: "100Mi"
            requests:
              cpu: "20m"
              memory: "20Mi"
          volumeMounts:
          - name: log
            mountPath: /log
            readOnly: true
          volumes:
          - name: log
            hostPath:
              path: /var/log/
```

Node selection

As mentioned previously, we can schedule DaemonSets to run on a subset of nodes as well. This can be achieved using something called **nodeSelectors**. These allow us to constrain the nodes a pod runs on, by looking for specific labels and metadata. They simply match key-value pairs on the labels for each node. We can add our own labels or use those that are assigned by default.

The default labels are listed in the following table:

Default node labels	Description
`kubernetes.io/hostname`	This shows the hostname of the underlying instance or machine
`beta.kubernetes.io/os`	This shows the underlying operating system as a report in the Go language
`beta.kubernetes.io/arch`	This shows the underlying processor architecture as a report in the Go language
`beta.kubernetes.io/instance-type`	This is the instance type of the underlying cloud provider (cloud-only)
`failure-domain.beta.kubernetes.io/region`	This is the region of the underlying cloud provider (cloud-only)
`failure-domain.beta.kubernetes.io/zone`	This is the fault-tolerance zone of the underlying cloud provider (cloud-only)

Table 5.1 - Kubernetes default node labels

We are not limited to DaemonSets, as nodeSelectors actually work with pod definitions as well. Let's take a closer look at a job example (a slight modification of our preceding long-task example).

First, we can see these on the nodes themselves. Let's get the names of our nodes:

```
$ kubectl get nodes
```

Use a name from the output of the previous command and plug it into this one:

```
$ kubectl describe node <node-name>
```

```
Name:                   kubernetes-minion-group-1l6g
Labels:                 beta.kubernetes.io/arch=amd64
                        beta.kubernetes.io/instance-type=n1-standard-2
                        beta.kubernetes.io/os=linux
                        failure-domain.beta.kubernetes.io/region=us-central1
                        failure-domain.beta.kubernetes.io/zone=us-central1-b
                        kubernetes.io/hostname=kubernetes-minion-group-1l6g
Taints:                 <none>
CreationTimestamp:      Wed, 11 Jan 2017 07:48:16 -0500
Phase:
Conditions:
```

Excerpt from node describe

Let's now add a nickname label to this node:

```
$ kubectl label nodes <node-name> nodenickname=trusty-steve
```

If we run the `kubectl describe node` command again, we will see this label listed next to the defaults. Now, we can schedule workloads and specify this specific node. The following listing `longtask-nodeselector.yaml` is a modification of our earlier long-running task with `nodeSelector` added:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: long-task-ns
spec:
  template:
    metadata:
      name: long-task-ns
    spec:
      containers:
      - name: long-task-ns
        image: docker/whalesay
        command: ["cowsay", "Finishing that task in a jiffy"]
      restartPolicy: OnFailure
      nodeSelector:
        nodenickname: trusty-steve
```

Create the job from this listing with `kubectl create -f`.

Once that succeeds, it will create a pod based on the preceding specification. Since we have defined `nodeSelector`, it will try to run the pod on nodes that have matching labels and fail if it finds no candidates. We can find the pod by specifying the job name in our query, as follows:

```
$ kubectl get pods -a -l job-name=long-task-ns
```

We use the `-a` flag to show all pods. Jobs are short lived and once they enter the completed state, they will not show up in a basic `kubectl get pods` query. We also use the `-l` flag to specify pods with the `job-name=long-task-ns` label. This will give us the pod name, which we can push into the following command:

```
$ kubectl describe pod <Pod-Name-For-Job> | grep Node:
```

The result should show the name of the node this pod was run on. If all has gone well, it should match the node we labeled a few steps earlier with the `trusty-steve` label.

Summary

Now, you should have a good foundation of the core constructs in Kubernetes. We explored the new Deployment abstraction and how it improves on the basic ReplicationController, allowing for smooth updates and solid integration with services and autoscaling. We also looked at other types of workload in jobs and DaemonSets. You learned how to run short-running or batch tasks, as well as how to run agents on every node in our cluster. Finally, we took a brief look at node selection and how that can be used to filter the nodes in the cluster used for our workloads.

We will build on what you learned in this chapter and look at stateful applications in the next chapter, exploring both critical application components and the data itself.

Questions

1. Name four use cases for Kubernetes deployments
2. Which element of a deployment definition tells the deployment which pod to manage?
3. Which flag do you need to activate in order to see the history of your changes?
4. Which underlying mechanism (a Kubernetes object, in fact) does a Deployment use in order to update your container images?
5. What's the name of the technology that lets your pods scale up and down according to CPU load?
6. Which type of workload should you run for an ephemeral, short-lived task?
7. What's the purpose of a `DaemonSet`?

5
Exploring Kubernetes Storage Concepts

In order to power modern microservices and other stateless applications, Kubernetes operators need to have a way to manage stateful data storage on the cluster. While it's advantageous to maintain as much state as possible outside of the cluster in dedicated database clusters as a part of cloud-native service offerings, there's often a need to keep a statement of record or state cluster for stateless and ephemeral services. We'll explore what's considered a more difficult problem in the container orchestration and scheduling world: managing locality-specific, mutable data in a world that relies on declarative state, decoupling physical devices from logical objects, and immutable approaches to system updates. We'll explore strategies for setting up reliable, replicated storage for modern database engines.

In this chapter, we will discuss how to attach persistent volumes and create storage for stateful applications and data. We will walk through storage concerns and how we can persist data across pods and the container life cycle. We will explore the `PersistentVolumes` types, as well as `PersistentVolumeClaim`. Finally, we will take a look at StatefulSets and how to use dynamic volume provisioning.

The following topics will be covered in the chapter:

- Persistent storage
- `PersistentVolumes`
- `PersistentVolumeClaim`
- Storage Classes
- Dynamic volume provisioning
- StatefulSets

Technical requirements

You'll need to have a running Kubernetes cluster to go through these examples. Please start your cluster up on your cloud provider of choice, or a local Minikube instance.

The code for this repository can be found here: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter05`.

Persistent storage

So far, we only worked with workloads that we could start and stop at will, with no issue. However, real-world applications often carry state and record data that we prefer (even insist) not to lose. The transient nature of containers themselves can be a big challenge. If you recall our discussion of layered filesystems in `Chapter 1`, *Introduction to Kubernetes*, the top layer is writable. (It's also frosting, which is delicious.) However, when the container dies, the data goes with it. The same is true for crashed containers that Kubernetes restarts.

This is where volumes or disks come into play. Volumes exist outside the container and are coupled to the pod, which allows us to save our important data across containers outages. Further more, if we have a volume at the pod level, data can be shared between containers in the same application stack and within the same pod. A volume itself on Kubernetes is a directory, which the Pod provides to the containers running on it. There are a number of different volume types available at `spec.volumes`, which we'll explore, and they're mounted into containers with the `spec.containers.volumeMounts` parameter.

To see all the types of volumes available, visit `https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes`.

Docker itself has some support for volumes, but Kubernetes gives us persistent storage that lasts beyond the lifetime of a single container. The volumes are tied to pods and live and die with those pods. Additionally, a pod can have multiple volumes from a variety of sources. Let's take a look at some of these sources.

Temporary disks

One of the easiest ways to achieve improved persistence amid container crashes and data sharing within a pod is to use the `emptydir` volume. This volume type can be used with either the storage volumes of the node machine itself or an optional RAM disk for higher performance.

Again, we improve our persistence beyond a single container, but when a pod is removed, the data will be lost. A machine reboot will also clear any data from RAM-type disks. There may be times when we just need some shared temporary space or have containers that process data and hand it off to another container before they die. Whatever the case, here is a quick example of using this temporary disk with the RAM-backed option.

Open your favorite editor and create a `storage-memory.yaml` file and type the following code:

```
apiVersion: v1
kind: Pod
metadata:
  name: memory-pd
spec:
  containers:
  - image: nginx:latest
    ports:
    - containerPort: 80
    name: memory-pd
    volumeMounts:
    - mountPath: /memory-pd
      name: memory-volume
  volumes:
  - name: memory-volume
    emptyDir:
      medium: Memory
```

The preceding example is probably second nature by now, but we will once again issue a `create` command followed by an `exec` command to see the folders in the container:

```
$ kubectl create -f storage-memory.yaml
$ kubectl exec memory-pd -- ls -lh | grep memory-pd
```

This will give us a Bash shell in the container itself. The `ls` command shows us a `memory-pd` folder at the top level. We use `grep` to filter the output, but you can run the command without `| grep memory-pd` to see all folders:

```
/home/k8s/nodejs# kubectl.sh exec memory-pd -- ls -lh | grep memory
drwxrwxrwt  2 root root   40 Oct 24 15:21 memory-pd
```

Temporary storage inside a container

Again, this folder is temporary as everything is stored in the node's (minion's) RAM. When the node gets restarted, all the files will be erased. We will look at a more permanent example next.

Cloud volumes

Let's move on to something more robust. There are two types of PersistentVolumes that we'll touch base with in order to explain how you can use AWS's and GCE's block storage engines to provide stateful storage for your Kubernetes cluster. Given that many companies have already made significant investment in cloud infrastructure, we'll get you up and running with two key examples. You can consider these types of volumes or persistent volumes as storage classes. These are different from the `emptyDir` that we created before, as the contents of a GCE persistent disk or AWS EBS volume will persist even if a pod is removed. Looking ahead, this provides operators with the clever feature of being able to pre-populate data in these drives and can also be switched between pods.

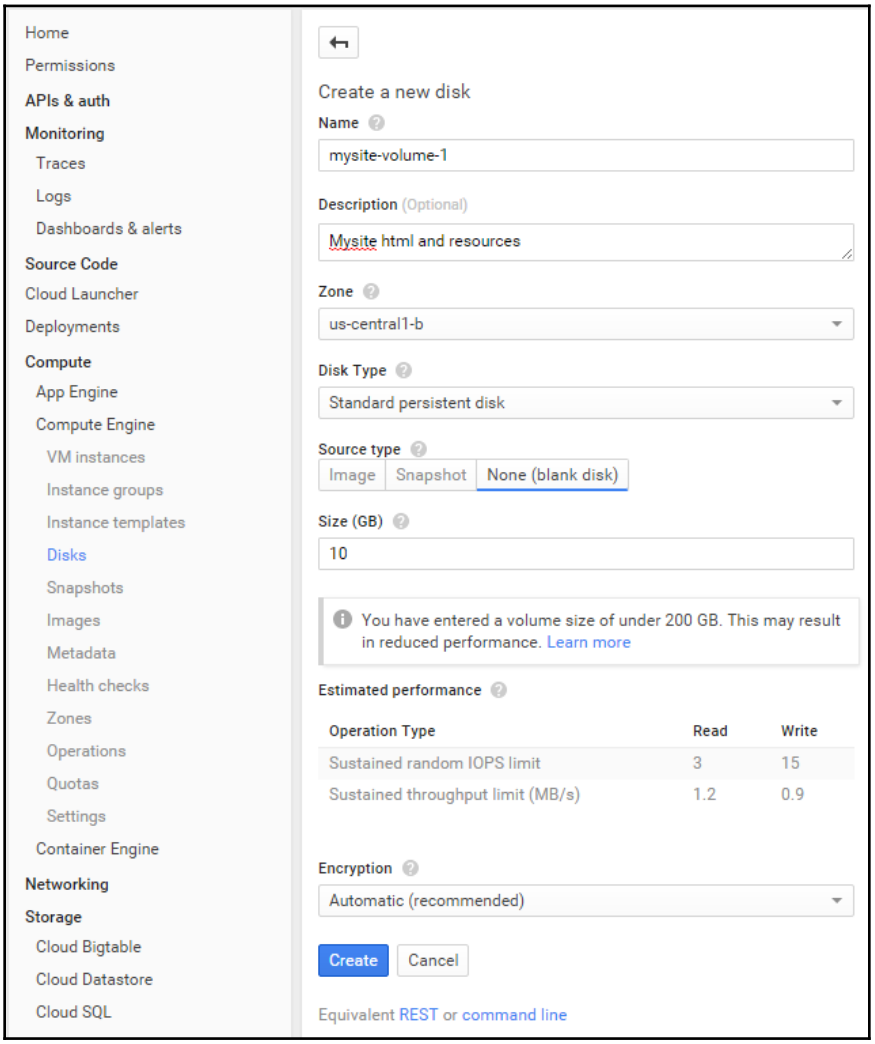
GCE Persistent Disks

Let's mount a `gcePersistentDisk` first. You can see more information about these drives here: `https://cloud.google.com/compute/docs/disks/`.

Google Persistent Disk is durable and high performance block storage for the Google Cloud Platform. Persistent Disk provides SSD and HDD storage, which can be attached to instances running in either Google Compute Engine or Google Container Engine. Storage volumes can be transparently resized, quickly backed up, and offer the ability to support simultaneous readers.

You'll need to create a Persistent Disk using the GCE GUI, API, or CLI before we're able to use it in our cluster, so let's get started:

1. From the console, in **Compute Engine**, go to **Disks**. On this new screen, click on the **Create Disk** button. We'll be presented with a screen similar to the following **GCE new persistent disk** screenshot:



GCE new persistent disk

2. Choose a name for this volume and give it a brief description. Make sure that **Zone** is the same as the nodes in your cluster. GCE Persistent Disks can only be attached to machines in the same zone.
3. Enter `mysite-volume-1` in the **Name** field. Choose a zone matching at least one node in your cluster. Choose **None (blank disk)** for **Source type** and give `10` (10 GB) as the value in **Size (GB)**. Finally, click on **Create**:

The nice thing about Persistent Disks on GCE is that they allow for mounting to multiple machines (nodes in our case). However, when mounting to multiple machines, the volume must be in read-only mode. So, let's first mount this to a single pod, so we can create some files. Use the following code to make a `storage-gce.yaml` file to create a pod that will mount the disk in read/write mode:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-gce
spec:
  containers:
  - image: nginx:latest
    ports:
    - containerPort: 80
    name: test-gce
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: gce-pd
  volumes:
  - name: gce-pd
    gcePersistentDisk:
      pdName: mysite-volume-1
      fsType: ext4
```

First, let's issue a `create` command followed by a `describe` command to find out which node it is running on:

```
$ kubectl create -f storage-gce.yaml
$ kubectl describe pod/test-gce
```

Note the node and save the pod IP address for later. Then, open an SSH session into that node:

```
Name:           test-gce
Namespace:      default
Node:           kubernetes-minion-group-zwpm/10.128.0.4
Start Time:     Sun, 15 Jan 2017 16:51:02 -0500
Labels:         <none>
Status:         Running
IP:             10.244.4.5
Controllers:    <none>
Containers:
  test-gce:
    Container ID:       docker://15871d81eb72557cc230df70a5c724617289d710a550da66e4dfaf7083
    Image:              nginx:latest
    Image ID:           docker://sha256:01f818af747d88b4ebca7cdabd0c581e406e0e790be72678d25
    Port:               80/TCP
    Requests:
      cpu:              100m
    State:              Running
      Started:          Sun, 15 Jan 2017 16:53:00 -0500
    Ready:              True
    Restart Count:      0
    Volume Mounts:
      /usr/share/nginx/html from gce-pd (rw)
      /var/run/secrets/kubernetes.io/serviceaccount from default-token-728d1 (ro)
    Environment Variables:      <none>
Conditions:
  Type          Status
  Initialized   True
  Ready         True
  PodScheduled  True
Volumes:
  gce-pd:
    Type:       GCEPersistentDisk (a Persistent Disk resource in Google Compute Engine)
    PDName:     mysite-volume-1
    FSType:     ext4
    Partition:  0
    ReadOnly:   false
  default-token-728d1:
    Type:       Secret (a volume populated by a Secret)
    SecretName: default-token-728d1
QoS Class:      Burstable
Tolerations:    <none>
```

Pod described with persistent disk

Type the following command:

```
$ gcloud compute --project "<Your project ID>" ssh --zone "<your gce zone>"
"<Node running test-gce pod>"
```

Since we've already looked at the volume from inside the running container, let's access it directly from the node (minion) itself this time. We will run a `df` command to see where it is mounted, but we will need to switch to root first:

```
$ sudo su -
$ df -h | grep mysite-volume-1
```

As you can see, the GCE volume is mounted directly to the node itself. We can use the mount path listed in the output of the earlier `df` command. Use `cd` to change to the folder now. Then, create a new file named `index.html` with your favorite editor:

```
$ cd /var/lib/kubelet/plugins/kubernetes.io/gce-pd/mounts/mysite-volume-1
$ vi index.html
```

Enter a quaint message, such as `Hello from my GCE PD!`. Now, save the file and exit the editor. If you recall from the `storage-gce.yaml` file, the Persistent Disk is mounted directly to the nginx HTML directory. So, let's test this out while we still have the SSH session open on the node. Do a simple `curl` command to the pod IP we wrote down earlier:

```
$ curl <Pod IP from Describe>
```

You should see `Hello from my GCE PD!` or whatever message you saved in the `index.html` file. In a real-world scenario, we can use the volume for an entire website or any other central storage. Let's take a look at running a set of load balanced web servers all pointing to the same volume.

First, leave the SSH session with two `exit` commands. Before we proceed, we will need to remove our `test-gce` pod so that the volume can be mounted read-only across a number of nodes:

```
$ kubectl delete pod/test-gce
```

Now, we can create an `ReplicationController` that will run three web servers, all mounting the same Persistent Disk, as follows. Save the following code as the `http-pd-controller.yaml` file:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: http-pd
  labels:
    name: http-pd
spec:
  replicas: 3
```

```
  selector:
    name: http-pd
  template:
    metadata:
      name: http-pd
      labels:
        name: http-pd
    spec:
      containers:
      - image: nginx:latest
        ports:
        - containerPort: 80
        name: http-pd
        volumeMounts:
        - mountPath: /usr/share/nginx/html
          name: gce-pd
      volumes:
      - name: gce-pd
        gcePersistentDisk:
          pdName: mysite-volume-1
          fsType: ext4
          readOnly: true
```

Let's also create an external service and save it as the `http-pd-service.yaml` file, so we can see it from outside the cluster:

```
apiVersion: v1
kind: Service
metadata:
  name: http-pd
  labels:
    name: http-pd
spec:
  type: LoadBalancer
  ports:
  - name: http
    protocol: TCP
    port: 80
  selector:
    name: http-pd
```

Go ahead and create these two resources now. Wait a few moments for the external IP to get assigned. After this, a `describe` command will give us the IP we can use in a browser:

```
$ kubectl describe service/http-pd
```

The following screenshot is the result of the preceding command:

```
Name:                   http-pd
Namespace:              default
Labels:                 name=http-pd
Selector:               name=http-pd
Type:                   LoadBalancer
IP:                     10.0.118.195
LoadBalancer Ingress:   130.211.186.84
Port:                   http    80/TCP
NodePort:               http    32429/TCP
Endpoints:              10.244.2.15:80,10.244.2.16:80,10.244.3.5:80
Session Affinity:       None
No events.
```

K8s service with GCE PD shared across three pods

If you don't see the `LoadBalancer Ingress` field yet, it probably needs more time to get assigned. Type the IP address from `LoadBalancer Ingress` into a browser, and you should see your familiar `index.html` file show up with the text we entered previously!

AWS Elastic Block Store

K8s also supports AWS **Elastic Block Store** (**EBS**) volumes. Like the GCE Persistent Disks, EBS volumes are required to be attached to an instance running in the same availability zone. A further limitation is that EBS can only be mounted to a single instance at one time. Similarly to before, you'll need to create an EBS volume using API calls, the CLI, or you'll need to log in to the GUI manually and create the volume referenced by `volumeID`. If you're authorized in the AWS CLI, you can use the following command to create a volume:

```
$ aws ec2 create-volume --availability-zone=us-west-1a eu-west-1a --size=20
--volume-type=gp2
```

Make sure that your volume is created in the same region as your Kubernetes cluster!

For brevity, we will not walk through an AWS example, but a sample YAML file is included to get you started. Again, remember to create the EBS volume before your pod. Save the following code as the `storage-aws.yaml` file:

```
apiVersion: v1
kind: Pod
metadata:
  name: test-aws
spec:
  containers:
  - image: nginx:latest
```

```
    ports:
    - containerPort: 80
    name: test-aws
    volumeMounts:
    - mountPath: /usr/share/nginx/html
      name: aws-pd
  volumes:
  - name: aws-pd
    awsElasticBlockStore:
      volumeID: aws://<availability-zone>/<volume-id>
      fsType: ext4
```

Other storage options

Kubernetes supports a variety of other types of storage volumes. A full list can be found here: `https://kubernetes.io/docs/concepts/storage/volumes/#types-of-volumes`.

Here are a few that may be of particular interest:

- `nfs`: This type allows us to mount a **Network File Share** (**NFS**), which can be very useful for both persisting the data and sharing it across the infrastructure
- `gitrepo`: As you might have guessed, this option clones a Git repository into a new and empty folder

PersistentVolumes and Storage Classes

Thus far, we've seen examples of directly provisioning the storage within our pod definitions. This works quite well if you have full control over your cluster and infrastructure, but at larger scales, application owners will want to use storage that is managed separately. Typically, a central IT team or the cloud provider will take care of the details behind provisioning storage and leave the application owners to worry about their primary concern, the application itself. This separation of concerns and duties in Kubernetes allows you to structure your engineering focus around a storage subsystem that can be managed by a distinct group of engineers.

In order to accommodate this, we need some way for the application to specify and request storage without being concerned with how that storage is provided. This is where `PersistentVolumes` and `PersistentVolumeClaim` come into play.

`PersistentVolumes` are similar to the volumes we created earlier, but they are provided by the cluster administrator and are not dependent on a particular pod. `PersistentVolumes` are a resource that's provided to the cluster just like any other object. The Kubernetes API provides an interface for this object in the form of NFS, EBS Persistent Disks, or any other volume type described before. Once the volume has been created, you can use `PersistentVolumeClaims` to request storage for your applications.

`PersistentVolumeClaims` is an abstraction that allows users to specify the details of the storage needed. We can defined the amount of storage, as well as the access type, such as `ReadWriteOnce` (read and write by one node), `ReadOnlyMany` (read-only by multiple nodes), and `ReadWriteMany` (read and write by many nodes). The cluster operators are in charge of providing a wide variety of storage options for application operators in order to meet requirements across a number of different access modes, sizes, speeds, and durability without requiring the end users to know the details of that implementation. The modes supported by cluster operators is dependent on the backing storage provider. For example, we saw in the AWS `aws-ebs` example that mounting to multiple nodes was not an option, while with GCP Persistent Disks could be shared among several nodes in read-only mode.

Additionally, Kubernetes provides two other methods for specifying certain groupings or types of storage volumes. The first is the use of selectors, as we have seen previously for pod selection. Here, labels can be applied to storage volumes and then claims can reference these labels to further filter the volume they are provided. Second, Kubernetes has the concept of `StorageClass`, which allows us specify a storage provisioner and parameters for the types of volumes it provisions.

`PersistentVolumes` and `PersistentVolumeClaims` have a life cycle that involves the following phases:

- Provisioning
- Static or dynamic
- Binding
- Using
- Reclaiming
- Delete, retain, or recycle

We will dive into Storage Classes in the next section, but here is a quick example of a `PersistentVolumeClaim` for illustration purposes. You can see in the annotations that we request `1Gi` of storage in `ReadWriteOnce` mode with a `StorageClass` of `solidstate` and a label of `aws-storage`. Save the following code as the `pvc-example.yaml` file:

```
kind: PersistentVolumeClaim
apiVersion: v1
```

```
metadata:
  name: demo-claim
spec:
  accessModes:
  - ReadWriteOnce
  volumeMode: Filesystem
  resources:
    requests:
      storage: 1Gi
  storageClassName: ssd
  selector:
    matchLabels:
      release: "aws-storage"
  matchExpressions:
      - {key: environment, operator: In, values: [dev, stag, uat]}
```

As of Kubernetes version 1.8, there's also alpha support for expanding `PersistentVolumeClaim` for `gcePersistentDisk`, `awsElasticBlockStore`, `Cinder`, `glusterfs`, and `rbd` volume claim types. These are similar to the thin provisioning that you may have seen with systems such as VMware, and they allow for resizing of a storage class via the `allowVolumeExpansion` field as long as you're running either XFS or Ext3/Ext4 filesystems. Here's a quick example of what that looks like:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
 name: Cinder-volume-01
provisioner: kubernetes.io/cinder
parameters:
 resturl: "http://192.168.10.10:8080"
 restuser: ""
 secretNamespace: ""
 secretName: ""
allowVolumeExpansion: true
```

Dynamic volume provisioning

Now that we've explored how to build from volumes, storage classes, persistent volumes, and persistent volume claims, let's take a look at how to make that all dynamic and take advantage of the built-in scaling of the cloud! Dynamic provisioning removes the need for pre-crafted storage; it relies on requests from application users instead. You use the `StorageClass` API object to create dynamic resources.

First, we can create a manifest that will define the type of storage class that we'll use for our dynamic storage. We'll use a vSphere example here to try out another storage class:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: durable-medium
provisioner: kubernetes.io/vsphere-volume
parameters:
  type: thin
```

Once we have the manifest, we can use this storage by including it as a class in a new `PersistentVolumeClaim`. You may remember this as `volume.beta.kubernetes.io/storage-class` in earlier, pre-1.6 versions of Kubernetes, but now you can simply include this property in the `PersistentVolumeClaim` object. Keep in mind that the value of `storageClassName` must match the available, dynamic `StorageClass` that the cluster operators have provided. Here's an example of that:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: webtier-vclaim-01
spec:
 accessModes:
   - ReadWriteMany
 storageClassName: durable-medium
 resources:
   requests:
     storage: 20Gi
```

When this claim is removed, the storage is dynamically deleted. You can make this a cluster default by ensuring that the `DefaultStorageClass` admission controller is turned on, and after you ensure that one `StorageClass` object is set to default.

StatefulSets

The purpose of StatefulSets is to provide some consistency and predictability to application deployments with stateful data. Thus far, we have deployed applications to the cluster, defining loose requirements around required resources such as compute and storage. The cluster has scheduled our workload on any node that can meet these requirements. While we can use some of these constraints to deploy in a more predictable manner, it will be helpful if we had a construct built to help us provide this consistency.

StatefulSets were set to GA in 1.6 as we went to press. There were previously beta in version 1.5 and were known as Pet Sets prior to that (alpha in 1.3 and 1.4).

This is where StatefulSets come in. StatefulSets provide us first with numbered and reliable naming for both network access and storage claims. The pods themselves are named with the following convention, where `N` is from 0 to the number of replicas:

```
"Name of Set"-N
```

This means that a StatefulSet called `db` with three replicas will create the following pods:

```
db-0
db-1
db-2
```

This gives Kubernetes a way to associate network names and `PersistentVolumes` with specific pods. Additionally, it also serves to order the creation and termination of pods. Pod will be started from `0` to `N` and terminated from `N` to `0`.

A stateful example

Let's take a look at an example of a stateful application. First, we will want to create and use a `StorageClass`, as we discussed earlier. This will allow us to hook into the Google Cloud Persistent Disk provisioner. The Kubernetes community is building provisioners for a variety of `StorageClasses`, including GCP and AWS. Each provisioner has its own set of parameters available. Both GCP and AWS providers let you choose the type of disk (solid-state, standard, and so on) as well as the fault zone that is needed to match the pod attaching to it. AWS additionally allows you to specify encryption parameters as well as IOPs for provisioned IOPs volumes. There are a number of other provisioners in the works, including Azure and a variety of non-cloud options. Save the following code as `solidstate-sc.yaml` file:

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
  name: solidstate
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
  zone: us-central1-b
```

Use the following command with the preceding listing to create a `StorageClass` kind of SSD drive in `us-central1-b`:

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

Next, we will create a `StatefulSet` kind with our trusty `httpwhalesay` demo. While this application does include any real state, we can see the storage claims and explore the communication path as shown in the listing `sayhey-statefulset.yaml`:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: whaleset
spec:
  serviceName: sayhey-svc
  replicas: 3
  template:
    metadata:
      labels:
        app: sayhey
    spec:
      terminationGracePeriodSeconds: 10
      containers:
      - name: sayhey
        image: jonbaier/httpwhalesay:0.2
        command: ["node", "index.js", "Whale it up!."]
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
      annotations:
        volume.beta.kubernetes.io/storage-class: solidstate
    spec:
      accessModes: [ "ReadWriteOnce" ]
      resources:
        requests:
          storage: 1Gi
```

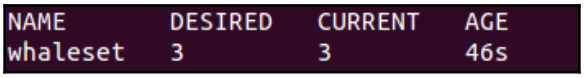
Use the following command to start the creation of this StatefulSet. If you observe pod creation closely, you will see it create `whaleset-0`, `whaleset-1`, and `whaleset-2` in succession:

```
$ kubectl create -f sayhey-statefulset.yaml
```

Immediately after this, we can see our StatefulSet and the corresponding pods using the familiar `get` subcommand:

```
$ kubectl get statefulsets
$ kubectl get pods
```

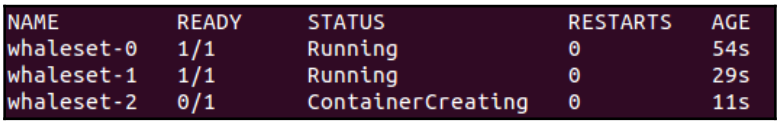
These pods should create an output similar to the following images:



```
NAME       DESIRED   CURRENT   AGE
whaleset   3         3         46s
```

StatefulSet listing

The `get pods` output will show the following:



```
NAME          READY     STATUS              RESTARTS   AGE
whaleset-0    1/1       Running             0          54s
whaleset-1    1/1       Running             0          29s
whaleset-2    0/1       ContainerCreating   0          11s
```

Pods created by StatefulSet

Depending on your timing, the pods may still be being created. As you can see in the preceding screenshot, the third container is still being spun up.

We can also see the volumes the set has created and claimed for each pod. First are the `PersistentVolumes` themselves:

```
$ kubectl get pv
```

The preceding command should show the three `PersistentVolumes` named `www-whaleset-N`. We notice the size is `1Gi` and the access mode is set to **ReadWriteOnce** (**RWO**), just as we defined in our `StorageClass`:

```
NAME             STATUS    VOLUME                                     CAPACITY   ACCESSMODES
 AGE
www-whaleset-0   Bound     pvc-43346a3d-e024-11e6-af6d-42010a800002   1Gi        RWO
 4m
www-whaleset-1   Bound     pvc-43381dc9-e024-11e6-af6d-42010a800002   1Gi        RWO
 4m
www-whaleset-2   Bound     pvc-433a3864-e024-11e6-af6d-42010a800002   1Gi        RWO
 4m
```

The PersistentVolumes listing

Next, we can look at the `PersistentVolumeClaim` that reserves the volumes for each pod:

```
$ kubectl get pvc
```

The following is the output of the preceding command:

```
NAME                                       CAPACITY   ACCESSMODES   RECLAIMPOLICY   STATUS
CLAIM                     REASON    AGE
pvc-43346a3d-e024-11e6-af6d-42010a800002   1Gi        RWO           Delete          Bound
default/www-whaleset-0                    4m
pvc-43381dc9-e024-11e6-af6d-42010a800002   1Gi        RWO           Delete          Bound
default/www-whaleset-1                    4m
pvc-433a3864-e024-11e6-af6d-42010a800002   1Gi        RWO           Delete          Bound
default/www-whaleset-2                    4m
```

The PersistentVolumeClaim listing

You'll notice many of the same settings here as with the `PersistentVolumes` themselves. You might also notice the end of the claim name (or `PersistentVolumeClaim` name in the previous listing) looks like `www-whaleset-N`. `www` is the mount name we specified in the preceding YAML definition. This is then appended to the pod name to create the actual `PersistentVolume` and `PersistentVolumeClaim` name. One more area we can ensure that the proper disk is linked with it's matching pod.

Another area where this alignment is important is in network communication. StatefulSets also provide consistent naming here. Before we can do this, let's create a service endpoint `sayhey-svc.yaml`, so we have a common entry point for incoming requests:

```
apiVersion: v1
kind: Service
metadata:
  name: sayhey-svc
  labels:
    app: sayhey
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: sayhey
```

```
$ kubectl create -f sayhey-svc.yaml
```

Now, let's open a shell in one of the pods and see if we can communicate with another in the set:

```
$ kubectl exec whaleset-0 -i -t bash
```

The preceding command gives us a bash shell in the first `whaleset` pod. We can now use the service name to make a simple HTTP request. We can use both the short name, `sayhey-svc`, and the fully qualified name, `sayhey-svc.default.svc.cluster.local`:

```
$ curl sayhey-svc
$ curl sayhey-svc.default.svc.cluster.local
```

You'll see an output similar to the following screenshot. The service endpoint acts as a common communication point for all three pods:

HTTP whalesay curl output (whalesay-0 Pod)

Now, let's see if we can communicate with a specific pod in the StatefulSet. As we noticed earlier, the StatefulSet named the pods in an orderly manner. It also gives them hostnames in a similar fashion so that there is a specific DNS entry for each pod in the set. Again, we will see the convention of `"Name of Set"-N` and then add the fully qualified service URL. The following example shows this for `whaleset-1`, which is the second pod in our set:

```
$ curl whaleset-1.sayhey-svc.default.svc.cluster.local
```

Running this command from our existing Bash shell in `whaleset-0` will show us the output from `whaleset-1`:

HTTP whalesay curl output (whalesay-1 Pod)

You can exit out of this shell now with `exit`.

For learning purposes, it may also be instructive to describe some of the items from this section in more detail. For example, `kubectl describe svc sayhey-svc` will show us all three pod IP address in the service endpoints.

Summary

In this chapter, we explored a variety of persistent storage options and how to implement them with our pods. We looked at `PersistentVolumes` and also `PersistentVolumeClaim`, which allow us to separate storage provisioning and application storage requests. Additionally, we looked at `StorageClasses` for provisioning groups of storage according to a specification.

We also explored the new StatefulSets abstraction and learned how we can deploy stateful applications in a consistent and ordered manner. In the next chapter, we will look at how to integrate Kubernetes with Continuous Integration and Delivery pipelines.

Questions

1. Name four kinds of volumes that Kubernetes supports
2. What's the parameter that you can use to enable a simple, semi-persistent temporary disk?
3. Name two backing technologies that make `PersistentVolumes` easy to implement with **Cloud Service Providers** (**CSPs**)
4. What's a good reason for creating different types of `StorageClasses`?
5. Name two phases in the `PersistentVolume` and `PersistentVolumeClaim` lifecycle
6. Which Kubernetes object is used to provide a stateful storage-based application?

Further reading

- If you'd like to know more about dynamic storage provisioning, please read this blog post: `https://kubernetes.io/blog/2017/03/dynamic-provisioning-and-storage-classes-kubernetes/`
- If you'd like to know more about the cutting edge of the **Storage Special Interest Group** (**SIG**), you can read about it here: `https://github.com/kubernetes/community/tree/master/sig-storage`

6
Application Updates, Gradual Rollouts, and Autoscaling

This chapter will expand upon the core concepts, and show you how to roll out updates and test new features of your application with minimal disruption to uptime. It will cover the basics of doing application updates, gradual rollouts, and A/B testing. In addition, we will look at scaling the Kubernetes cluster itself.

In version 1.2, Kubernetes released a Deployments API. Deployments are the recommended way to deal with scaling and application updates going forward. As mentioned in previous chapters, `ReplicationControllers` are no longer the recommended manner for managing application updates. However, as they're still core functionality for many operators, we will explore rolling updates in this chapter as an introduction to the scaling concept and then dive into the preferred method of using Deployments in the next chapter.

We'll also investigate the functionality of Helm and Helm Charts that will help you manage Kubernetes resources. Helm is a way to manage packages in Kubernetes much in the same way that `apt/yum` manage code in the Linux ecosystem. Helm also lets you share your applications with others, and most importantly create reproducible builds of Kubernetes applications.

In this chapter, we will cover the following topics:

- Application scaling
- Rolling updates
- A/B testing
- Application autoscaling
- Scaling up your cluster
- Using Helm

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in, or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes over the web: `https://labs.play-with-k8s.com/`.

Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter06`.

Example setup

Before we start exploring the various capabilities built into Kubernetes for scaling and updates, we will need a new example environment. We are going to use a variation of our previous container image with a blue background (refer to the *v0.1 and v0.2 (side by side)* image, later in this chapter, for a comparison). We have the following code in the `pod-scaling-controller.yaml` file:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-scale
  labels:
    name: node-js-scale
spec:
  replicas: 1
  selector:
    name: node-js-scale
  template:
    metadata:
      labels:
        name: node-js-scale
    spec:
      containers:
      - name: node-js-scale
        image: jonbaier/pod-scaling:0.1
        ports:
        - containerPort: 80
```

Save the following code as `pod-scaling-service.yaml` file:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-scale
  labels:
    name: node-js-scale
spec:
  type: LoadBalancer
  sessionAffinity: ClientIP
  ports:
  - port: 80
  selector:
    name: node-js-scale
```

Create these services with the following commands:

```
$ kubectl create -f pod-scaling-controller.yaml
$ kubectl create -f pod-scaling-service.yaml
```

The public IP address for the service may take a moment to create.

Scaling up

Over time, as you run your applications in the Kubernetes cluster, you will find that some applications need more resources, whereas others can manage with fewer resources. Instead of removing the entire `ReplicationControllers` (and associated pods), we want a more seamless way to scale our application up and down.

Thankfully, Kubernetes includes a `scale` command, which is suited specifically for this purpose. The `scale` command works both with `ReplicationControllers` and the new Deployments abstraction. For now, we will explore its use with `ReplicationControllers`. In our new example, we have only one replica running. You can check this with a `get pods` command:

```
$ kubectl get pods -l name=node-js-scale
```

Let's try scaling that up to three with the following command:

```
$ kubectl scale --replicas=3 rc/node-js-scale
```

If all goes well, you'll simply see the `scaled` word on the output of your Terminal window.

Optionally, you can specify the `--current-replicas` flag as a verification step. The scaling will only occur if the actual number of replicas currently running matches this count.

After listing our pods once again, we should now see three pods running with a name similar to `node-js-scale-XXXXX`, where the `X` characters are a random string.

You can also use the `scale` command to reduce the number of replicas. In either case, the `scale` command adds or removes the necessary pod replicas, and the service automatically updates and balances across new or remaining replicas.

Smooth updates

The scaling of our application up and down as our resource demands change is useful for many production scenarios, but what about simple application updates? Any production system will have code updates, patches, and feature additions. These could be occurring monthly, weekly, or even daily. Making sure that we have a reliable way to push out these changes without interruption to our users is a paramount consideration.

Once again, we benefit from the years of experience the Kubernetes system is built on. There is built-in support for rolling updates with the 1.0 version. The `rolling-update` command allows us to update entire `ReplicationControllers` or just the underlying Docker image used by each replica. We can also specify an update interval, which will allow us to update one pod at a time and wait until proceeding to the next.

Let's take our scaling example and perform a rolling update to the 0.2 version of our container image. We will use an update interval of 2 minutes, so we can watch the process as it happens in the following way:

```
$ kubectl rolling-update node-js-scale --image=jonbaier/pod-scaling:0.2 --
update-period="2m"
```

You should see some text about creating a new `ReplicationControllers` named `node-js-scale-XXXXX`, where the `X` characters will be a random string of numbers and letters. In addition, you will see the beginning of a loop that starts one replica of the new version and removes one from the existing `ReplicationControllers`. This process will continue until the new `ReplicationControllers` has the full count of replicas running.

If we want to follow along in real time, we can open another Terminal window and use the `get pods` command, along with a label filter, to see what's happening:

```
$ kubectl get pods -l name=node-js-scale
```

This command will filter for pods with `node-js-scale` in the name. If you run this after issuing the `rolling-update` command, you should see several pods running as it creates new versions and removes the old ones one by one.

The full output of the previous `rolling-update` command should look something like this screenshot:

```
Creating node-js-scale-10ea08ff9a118ac6a93f85547ed2d8f6
At beginning of loop: node-js-scale replicas: 2, node-js-scale-10ea08ff9a118ac6a
93f85547ed2d8f6 replicas: 1
Updating node-js-scale replicas: 2, node-js-scale-10ea08ff9a118ac6a93f85547ed2d8
f6 replicas: 1
At end of loop: node-js-scale replicas: 2, node-js-scale-10ea08ff9a118ac6a93f855
47ed2d8f6 replicas: 1
At beginning of loop: node-js-scale replicas: 1, node-js-scale-10ea08ff9a118ac6a
93f85547ed2d8f6 replicas: 2
Updating node-js-scale replicas: 1, node-js-scale-10ea08ff9a118ac6a93f85547ed2d8
f6 replicas: 2
At end of loop: node-js-scale replicas: 1, node-js-scale-10ea08ff9a118ac6a93f855
47ed2d8f6 replicas: 2
At beginning of loop: node-js-scale replicas: 0, node-js-scale-10ea08ff9a118ac6a
93f85547ed2d8f6 replicas: 3
Updating node-js-scale replicas: 0, node-js-scale-10ea08ff9a118ac6a93f85547ed2d8
f6 replicas: 3
At end of loop: node-js-scale replicas: 0, node-js-scale-10ea08ff9a118ac6a93f855
47ed2d8f6 replicas: 3
Update succeeded. Deleting old controller: node-js-scale
Renaming node-js-scale-10ea08ff9a118ac6a93f85547ed2d8f6 to node-js-scale
node-js-scale
```

The scaling output

As we can see here, Kubernetes is first creating a new `ReplicationController` named `node-js-scale-10ea08ff9a118ac6a93f85547ed28f6`. K8s then loops through one by one, creating a new pod in the new controller and removing one from the old. This continues until the new controller has the full replica count and the old one is at zero. After this, the old controller is deleted and the new one is renamed with the original controller's name.

If you run a `get pods` command now, you'll notice that the pods still all have a longer name. Alternatively, we could have specified the name of a new controller in the command, and Kubernetes will create a new `ReplicationControllers` and pods using that name. Once again, the controller of the old name simply disappears after the update is completed. I recommend that you specify a new name for the updated controller to avoid confusion in your pod naming down the line. The same `update` command with this method will look like this:

```
$ kubectl rolling-update node-js-scale node-js-scale-v2.0 --
image=jonbaier/pod-scaling:0.2 --update-period="2m"
```

Using the static external IP address from the service we created in the first section, we can open the service in a browser. We should see our standard container information page. However, you'll notice that the title now says **Pod Scaling v0.2** and the background is light yellow:

v0.1 and v0.2 (side by side)

It's worth noting that, during the entire update process, we've only been looking at pods and `ReplicationControllers`. We didn't do anything with our service, but the service is still running fine and now directing to the new version of our pods. This is because our service is using label selectors for membership. Because both our old and new replicas use the same labels, the service has no problem using the new pods to service requests. The updates are done on the pods one by one, so it's seamless for the users of the service.

Testing, releases, and cutovers

The rolling update feature can work well for a simple blue-green deployment scenario. However, in a real-world blue-green deployment with a stack of multiple applications, there can be a variety of inter-dependencies that require in-depth testing. The `update-period` command allows us to add a `timeout` flag where some testing can be done, but this will not always be satisfactory for testing purposes.

Similarly, you may want partial changes to persist for a longer time and all the way up to the load balancer or service level. For example, you may wish to run an A/B test on a new user interface feature with a portion of your users. Another example is running a canary release (a replica in this case) of your application on new infrastructure, such as a newly added cluster node.

Let's take a look at an A/B testing example. For this example, we will need to create a new service that uses `sessionAffinity`. We will set the affinity to `ClientIP`, which will allow us to forward clients to the same backend pod. The following listing `pod-AB-service.yaml` is the key if we want a portion of our users to see one version while others see another:

```
apiVersion: v1
kind: Service
metadata:
  name: node-js-scale-ab
  labels:
    service: node-js-scale-ab
spec:
  type: LoadBalancer
  ports:
  - port: 80
  sessionAffinity: ClientIP
  selector:
    service: node-js-scale-ab
```

Create this service as usual with the `create` command, as follows:

```
$ kubectl create -f pod-AB-service.yaml
```

This will create a service that will point to our pods running both version 0.2 and 0.3 of the application. Next, we will create the two `ReplicationControllers` that create two replicas of the application. One set will have version 0.2 of the application, and the other will have version 0.3, as shown in the listing `pod-A-controller.yaml`and `pod-B-controller.yaml`:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-scale-a
  labels:
    name: node-js-scale-a
    version: "0.2"
    service: node-js-scale-ab
spec:
  replicas: 2
  selector:
    name: node-js-scale-a
    version: "0.2"
    service: node-js-scale-ab
  template:
    metadata:
      labels:
        name: node-js-scale-a
        version: "0.2"
        service: node-js-scale-ab
    spec:
      containers:
      - name: node-js-scale
        image: jonbaier/pod-scaling:0.2
        ports:
        - containerPort: 80
        livenessProbe:
          # An HTTP health check
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 5
        readinessProbe:
          # An HTTP health check
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-js-scale-b
  labels:
    name: node-js-scale-b
    version: "0.3"
    service: node-js-scale-ab
spec:
  replicas: 2
  selector:
    name: node-js-scale-b
    version: "0.3"
    service: node-js-scale-ab
  template:
    metadata:
      labels:
        name: node-js-scale-b
        version: "0.3"
        service: node-js-scale-ab
    spec:
      containers:
      - name: node-js-scale
        image: jonbaier/pod-scaling:0.3
        ports:
        - containerPort: 80
        livenessProbe:
          # An HTTP health check
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 5
        readinessProbe:
          # An HTTP health check
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 30
          timeoutSeconds: 1
```

Note that we have the same service label, so these replicas will also be added to the service pool based on this selector. We also have `livenessProbe` and `readinessProbe` defined to make sure that our new version is working as expected. Again, use the `create` command to spin up the controller:

```
$ kubectl create -f pod-A-controller.yaml
$ kubectl create -f pod-B-controller.yaml
```

Now, we have a service balancing both versions of our app. In a true A/B test, we would now want to start collecting metrics on the visits to each version. Again, we have `sessionAffinity` set to `ClientIP`, so all requests will go to the same pod. Some users will see v0.2, and some will see v0.3.

Because we have `sessionAffinity` turned on, your test will likely show the same version every time. This is expected, and you would need to attempt a connection from multiple IP addresses to see both user experiences with each version.

Since the versions are each on their own pod, one can easily separate logging and even add a logging container to the pod definition for a sidecar logging pattern. For brevity, we will not cover that setup in this book, but we will look at some of the logging tools in `Chapter 8`, *Monitoring and Logging*.

We can start to see how this process will be useful for a canary release or a manual blue-green deployment. We can also see how easy it is to launch a new version and slowly transition over to the new release.

Let's look at a basic transition quickly. It's really as simple as a few `scale` commands, which are as follows:

```
$ kubectl scale --replicas=3 rc/node-js-scale-b
$ kubectl scale --replicas=1 rc/node-js-scale-a
$ kubectl scale --replicas=4 rc/node-js-scale-b
$ kubectl scale --replicas=0 rc/node-js-scale-a
```

Use the `get pods` command combined with the `-l` filter in between the `scale` commands to watch the transition as it happens.

Now, we have fully transitioned over to version 0.3 (`node-js-scale-b`). All users will now see version 0.3 of the site. We have four replicas of version 0.3 and none of 0.2. If you run a `get rc` command, you will notice that we still have an `ReplicationControllers` for 0.2 (`node-js-scale-a`). As a final cleanup, we can remove that controller completely, as follows:

```
$ kubectl delete rc/node-js-scale-a
```

Application autoscaling

A recent feature addition to Kubernetes is that of the Horizontal Pod Autoscaler. This resource type is really useful as it gives us a way to automatically set thresholds for scaling our application. Currently, that support is only for CPU, but there is alpha support for custom application metrics as well.

Let's use the `node-js-scale` `ReplicationController` from the beginning of the chapter and add an autoscaling component. Before we start, let's make sure we are scaled back down to one replica using the following command:

```
$ kubectl scale --replicas=1 rc/node-js-scale
```

Now, we can create a Horizontal Pod Autoscaler, `node-js-scale-hpa.yaml` with the following `hpa` definition:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: node-js-scale
spec:
  minReplicas: 1
  maxReplicas: 3
  scaleTargetRef:
    apiVersion: v1
    kind: ReplicationController
    name: node-js-scale
  targetCPUUtilizationPercentage: 20
```

Go ahead and create this with the `kubectl create -f` command. Now, we can list the Horizontal Pod Autoscaler and get a description as well:

```
$ kubectl get hpa
```

We can also create autoscaling in the command line with the `kubectl autoscale` command. The preceding YAML will look like the following:
`$ kubectl autoscale rc/node-js-scale --min=1 --max=3 --cpu-percent=20`

This will show us an autoscaler on `node-js-scale ReplicationController` with a target CPU of 30%. Additionally, you will see that minimum pods is set to 1 and maximum to 3:

```
NAME            REFERENCE                                  TARGET    CURRENT
   MINPODS   MAXPODS   AGE
node-js-scale   ReplicationController/node-js-scale   30%       0%
   1         3         2d
```

Horizontal pod autoscaler with no load

Let's also query our pods to see how many are running right now:

```
$ kubectl get pods -l name=node-js-scale
```

We should see only one `node-js-scale` pod because our Horizontal Pod Autoscaler is showing 0% utilization, so we will need to generate some load. We will use the popular `boom` application common in many container demos. The following listing `boomload.yaml` will help us create continuous load until we can hit the CPU threshold for the autoscaler:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: boomload
spec:
  replicas: 1
  selector:
    app: loadgenerator
  template:
    metadata:
      labels:
        app: loadgenerator
    spec:
      containers:
      - image: williamyeh/boom
        name: boom
        command: ["/bin/sh","-c"]
        args: ["while true ; do boom http://node-js-scale/ -c 10 -n 100
        ; sleep 1 ; done"]
```

Use the `kubectl create -f` command with this listing and then be ready to start

monitoring the `hpa`. We can do this with the `kubectl get hpa` command we used earlier.

It may take a few moments, but we should start to see the current CPU utilization increase. Once it goes above the 20% threshold we set, the autoscaler will kick in:

```
NAME           REFERENCE                                TARGET    CURRENT
   MINPODS   MAXPODS   AGE
node-js-scale   ReplicationController/node-js-scale   30%       49%
   1         3         2d
```

Horizontal pod autoscaler after load starts

Once we see this, we can run `kubectl get pod` again and see there are now several `node-js-scale` pods:

```
$ kubectl get pods -l name=node-js-scale
```

We can clean up now by killing our load generation pod:

```
$ kubectl delete rc/boomload
```

Now, if we watch the `hpa`, we should start to see the CPU usage drop. It may take a few minutes, but eventually we will go back down to 0% CPU load.

Scaling a cluster

All these techniques are great for scaling the application, but what about the cluster itself? At some point, you will pack the nodes full and need more resources to schedule new pods for your workloads.

Autoscaling

When you create your cluster, you can customize the starting number of nodes (minions) with the `NUM_MINIONS` environment variable. By default, it is set to 4.

Additionally, the Kubernetes team has started to build autoscaling capability into the cluster itself. Currently, this is only supported on GCE and GKE, but work is being done on other providers. This capability utilizes the `KUBE_AUTOSCALER_MIN_NODES`, `KUBE_AUTOSCALER_MAX_NODES`, and `KUBE_ENABLE_CLUSTER_AUTOSCALER` environment variables.

The following example shows how to set the environment variables for autoscaling before running `kube-up.sh`:

```
$ export NUM_MINIONS=5
$ export KUBE_AUTOSCALER_MIN_NODES=2
$ export KUBE_AUTOSCALER_MAX_NODES=5
$ export KUBE_ENABLE_CLUSTER_AUTOSCALER=true
```

Also, bear in mind that changing this after the cluster is started will have no effect. You would need to tear down the cluster and create it once again. Thus, this section will show you how to add nodes to an existing cluster without rebuilding it.

Once you start a cluster with these settings, your cluster will automatically scale up and down with the minimum and maximum limits based on compute resource usage in the cluster.

GKE clusters also support autoscaling when launched, when using the alpha features. The preceding example will use a flag such as `--enable-autoscaling --min-nodes=2 --max-nodes=5` in a command-line launch.

Scaling up the cluster on GCE

If you wish to scale out an existing cluster, we can do it with a few steps. Manually scaling up your cluster on GCE is actually quite easy. The existing plumbing uses managed instance groups in GCE, which allow you to easily add more machines of a standard configuration to the group via an instance template.

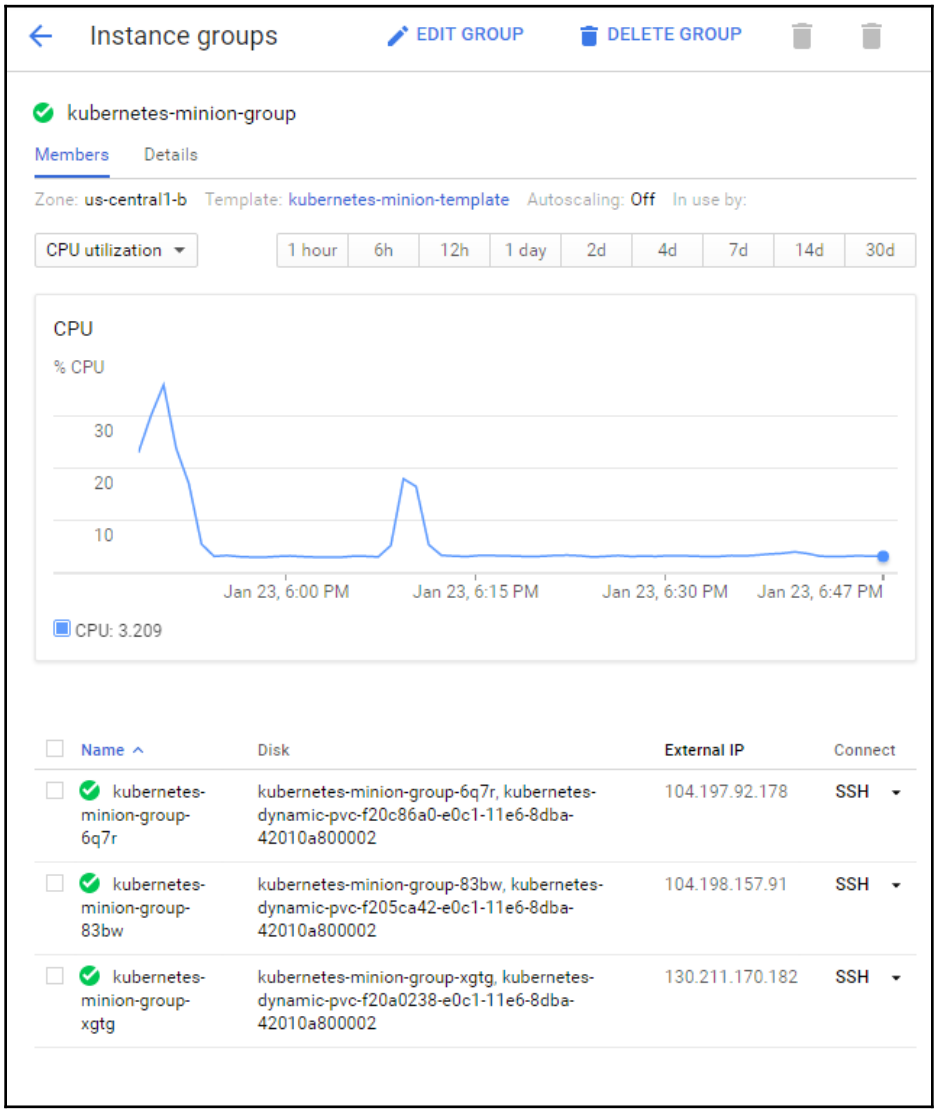
You can see this template easily in the GCE console. First, open the console; by default, this should open your default project console. If you are using another project for your Kubernetes cluster, simply select it from the project drop-down at the top of the page.

On the side panel, look under **Compute** and then **Compute Engine**, and select **Instance templates**. You should see a template titled **kubernetes-minion-template**. Note that the name could vary slightly if you've customized your cluster naming settings. Click on that template to see the details. Refer to the following screenshot:

The GCE Instance template for minions

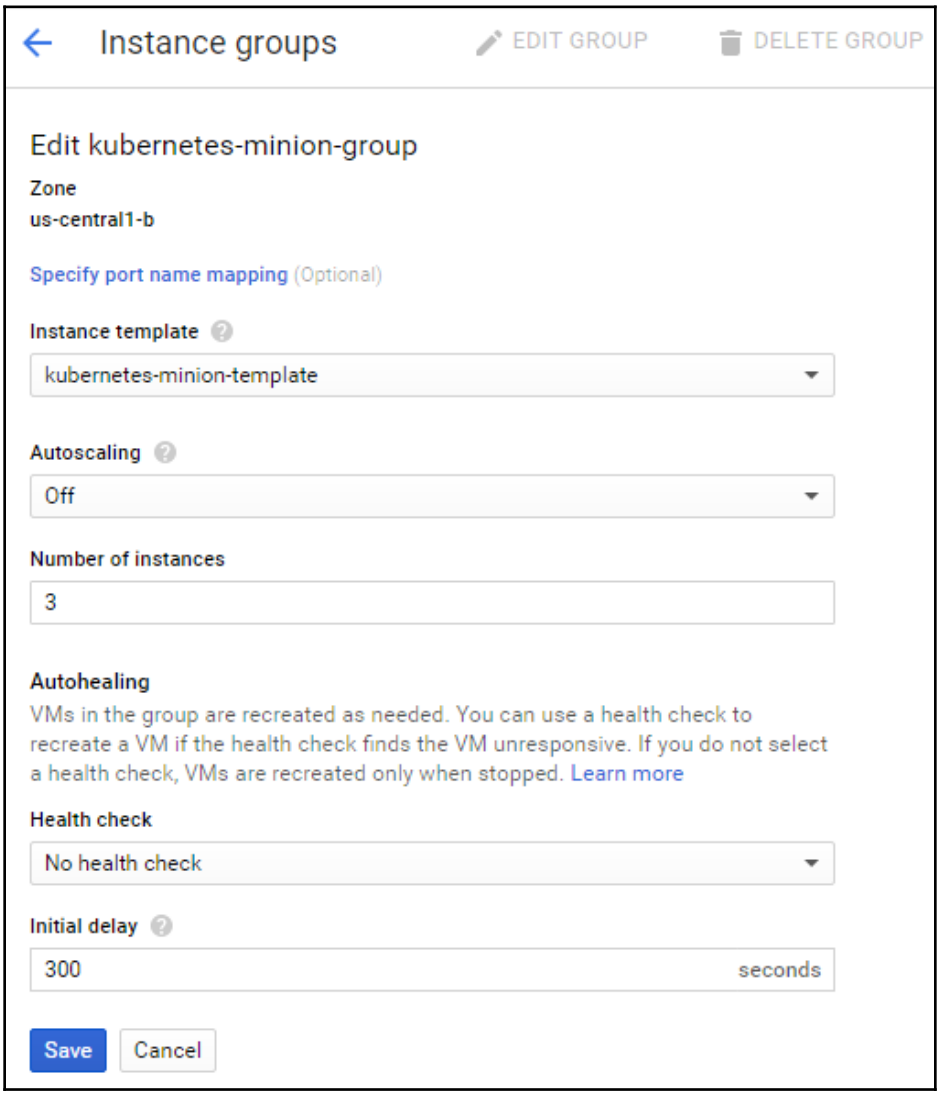
You'll see a number of settings, but the meat of the template is under the **Custom** metadata. Here, you will see a number of environment variables and also a startup script that is run after a new machine instance is created. These are the core components that allow us to create new machines and have them automatically added to the available cluster nodes.

Because the template for new machines is already created, it is very simple to scale out our cluster in GCE. Once in the **Compute** section of the console, simply go to **Instance groups** located right above the **Instance templates** link on the side panel. Again, you should see a group titled **kubernetes-minion-group** or something similar. Click on that group to see the details, as shown in the following screenshot:



The GCE instance group for minions

You'll see a page with a CPU metrics graph and three instances listed here. By default, the cluster creates three nodes. We can modify this group by clicking on the **EDIT GROUP** button at the top of the page:



The GCE instance group edit page

You should see **kubernetes-minion-template** selected in the **Instance template** that we reviewed a moment ago. You'll also see an **Autoscaling** setting, which is **Off** by default, and an instance count of `3`. Simply increment this to `4` and click on **Save**. You'll be taken back to the group details page and you'll see a pop-up dialog showing the pending changes.

You'll also see some auto healing properties on the **Instance groups** edit page. This recreates failed instances and allows you to set health checks, as well as an initial delay period before an action is taken.

In a few minutes, you'll have a new instance listed on the details page. We can test that this is ready using the `get nodes` command from the command line:

```
$ kubectl get nodes
```

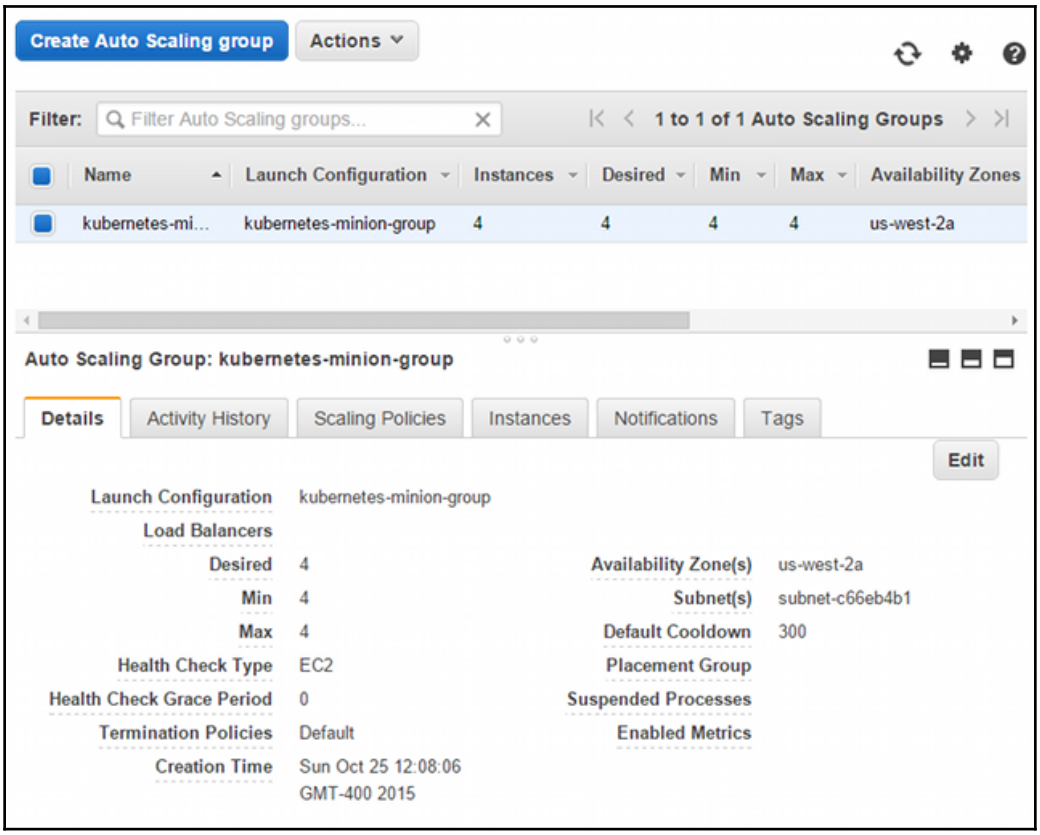
A word of caution on autoscaling and scaling down in general:
First, if we repeat the earlier process and decrease the countdown to four, GCE will remove one node. However, it will not necessarily be the node you just added. The good news is that pods will be rescheduled on the remaining nodes. However, it can only reschedule where resources are available. If you are close to full capacity and shut down a node, there is a good chance that some pods will not have a place to be rescheduled. In addition, this is not a live migration, so any application state will be lost in the transition. The bottom line is that you should carefully consider the implications before scaling down or implementing an autoscaling scheme.

For more information on general autoscaling in GCE, refer to the `https://cloud.google.com/compute/docs/autoscaler/?hl=en_US#scaling_based_on_cpu_utilization` link.

Scaling up the cluster on AWS

The AWS provider code also makes it very easy to scale up your cluster. Similar to GCE, the AWS setup uses autoscaling groups to create the default four minion nodes. In the future, the autoscaling groups will hopefully be integrated into the Kubernetes cluster autoscaling functionality. For now, we will walk though a manual setup.

This can also be easily modified using the CLI or the web console. In the console, from the EC2 page, simply go to the **Auto Scaling Groups** section at the bottom of the menu on the left. You should see a name similar to **kubernetes-minion-group**. Select this group and you will see the details shown in the following screenshot:



Kubernetes minion autoscaling details

We can scale this group up easily by clicking on **Edit**. Then, change the **Desired**, **Min**, and **Max** values to `5` and click on **Save**. In a few minutes, you'll have the fifth node available. You can once again check this using the `get nodes` command.

Scaling down is the same process, but remember that we discussed the same considerations in the previous *Scaling up the cluster on GCE* section. Workloads could get abandoned or, at the very least, unexpectedly restarted.

Scaling manually

For other providers, creating new minions may not be an automated process. Depending on your provider, you'll need to perform various manual steps. It can be helpful to look at the provider-specific scripts in the `cluster` directory.

Managing applications

At the time of this book's writing, new software has emerged that hopes to tackle the problem of managing Kubernetes applications from a holistic perspective. As application installation and continued management grows more complex, software such as Helm hopes to ease the pain for cluster operators creating, versioning, publishing, and exporting application installation and configuration for other operators. You may have also heard the term GitOps, which uses Git as the source of truth from which all Kubernetes instances can be managed.

While we'll jump deeper into **Continuous Integration and Continuous Delivery** (**CI/CD**) in the next chapter, let's see what advantages can be gained by taking advantage of package management within the Kubernetes ecosystem. First, it's important to understand what problem we're trying to solve when it comes to package management within the Kubernetes ecosystem. Helm and programs like it have a lot in common with package managers such as `apt`, `yum`, `rpm`, `dpgk`, Aptitude, and Zypper. These pieces of software helped users cope during the early days of Linux, where programs were simply distributed as source code, with installation documents, configuration files, and the necessary moving pieces left to the operator to set up. These days of course Linux distributions use a great many pre-built packages, which are made available to the user community for consumption on their operating system of choice. In many ways, we're in those early days of software management for Kubernetes, with many different methods for installing software within many different layers of the Kubernetes system. But are there other reasons for wanting a GNU Linux-style package manager for Kubernetes? Perhaps you feel confident that by using containers, or Git and configuration management, you can manage on your own.

Keep in mind the that there several important dimensions to consider when it comes to application management in a Kubernetes cluster:

1. You want to be able to leverage the experience of others. When you install software in your cluster, you want to be able to take advantage of the expertise of the teams that built the software you're running, or experts who've set it up in a way to perform best.

2. You want a repeatable, auditable method of maintaining the application-specific configuration of your cluster across environments. It's difficult to build in environment-specific memory settings, for example, across environments using simpler tools such as cURL, or within a `makefile` or other package compilation tools.

In short, we want to take advantage of the expertise of the ecosystem when deploying technologies such as databases, caching layers, web servers, key/value stores, and other technologies that you're likely to run on your Kubernetes cluster. There are a lot of potential players in this part of the ecosystem, such as Landscaper (`https://github.com/Eneco/landscaper`), Kubepack (`https://github.com/kubepack/pack`), Flux (`https://github.com/weaveworks/flux`), Armada (`https://github.com/att-comdev/armada`), and helmfile (`https://cdp.packtpub.com/getting_started_with_kubernetes__third_edition/wp-admin/post.php?post=29action=pdfpreview`). In this section in particular, we're going to look at Helm (`https://github.com/helm/helm`), which has recently been accepted into the CNCF as an incubating project, and its approach to the problems we've described here.

Getting started with Helm

We'll see how Helm makes it easier to manage Kubernetes applications using charts, which are packages that contain a description of the package in the form of `chart.yml`, and several templates that contain manifests Kubernetes can use to manipulate objects within its systems.

Note: Kubernetes is built with a philosophy of the operator defining a desired end state, with Kubernetes working over time and eventual consistency to enforce that state. Helm's approach to application management follows the same principles. Just as you can manage objects via `kubectl` with imperative commands, imperative objective configuration, and declarative object configuration, Helm takes advantage of the declarative object style, which has the highest functionality curve and highest difficulty.

Let's get started quickly with Helm. First, make sure that you SSH into your Kubernetes cluster that we've been using. You'll notice that as with many Kubernetes pieces, we're going to use Kubernetes to install Helm and its components. You can also use a local installation of Kubernetes from Minikube. First, check and make sure that `kubectl` is set to use the correct cluster:

```
$ kubectl config current-context
kubernetes-admin@kubernetes
Next up, let's grab the helm install script and install it locally. Make
sure to read the script through first so you're comfortable with that it
does!
```

Next up, let's grab the Helm install script and install it locally. Make sure to read the script through first so you're comfortable with what it does!

You can read through the script contents here: `https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get`.

Now, let's run the install script and grab the pieces:

```
master $ curl
https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get >
get_helm.sh
% Total % Received % Xferd Average Speed Time Time Time Current
Dload Upload Total Spent Left Speed
100 6740 100 6740 0 0 22217 0 --:--:-- --:--:-- --:--:-- 22244
master $ chmod 700 get_helm.sh
$ ./get_helm.sh
master $ ./get_helm.sh
Helm v2.9.1 is available. Changing from version v2.8.2.
Downloading
https://kubernetes-helm.storage.googleapis.com/helm-v2.9.1-linux-amd64.tar.
gz
Preparing to install into /usr/local/bin
helm installed into /usr/local/bin/helm
Run 'helm init' to configure helm
```

Now that we've pulled and installed Helm, we can install Tiller on the cluster using `helm init`. You can also run Tiller locally for development, but for production installations and this demo, we'll run Tiller inside the cluster directly as a component itself. Tiller will use the previous context when configuring itself, so make sure that you're using the correct endpoint:

```
master $ helm init
Creating /root/.helm
Creating /root/.helm/repository
Creating /root/.helm/repository/cache
Creating /root/.helm/repository/local
Creating /root/.helm/plugins
Creating /root/.helm/starters
Creating /root/.helm/cache/archive
Creating /root/.helm/repository/repositories.yaml
Adding stable repo with URL:
https://kubernetes-charts.storage.googleapis.com
master $ helm init
Creating /root/.helm
Creating /root/.helm/repository
```

```
Creating /root/.helm/repository/cache
Creating /root/.helm/repository/local
Creating /root/.helm/plugins
Creating /root/.helm/starters
Creating /root/.helm/cache/archive
Creating /root/.helm/repository/repositories.yaml
Adding stable repo with URL:
https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /root/.helm.
Tiller (the Helm server-side component) has been installed into your
Kubernetes Cluster.
Please note: by default, Tiller is deployed with an insecure 'allow
unauthenticated users' policy.
For more information on securing your installation see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
```

Now that we've installed Helm, let's see what it's like to manage applications directly by installing MySQL using one of the official stable charts. We'll make sure we have the latest repositories and then install it:

```
$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Skip local chart repository
...Successfully got an update from the "stable" chart repository
Update Complete.  Happy Helming!
```

You can get a sneak preview of the power of Helm managed MySQL by running the `install` command, `helm install stable/mysql`, which is helm's version of man pages for the application install:

```
$ helm install stable/mysql
NAME:   guilded-otter
LAST DEPLOYED: Mon Jun  4 01:49:46 2018
NAMESPACE: default
STATUS: DEPLOYED
RESOURCES:
==> v1beta1/Deployment
NAME                  DESIRED CURRENT UP-TO-DATE  AVAILABLE AGE
guilded-otter-mysql  1 1 1         0 0s
==> v1/Pod(related)
NAME                                    READY STATUS RESTARTS AGE
guilded-otter-mysql-5dd65c77c6-46hd4  0/1 Pending 0 0s
==> v1/Secret
NAME                  TYPE DATA AGE
guilded-otter-mysql  Opaque 2 0s
==> v1/ConfigMap
```

```
NAME                        DATA AGE
guilded-otter-mysql-test    1 0s
==> v1/PersistentVolumeClaim
NAME                  STATUS VOLUME CAPACITY  ACCESS MODES STORAGECLASS AGE
guilded-otter-mysql   Pending 0s
==> v1/Service
NAME                  TYPE CLUSTER-IP    EXTERNAL-IP PORT(S) AGE
guilded-otter-mysql   ClusterIP 10.105.59.60  <none> 3306/TCP 0s
```

Helm installs a number of pieces here, which we recognize as Kubernetes objects, including Deployment, Secret, and ConfigMap. You can view your installation of MySQL with `helm ls`, and delete your MySQL installation with `helm delete <cluster_name>`. You can also create your own charts with `helm init <chart_name>` and lint those charts with Helm lint.

If you'd like to learn more about the powerful tools available to you with Helm, check out the docs: `https://docs.helm.sh/`. We'll also dive into more comprehensive examples in the next chapter when we look at CI/CD.

Summary

We should now be a bit more comfortable with the basics of application scaling in Kubernetes. We also looked at the built-in functions in order to roll updates as well as a manual process for testing and slowly integrating updates. We took a look at how to scale the nodes of our underlying cluster and increase the overall capacity for our Kubernetes resources. Finally, we explored some of the new autoscaling concepts for both the cluster and our applications themselves.

In the next chapter, we will look at the latest techniques for scaling and updating applications with the new `deployments` resource type, as well as some of the other types of workloads we can run on Kubernetes.

Questions

1. What is the name of the command that allows you to increase the number of replication controllers and the new Deployments abstraction in order to meet application needs?
2. What is the name of the strategy for providing smooth rollouts to applications without interrupting the user experience?
3. What is one type of session affinity available during deployment?
4. What is the recent addition to Kubernetes that allows for pods in the cluster to scale horizontally?
5. Which environment variables, if set, allow the cluster to scale Kubernetes nodes with demand?
6. Which software tool allows you to install applications and leverage the expertise of those product team's installation settings?
7. What is a Helm install file called?

Further reading

If you'd like to read more about Helm, check out its web page here: `https://www.helm.sh/blog/index.html`. If you'd like to read more about the software behind cluster autoscaling, check out the Kubernetes `autoscaler` repository: `https://github.com/kubernetes/autoscaler`.

7
Designing for Continuous Integration and Delivery

This chapter will show the reader how to integrate their build pipeline and deployments with a Kubernetes cluster. It will cover the concept of using gulp.js and Jenkins in conjunction with your Kubernetes cluster. We'll also use Helm and Minikube to show you another demo of how Continuous Integration and Delivery works with newer, more advanced methods.

The following topics will be covered in the chapter:

- Integrating Kubernetes with a Continuous Deployment pipeline
- Using gulp.js with Kubernetes
- Integrating Jenkins with Kubernetes
- Installing and using Helm and Jenkins

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in, or you can use a local Minikube instance of Kubernetes. You can also use the Play with Kubernetes app, designed for use over the web, at `https://labs.play-with-k8s.com/`.

Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter07`.

Integrating Kubernetes with a continuous delivery pipeline

Continuous integration and delivery are key components in modern development shops. **Continuous Integration/Continuous Delivery** (**CI/CD**) often easy to remove them after builds are run. In addition, if you already have a large portion of infrastructure available on your cluster, it can make sense to utilize the idle capacity for builds and testing.

In this article, we will explore two popular tools used in building and deploying software:

- **gulp.js**: This is a simple task runner used to automate the build process using JavaScript and Node.js
- **Jenkins**: This is a fully-fledged continuous integration server

gulp.js

gulp.js gives us the framework to do build as code. Similar to Infrastructure as code, this allows us to programmatically define our build process. We will walk through a short example to demonstrate how you can create a complete workflow, from a Docker image build through to the final Kubernetes service.

Prerequisites

For this section of the article, you will need a Node.js environment installed and ready, including the **node package manager** (**npm**). If you do not already have these packages installed, you can find instructions for installing them at `https://docs.npmjs.com/getting-started/installing-node`.

You can check whether or not Node.js is installed correctly by using the `node -v` command.

You'll also need Docker CE and a Docker Hub account to push a new image. You can find instructions to install Docker CE at `https://docs.docker.com/installation/`. You can easily create a DockerHub account at `https://hub.docker.com/`.

After you have your credentials, you can log in with the CLI using the `$ docker login` command.

gulp.js build example

Let's start by creating a project directory named `node-gulp`:

```
$ mkdir node-gulp
$ cd node-gulp
```

Next, we will install the `gulp` package and then check whether it's ready by running the `npm` command with the version flag, as follows:

```
$ npm install -g gulp
```

You may need to open a new Terminal window to make sure that `gulp` is on your path. Also, make sure to navigate back to your `node-gulp` directory with the following command:

```
 $ gulp -v
```

Next, we will install `gulp` locally in our project folder, along with the `gulp-git` and `gulp-shell` plugins, as follows:

```
$ npm install --save-dev gulp
$ npm install gulp-git -save
$ npm install --save-dev gulp-shell
```

Finally, we need to create a Kubernetes controller and service definition file, as well as a `gulpfile.js` file, to run all of our tasks. Again, these files are available in the book file bundle, should you wish to copy them straight over instead. Refer to the following `node-gulp-controller.yaml` file:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: node-gulp
  labels:
    name: node-gulp
spec:
  replicas: 1
  selector:
    name: node-gulp
  template:
    metadata:
```

```
      labels:
        name: node-gulp
    spec:
      containers:
      - name: node-gulp
        image: <your username>/node-gulp:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
```

As you can see in the preceding code, we have a basic controller. You will need to replace `<your username>/node-gulp:latest` with your Docker Hub username. Save the following code as `node-gulp-service.yaml` file:

```
apiVersion: v1
kind: Service
metadata:
  name: node-gulp
  labels:
    name: node-gulp
spec:
  type: LoadBalancer
  ports:
  - name: http
    protocol: TCP
    port: 80
  selector:
    name: node-gulp
```

Next, we have a simple service that selects the pods from our controller and creates an external load balancer for access, as earlier:

```
var gulp = require('gulp');
var git = require('gulp-git');
var shell = require('gulp-shell');

// Clone a remote repo
gulp.task('clone', function(){
  return
git.clone('https://github.com/jonbaierCTP/getting-started-with-kubernetes-s
e.git', function (err) {
    if (err) throw err;
  });

});

// Update codebase
gulp.task('pull', function(){
```

```
  return git.pull('origin', 'master', {cwd: './getting-started-with-
kubernetes-se'}, function (err) {
    if (err) throw err;
  });
});

//Build Docker image
gulp.task('docker-build', shell.task([
  'docker build -t <your username>/node-gulp ./getting-started-with-
kubernetes-se/docker-image-source/container-info/',
  'docker push <your username>/node-gulp'
]));

//Run new pod
gulp.task('create-kube-pod', shell.task([
  'kubectl create -f node-gulp-controller.yaml',
  'kubectl create -f node-gulp-service.yaml'
]));

//Update pod
gulp.task('update-kube-pod', shell.task([
  'kubectl delete -f node-gulp-controller.yaml',
  'kubectl create -f node-gulp-controller.yaml'
]));
```

Finally, we have the preceding `gulpfile.js` file. This is where all of our build tasks are defined. Again, fill in your own Docker Hub username in both of the `<your username>/node-gulp` sections.

Looking through the file, first, we can see that the clone task downloads our image source code from GitHub. The pull tasks execute a `git` pull on the cloned repository. Next, the `docker-build` command builds an image from the `container-info` sub folder and pushes it to Docker Hub. Finally, we have the `create-kube-pod` and `update-kube-pod` commands. As you can probably guess, the `create-kube-pod` command creates our controller and service for the first time, whereas the `update-kube-pod` command simply replaces the controller.

Let's go ahead and run these commands and see our end-to-end workflow:

```
$ gulp clone
$ gulp docker-build
```

The first time through, you can also run the `create-kube-pod` command, as follows:

```
$ gulp create-kube-pod
```

This is all there is to it. If we run a quick `kubectl` describe command for the `node-gulp` service, we can get the external IP for our new service. Browse to that IP and you'll see the familiar `container-info` application running. Note that the host starts with `node-gulp`, just as we named it in the previously mentioned pod definition:

On subsequent updates, run the `pull` and `update-kube-pod` commands, as shown here:

```
$ gulp pull
$ gulp docker-build
$ gulp update-kube-pod
```

This is a very simple example, but you can begin to see how easy it is to coordinate your build and deployment end to end with a few simple lines of code. Next, we will look at how to use Kubernetes to actually run builds using Jenkins.

The Kubernetes plugin for Jenkins

One way we can use Kubernetes for our CI/CD pipeline is to run our Jenkins build slaves in a containerized environment. Luckily, there is already a plugin, written by Carlos Sanchez, that allows you to run Jenkins slaves in Kubernetes' pods.

Prerequisites

You'll need a Jenkins server handy for this next example. If you don't have one you can use, there is a Docker image available at `https://hub.docker.com/_/jenkins/`.

Running it from the Docker CLI is as simple as the following command:

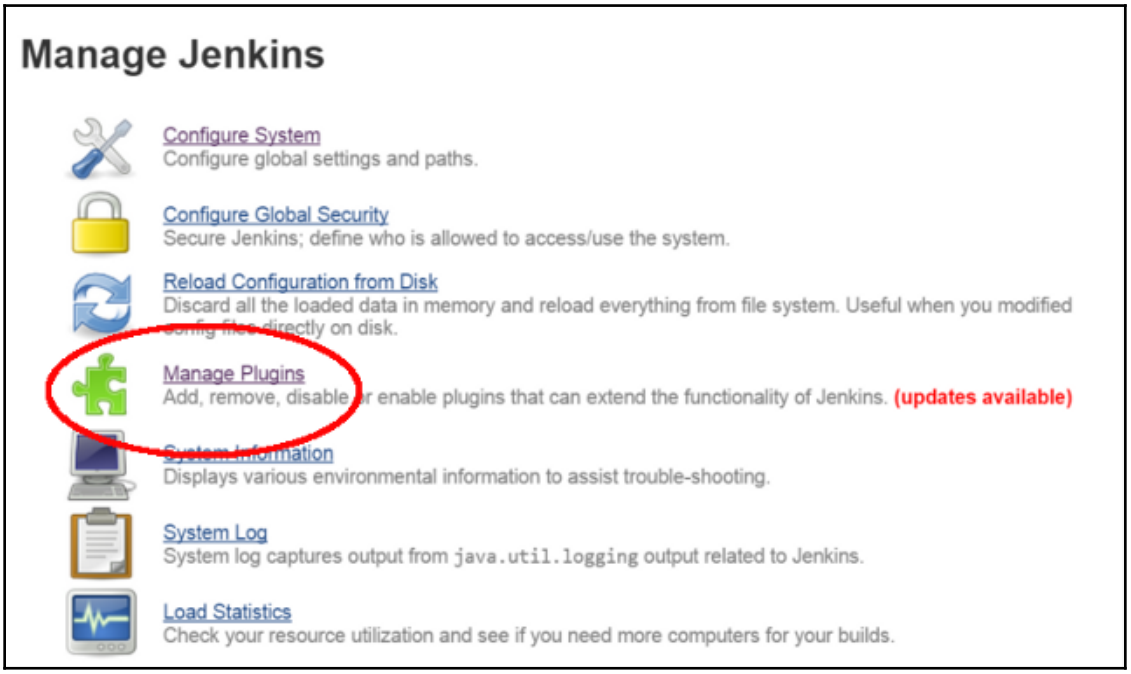
```
docker run --name myjenkins -p 8080:8080 -v /var/jenkins_home jenkins
```

Installing plugins

Log in to your Jenkins server, and from your home dashboard, click on **Manage Jenkins**.

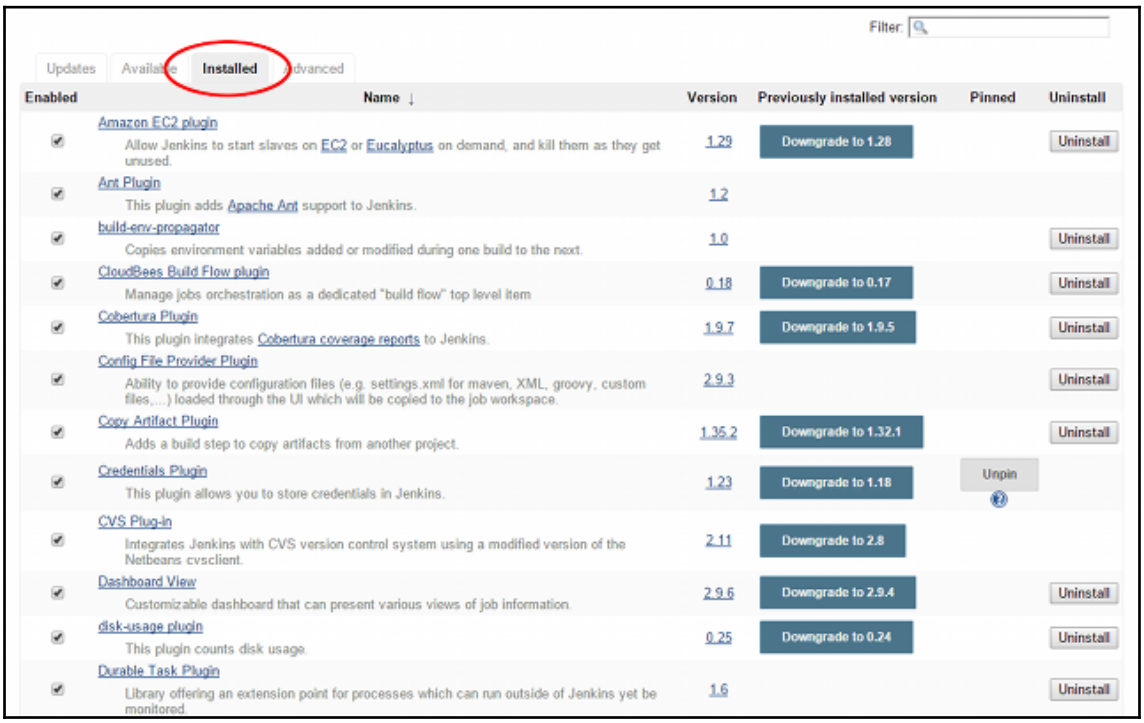
A note for those installing a new Jenkins server: when you first log in to the Jenkins server, it asks you to install plugins. Choose the default ones, or no plugins will be installed!

Then, on the **Manage Jenkins** page, select **Manage Plugins** from the list, as follows:



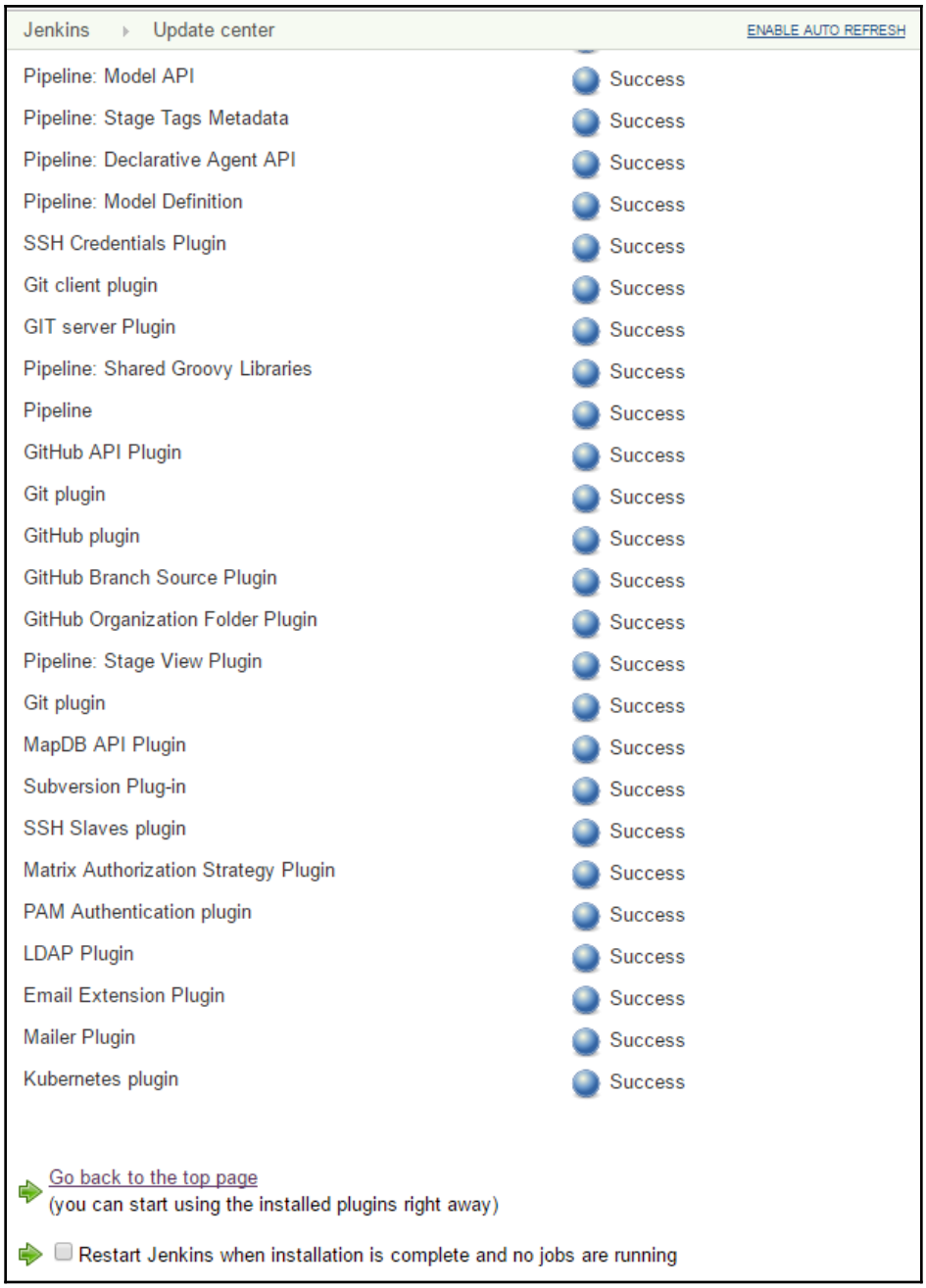
The main dashboard in Jenkins

The credentials plugin is required, but should be installed by default. We can check the **Installed** tab if in doubt, as shown in the following screenshot:



Installed plugins in Jenkins

Next, let's click on the **Available** tab. The Kubernetes plugin should be located under **Cluster Management and Distributed Build** or **Misc (cloud)**. There are many plugins, so you can alternatively search for `Kubernetes` on the page. Check the box for **Kubernetes plugin** and click on **Install without restart**. This will install the **Kubernetes plugin** and the **Durable Task Plugin**:



The plugin installation screen in Jenkins

If you wish to install a nonstandard version, or just like to tinker, you can optionally download the plugins. The latest Kubernetes and durable task plugins can be found here:

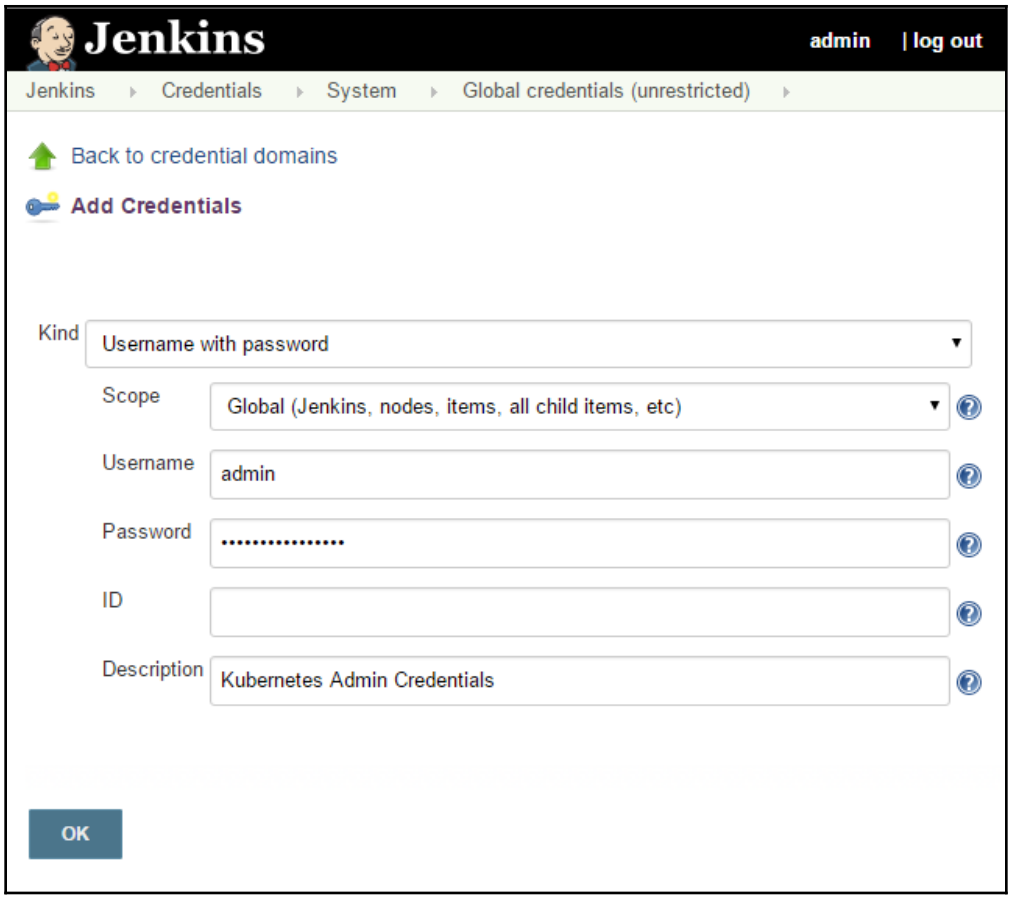
- **Kubernetes plugin**:
  `https://wiki.jenkins-ci.org/display/JENKINS/Kubernetes+Plugin`
- **Durable task plugin**:
  `https://wiki.jenkins-ci.org/display/JENKINS/Durable+Task+Plugin`

Next, we can click on the **Advanced** tab and scroll down to **Upload plugin**. Navigate to the `durable-task.hpi` file and click on **Upload**. You should see a screen that shows an installation progress bar. After a minute or two, it will update to **Success**.

Finally, install the main Kubernetes plugin. On the left-hand side, click on **Manage Plugins** and then the **Advanced** tab once again. This time, upload the `kubernetes.hpi` file and click on **Upload**. After a few minutes, the installation should be complete.

Configuring the Kubernetes plugin

Click on **Back to Dashboard**, or the **Jenkins** link in the top-left corner. Back on the main dashboard page, click on the **Credentials** link. Choose a domain from the list; in my case, I just used the default global credentials domain. Click on **Add Credentials**, and you'll be presented with the following screen:



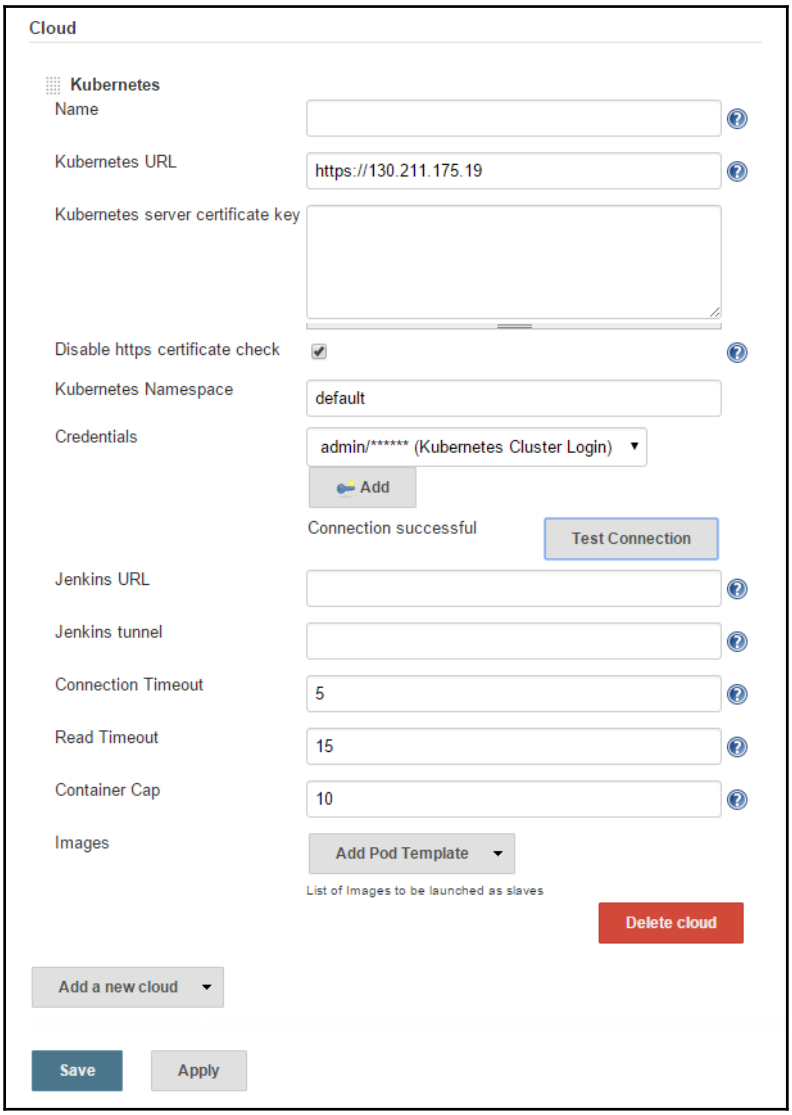
The Add Credentials screen

Leave **Kind** as **Username with password** and **Scope** as **Global (Jenkins, nodes, items, all child items, etc)**. Add your Kubernetes admin credentials. Remember that you can find these by running the following `config` command:

```
$ kubectl config view
```

You can leave **ID** blank, fill in **Description** with something sensible, and then click on the **OK** button.

Now that we have our credentials saved, we can add our Kubernetes server. Click on the **Jenkins** link in the top-left corner, and then **Manage Jenkins**. From there, select **Configure System** and scroll all the way down to the **Cloud** section. Select **Kubernetes** from the **Add a new cloud** drop-down menu and a **Kubernetes** section will appear, as follows:



The new Kubernetes cloud settings page in Jenkins

You'll need to specify the URL for your master in the form of `https://<Master IP>/`.

Next, choose the credentials we added from the drop-down menu. Since Kubernetes uses a self-signed certificate by default, you'll also need to check the **Disable https certificate check** checkbox.

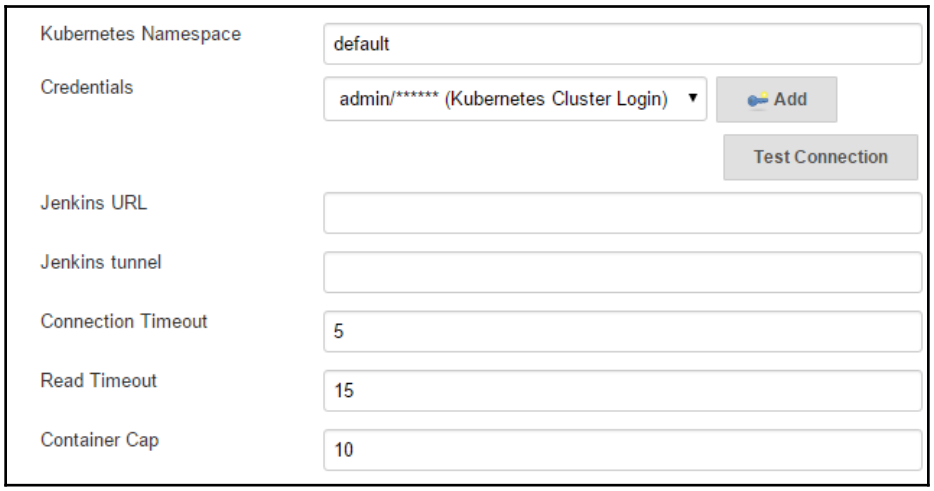
Click on **Test Connection**, and if all goes well, you should see **Connection successful** appear next to the button.

If you are using an older version of the plugin, you may not see the **Disable https certificate check** checkbox. If this is the case, you will need to install the self-signed certificate directly on the Jenkins master.

Finally, we will add a pod template by choosing **Kubernetes Pod Template** from the **Add Pod Template** drop-down menu next to **Images**.

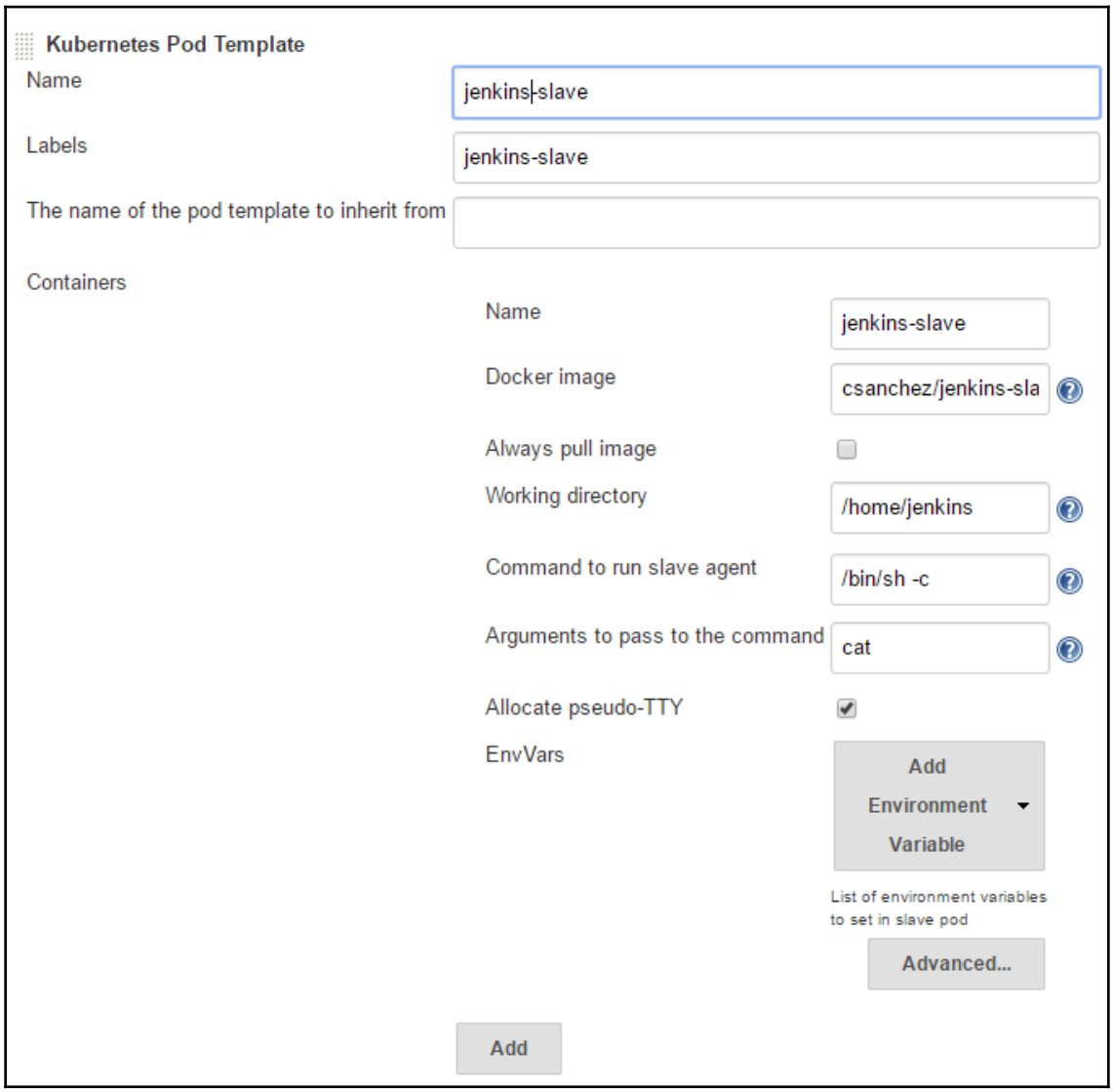
This will create another new section. Use `jenkins-slave` for the **Name** and **Labels** section. Click on **Add** next to **Containers** and again use `jenkins-slave` for the **Name.** Use `csanchez/jenkins-slave` for the **Docker Image** and leave `/home/jenkins` for the **Working Directory**.

Labels can be used later on in the build settings to force the build to use the Kubernetes cluster:



Kubernetes cluster addition

Here is the pod template that expands the cluster addition, as shown in the following screenshot:



The Kubernetes pod template

Click on **Save** and you are all set. Now, new builds created in Jenkins can use the slaves in the Kubernetes pod we just created.

Here is another note about firewalls. The Jenkins master will need to be reachable by all the machines in your Kubernetes cluster, as the pod could land anywhere. You can find out your port settings in Jenkins under **Manage Jenkins** | **Configure Global Security**.

Helm and Minikube

Let's try setting up some CI/CD with other tools, so we can experiment with the newest offerings in the Kubernetes ecosystem. First, let's explore how easy it is to install Jenkins with Helm.

First, open the Minikube dashboard so you can see what happens when we install various things. Do this with the following command:

```
$ minikube dashboard
```

Let's create a namespace for the Jenkins environment, as follows:

```
$ kubectl get namespaces
NAME          STATUS AGE
default       Active 3d
kube-public   Active 3d
kube-system   Active 3d
```

Now, let's create a template:

```
$ mkdir -p ~/gsw-k8s-helm && cd ~/gsw-k8s-helm
$ cat <<K8s >> namespace-jenkins.yaml
apiVersion: v1
kind: Namespace
metadata:
 name: gsw-k8s-jenkins
K8s
```

Now, you can create the namespace as follows:

```
kubectl create -f namespace-jenkins.yaml
namespace "gsw-k8s-jenkins" created
```

There are two ways to verify that it was actually created. First, you can take a look at the dashboard with the `minikube dashboard` command:

Secondly, you can look at the CLI with `kubectl get namespaces`:

```
$ helm-jenkins jesse$ kubectl get namespaces
NAME              STATUS AGE
default           Active 5d
gsw-k8s-jenkins   Active 1d
kube-public       Active 5d
kube-system       Active 5d
```

Let's create a persistent volume for Jenkins to take advantage of. This will allow us to persist data in the cluster when Minikube reboots. In a production environment, you'd need to use some type of block or driver for your storage. Let's create a `jenkins-volume.yaml` file called `jenkins-persist`.

Here's what you'll put into that file:

```
apiVersion: v1
kind: PersistentVolume
metadata:
 name: jenkins-persist
 namespace: jenkins-project
spec:
 storageClassName: jenkins-persist
 accessModes:
   - ReadWriteOnce
 capacity:
   storage: 20Gi
 persistentVolumeReclaimPolicy: Retain
 hostPath:
   path: /storage/jenkins-volume/
```

Now, let's create the volume for Jenkins to use:

```
$ kubectl create -f jenkins-volume.yaml
persistentvolume "jenkins-persist" created
```

Great! Now, we're ready to use Helm to install Jenkins nice and easily. Let's use the following values file with our installation:

```
# Default values for jenkins.
# This is a YAML-formatted file.
# Declare name/value pairs to be passed into your templates.
# name: value
## Overrides for generated resource names
# See templates/_helpers.tpl
# nameOverride:
# fullnameOverride:
Master:
 Name: jenkins-master
 Image: "jenkins/jenkins"
 ImageTag: "2.127"
 ImagePullPolicy: "Always"
 Component: "jenkins-master"
 UseSecurity: true
 AdminUser: admin
 # AdminPassword: <defaults to random>
 Cpu: "200m"
 Memory: "256Mi"
 ServicePort: 8080
 # For minikube, set this to NodePort, elsewhere use LoadBalancer # <to set
explicitly, choose port between 30000-32767>
ServiceType: NodePort
 NodePort: 32000
 ServiceAnnotations: {}
 ContainerPort: 8080
 # Enable Kubernetes Liveness and Readiness Probes
 HealthProbes: true
 HealthProbesTimeout: 60
 SlaveListenerPort: 50000
 LoadBalancerSourceRanges:
 - 0.0.0.0/0
 # List of plugins to be install during Jenkins master start
 InstallPlugins:
 - kubernetes:1.7.1
 - workflow-aggregator:2.5
 - workflow-job:2.21
 - credentials-binding:1.16
 - git:3.9.1
 - greenballs:1.15
```

```
 # Used to approve a list of groovy functions in pipelines used
the script-security plugin. Can be viewed under /scriptApproval
 ScriptApproval:
 - "method groovy.json.JsonSlurperClassic parseText
java.lang.String"
 - "new groovy.json.JsonSlurperClassic"
 - "staticMethod
org.codehaus.groovy.runtime.DefaultGroovyMethods leftShift
java.util.Map java.util.Map"
 - "staticMethod
org.codehaus.groovy.runtime.DefaultGroovyMethods split
java.lang.String"
 CustomConfigMap: false
 NodeSelector: {}
 Tolerations: {}
Agent:
 Enabled: true
 Image: jenkins/jnlp-slave
 ImageTag: 3.10-1
 Component: "jenkins-slave"
 Privileged: false
 Cpu: "200m"
 Memory: "256Mi"
 # You may want to change this to true while testing a new image
 AlwaysPullImage: false
 # You can define the volumes that you want to mount for this
container
 # Allowed types are: ConfigMap, EmptyDir, HostPath, Nfs, Pod,
Secret
 volumes:
 - type: HostPath
 hostPath: /var/run/docker.sock
 mountPath: /var/run/docker.sock
 NodeSelector: {}
Persistence:
 Enabled: true
 ## A manually managed Persistent Volume and Claim
 ## Requires Persistence.Enabled: true
 ## If defined, PVC must be created manually before volume will
be bound
 # ExistingClaim:
 ## jenkins data Persistent Volume Storage Class
 StorageClass: jenkins-pv
 Annotations: {}
 AccessMode: ReadWriteOnce
 Size: 20Gi
 volumes:
 # - name: nothing
```

```
 # emptyDir: {}
 mounts:
 # - mountPath: /var/nothing
 # name: nothing
 # readOnly: true
NetworkPolicy:
 # Enable creation of NetworkPolicy resources.
 Enabled: false
 # For Kubernetes v1.4, v1.5 and v1.6, use 'extensions/v1beta1'
 # For Kubernetes v1.7, use 'networking.k8s.io/v1'
 ApiVersion: networking.k8s.io/v1
## Install Default RBAC roles and bindings
rbac:
 install: true
 serviceAccountName: default
 # RBAC api version (currently either v1beta1 or v1alpha1)
 apiVersion: v1beta1
 # Cluster role reference
 roleRef: cluster-admin
```

Now that we've set the values file, let's use it to deploy Jenkins:

```
helm install --name gsw-k8s-jenkins -f jenkins-vaules.yaml stable/jenkins -
-namespace gsw-k8s-jenkins
NAME:   gsw-k8s-jenkins
LAST DEPLOYED: Mon Jun 18 22:44:34 2018
NAMESPACE: gsw-k8s-jenkins
STATUS: DEPLOYED
RESOURCES:
...
```

We can get the randomly generated Jenkins secret by addressing the Kubernetes secret store API:

```
$ kubectl get secret --namespace gsw-k8s-jenkins gsw-k8s-jenkins -o
jsonpath="{.data.jenkins-admin-password}" | base64 --decode; echo
<YOUR_PASSWORD_HERE>
```

Verify that Jenkins has installed using the following commands:

```
$ helm ls
NAME REVISION UPDATED STATUS CHART NAMESPACE
gsw-k8s-jenkins 1 Mon Jun 18 22:44:34 2018 DEPLOYED jenkins-0.16.3 gsw-k8s-
jenkins
```

Then, open up Jenkins' home page. You should be able to visit the home page at `http://192.168.99.100:3200`.

Bonus fun

fabric8 bills itself as an integration platform. It includes a variety of logging, monitoring, and continuous delivery tools. It also has a nice console, an API registry, and a 3D game that lets you shoot at your pods. It's a very cool project, and it actually runs on Kubernetes. The website for the project can be found at `http://fabric8.io/`.

fabric8 can be set up easily on your Kubernetes cluster with just a single command, so refer to `http://fabric8.io/guide/getStarted/gke.html/` for more information.

Summary

We looked at two continuous integration tools that can be used with Kubernetes. We did a brief walk-through, examining how to deploy the gulp.js task on our cluster. We also looked at a new plugin used to integrate Jenkins build slaves into your Kubernetes cluster. You should now have a better sense of how Kubernetes can integrate with your own CI/CD pipeline.

Questions

1. What type of software does gulp.js enable us to build?
2. What is the name of the popular CI/CD system that we installed?
3. What is an alternative method of installation for Jenkins?
4. What type of volume is required to run Jenkins on Kubernetes?
5. What is the other requirement for running Jenkins on Kubernetes?
6. What kind of controller is used when deploying with gulp.js?
7. What tool did we use to install gulp.js?

Further reading

If you'd like some additional information on the Node.js and gulp.js ecosystems, check out these titles:

- `https://www.packtpub.com/web-development/mastering-nodejs`
- `https://www.packtpub.com/web-development/learning-nodejs-development`

If you'd like some additional guidance on how to use Jenkins, read the following:

- `https://www.packtpub.com/networking-and-servers/learning-continuous-integration-jenkins`
- `https://www.packtpub.com/application-development/mastering-jenkins`
- `https://www.packtpub.com/virtualization-and-cloud/hands-continuous-integration-and-automation-jenkins-video`

8
Monitoring and Logging

This chapter will cover the use and customization of both built-in and third-party monitoring tools on our Kubernetes cluster. We will cover how to use the tools to monitor the health and performance of our cluster. In addition, we will look at built-in logging, the Google Cloud Logging service, and Sysdig.

The following topics will be covered in this chapter:

- How Kuberentes uses cAdvisor, Heapster, InfluxDB, and Grafana
- Customizing the default Grafana dashboard
- Using Fluentd and Grafana
- Installing and using logging tools
- Working with popular third-party tools, such as Stackdriver and Sysdig, to extend our monitoring capabilities

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in to it, or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes over the web: `https://labs.play-with-k8s.com/`.

Monitoring operations

Real-world monitoring goes far beyond checking whether a system is up and running. Although health checks like those you learned in `Chapter 2`, *Building a Foundation with Core Kubernetes Constructs*, in the *Health checks* section can help us isolate problem applications, operations teams can best serve the business when they can anticipate the issues and mitigate them before a system goes offline.

The best practices in monitoring are to measure the performance and usage of core resources and watch for trends that stray from the normal baseline. Containers are not different here, and a key component to managing our Kubernetes cluster is having a clear view of the performance and availability of the OS, network, system (CPU and memory), and storage resources across all nodes.

In this chapter, we will examine several options to monitor and measure the performance and availability of all our cluster resources. In addition, we will look at a few options for alerting and notifications when irregular trends start to emerge.

Built-in monitoring

If you recall from `Chapter 1`, *Introduction to Kubernetes*, we noted that our nodes were already running a number of monitoring services. We can see these once again by running the `get pods` command with the `kube-system` namespace specified as follows:

```
$ kubectl get pods --namespace=kube-system
```

The following screenshot is the result of the preceding command:

```
NAME                                                READY     STATUS    RESTARTS   AGE
etcd-empty-dir-cleanup-kubernetes-master            1/1       Running   2          2d
etcd-server-events-kubernetes-master                1/1       Running   2          2d
etcd-server-kubernetes-master                       1/1       Running   2          2d
fluentd-cloud-logging-kubernetes-master             1/1       Running   2          2d
fluentd-cloud-logging-kubernetes-minion-group-rh7t  1/1       Running   0          3m
fluentd-cloud-logging-kubernetes-minion-group-s345  1/1       Running   0          3m
fluentd-cloud-logging-kubernetes-minion-group-tp2h  1/1       Running   0          3m
heapster-v1.2.0-2805816975-80mjc                    4/4       Running   0          20h
kube-addon-manager-kubernetes-master                1/1       Running   2          2d
kube-apiserver-kubernetes-master                    1/1       Running   4          2d
kube-controller-manager-kubernetes-master           1/1       Running   2          2d
kube-dns-4101612645-bwsd4                           4/4       Running   0          20h
kube-dns-autoscaler-2715466192-gt3r7                1/1       Running   0          20h
kube-proxy-kubernetes-minion-group-rh7t             1/1       Running   0          4m
kube-proxy-kubernetes-minion-group-s345             1/1       Running   0          4m
kube-proxy-kubernetes-minion-group-tp2h             1/1       Running   0          3m
kube-scheduler-kubernetes-master                    1/1       Running   2          2d
kubernetes-dashboard-3543765157-65g1m               1/1       Running   0          20h
l7-default-backend-2234341178-g4wct                 1/1       Running   0          20h
l7-lb-controller-v0.8.0-kubernetes-master           1/1       Running   2          2d
monitoring-influxdb-grafana-v4-7x0n0                2/2       Running   0          20h
node-problem-detector-v0.1-1zfml                    1/1       Running   0          4m
node-problem-detector-v0.1-cjrtz                    1/1       Running   0          4m
node-problem-detector-v0.1-f87pp                    1/1       Running   2          2d
node-problem-detector-v0.1-vj001                    1/1       Running   0          4m
rescheduler-v0.2.1-kubernetes-master                1/1       Running   2          2d
```

System pod listing

Again, we see a variety of services, but how does this all fit together? If you recall, the node (formerly minions) section from `Chapter 2`, *Building a Foundation with Core Kubernetes Constructs*, each node is running a `kubelet`. The `kubelet` is the main interface for nodes to interact with and update the API server. One such update is the metrics of the node resources. The actual reporting of the resource usage is performed by a program named cAdvisor.

The cAdvisor program is another open source project from Google, which provides various metrics on container resource use. Metrics include CPU, memory, and network statistics. There is no need to tell cAdvisor about individual containers; it collects the metrics for all containers on a node and reports this back to the `kubelet`, which in turn reports to Heapster.

Google's open source projects: Google has a variety of open source projects related to Kubernetes. Check them out, use them, and even contribute your own code!

Both cAdvisor and Heapster are mentioned in the following sections of GitHub:

- **cAdvisor**: `https://github.com/google/cadvisor`
- **Heapster**: `https://github.com/kubernetes/heapster`

Contrib is a catch-all term for a variety of components that are not part of core Kubernetes. It can be found at `https://github.com/kubernetes/contrib`. LevelDB is a key store library that was used in the creation of InfluxDB. It can be found at `https://github.com/google/leveldb`.

Heapster is yet another open source project from Google; you may start to see a theme emerging here (see the preceding information box). Heapster runs in a container on one of the minion nodes and aggregates the data from a `kubelet`. A simple REST interface is provided to query the data.

When using the GCE setup, a few additional packages are set up for us, which saves us time and gives us a complete package to monitor our container workloads. As we can see from the preceding *System pod listing* screenshot, there is another pod with `influx-grafana` in the title.

InfluxDB is described on its official website as follows:

> *An open-source distributed time series database with no external dependencies.*

InfluxDB is based on a key store package (refer to the previous *Google's open source projects* information box) and is perfect to store and query event- or time-based statistics such as those provided by Heapster.

Finally, we have Grafana, which provides a dashboard and graphing interface for the data stored in InfluxDB. Using Grafana, users can create a custom monitoring dashboard and get immediate visibility into the health of their Kubernetes cluster, and therefore their entire container infrastructure.

Exploring Heapster

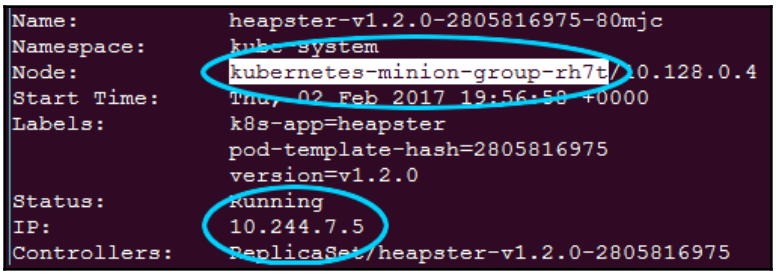
Let's quickly look at the REST interface by running SSH to the node that is running the Heapster pod. First, we can list the pods to find the one that is running Heapster, as follows:

```
$ kubectl get pods --namespace=kube-system
```

The name of the pod should start with `monitoring-heapster`. Run a `describe` command to see which node it is running on, as follows:

```
$ kubectl describe pods/<Heapster monitoring Pod> --namespace=kube-system
```

From the output in the following screenshot, we can see that the pod is running in `kubernetes-minion-merd`. Also note the IP for the pod, a few lines down, as we will need that in a moment:



Heapster pod details

Next, we can SSH to this box with the familiar `gcloud ssh` command, as follows:

```
$ gcloud compute --project "<Your project ID>" ssh --zone "<your gce zone>"
"<kubernetes minion from describe>"
```

From here, we can access the Heapster REST API directly using the pod's IP address. Remember that pod IPs are routable not only in the containers but also on the nodes themselves. The `Heapster` API is listening on port `8082`, and we can get a full list of metrics at `/api/v1/metric-export-schema/`.

Let's look at the list now by issuing a `curl` command to the pod IP address we saved from the `describe` command, as follows:

```
$ curl -G <Heapster IP from describe>:8082/api/v1/metric-export-schema/
```

We will see a listing that is quite long. The first section shows all the metrics available. The last two sections list fields by which we can filter and group. For your convenience, I've added the following tables which are a little bit easier to read:

Metric	Description	Unit	Type
`uptime`	The number of milliseconds since the container was started	ms	Cumulative
`cpu/usage`	The cumulative CPU usage on all cores	ns	Cumulative
`cpu/limit`	The CPU limit in millicores	-	Gauge
`memory/usage`	Total memory usage	Bytes	Gauge
`memory/working_set`	Total working set usage; the working set is the memory that is being used, and is not easily dropped by the kernel	Bytes	Gauge
`memory/limit`	The memory limit	Bytes	Gauge
`memory/page_faults`	The number of page faults	-	Cumulative
`memory/major_page_faults`	The number of major page faults	-	Cumulative
`network/rx`	The cumulative number of bytes received over the network	Bytes	Cumulative
`network/rx_errors`	The cumulative number of errors while receiving over the network	-	Cumulative
`network/tx`	The cumulative number of bytes sent over the network	Bytes	Cumulative
`network/tx_errors`	The cumulative number of errors while sending over the network	-	Cumulative
`filesystem/usage`	The total number of bytes consumed on a filesystem	Bytes	Gauge
`filesystem/limit`	The total size of filesystem in bytes	Bytes	Gauge
`filesystem/available`	The number of available bytes remaining in a the filesystem	Bytes	Gauge

Table 6.1. Available Heapster metrics

Field	Description	Label type
`nodename`	The node name where the container ran	Common
`hostname`	The host name where the container ran	Common

`host_id`	An identifier specific to a host, which is set by the cloud provider or user	Common
`container_base_image`	The user-defined image name that is run inside the container	Common
`container_name`	The user-provided name of the container or full container name for system containers	Common
`pod_name`	The name of the pod	Pod
`pod_id`	The unique ID of the pod	Pod
`pod_namespace`	The namespace of the pod	Pod
`namespace_id`	The unique ID of the namespace of the pod	Pod
`labels`	A comma-separated list of user-provided labels	Pod

Table 6.2. Available Heapster fields

Customizing our dashboards

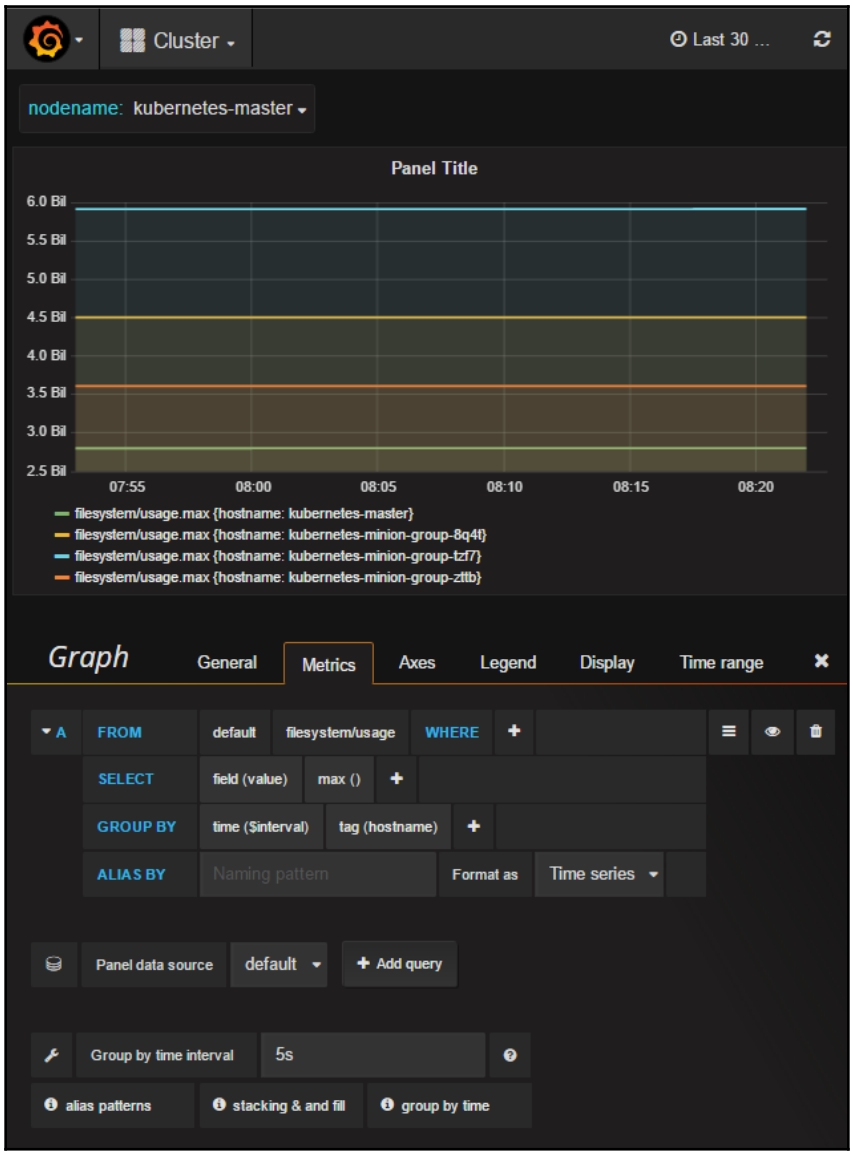
Now that we have the fields, we can have some fun. Recall the Grafana page that we looked at in `Chapter 1`, *Introduction to Kubernetes*. Let's pull that up again by going to our cluster's monitoring URL. Note that you may need to log in with your cluster credentials. Refer to the following format of the link you need to use: `https://<your master IP>/api/v1/proxy/namespaces/kube-system/services/monitoring-grafana`

We'll see the default **Home** dashboard. Click on the down arrow next to **Home** and select **Cluster**. This shows the Kubernetes cluster dashboard, and now we can add our own statistics to the board. Scroll all the way to the bottom and click on **Add a Row**. This should create a space for a new row and present a green tab on the left-hand side of the screen.

Let's start by adding a view into the filesystem usage for each node (minion). Click on the green tab to expand, and then select **Add Panel** and then **Graph**. An empty graph should appear on the screen, along with a query panel for our custom graph.

The first field in this panel should show a query that starts with **SELECT mean("value") FROM.** Click on the **A** character next to this field to expand it. Leave the first field next to **FROM** as **default** and then click on the next field with the **select measurement** value. A drop-down menu will appear with the Heapster metrics we saw in the previous tables. Select `filesystem/usage_bytes_gauge`. Now, in the **SELECT** row, click on **mean()** and then on the **x** symbol to remove it. Next, click on the + symbol on the end of the row and add **selectors** and **max**. Then, you'll see a **GROUP BY** row with **time($interval)** and **fill(none)**. Carefully click on **fill** and not on the **(none)** portion, and again on **x** to remove it.

Then, click on the **+** symbol at the end of the row and select **tag(hostname)**.Finally, at the bottom of the screen we should see a **Group by time interval.** Enter `5s` there and you should have something similar to the following screenshot:



Heapster pod details

Next, let's click on the **Axes** tab, so that we can set the units and legend. Under **Left Y Axis**, click on the field next to **Unit** and set it to **data | bytes** and **Label** to **Disk Space Used**. Under **Right Y Axis**, set **Unit** to **none | none**. Next, on the **Legend** tab, make sure to check **Show** in **Options** and **Max** in **Values**.

Now, let's quickly go to the **General** tab and choose a title. In my case, I named mine `Filesystem Disk Usage by Node (max).`

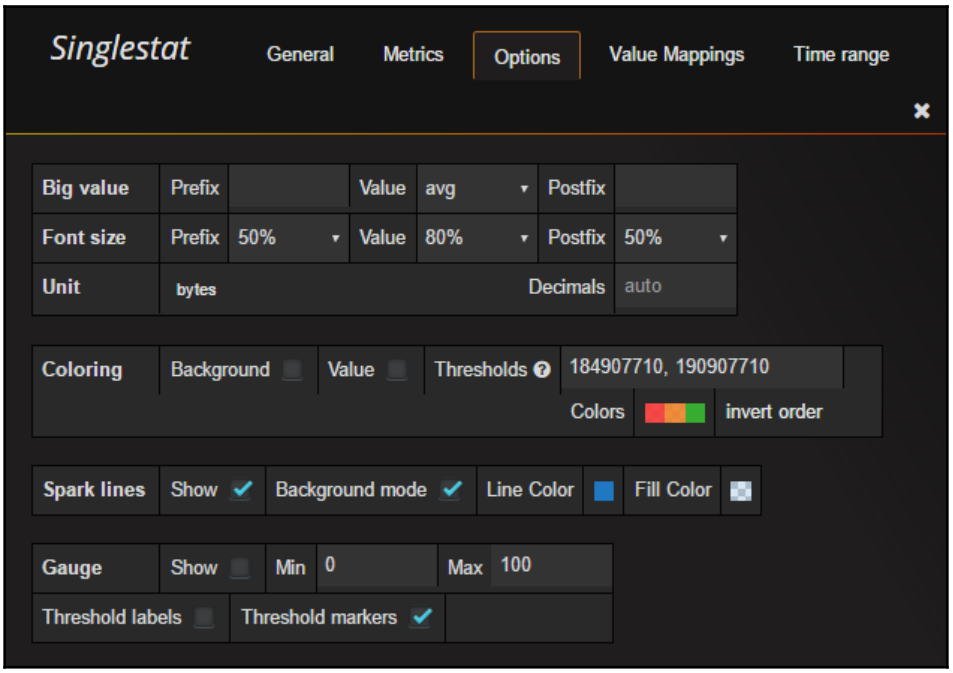
We don't want to lose this nice new graph we've created, so let's click on the save icon in the top-right corner. It looks like a floppy disk (you can do a Google image search if you don't know what this is).

After we click on the save icon, we will see a green dialog box that verifies that the dashboard was saved. We can now click the **x** symbol above the graph details panel and below the graph itself.

This will return us to the dashboard page. If we scroll all the way down, we will see our new graph. Let's add another panel to this row. Again, use the *green* tab and then select **Add Panel | singlestat**. Once again, an empty panel will appear with a setting form below it.

Let's say we want to watch a particular node and monitor network usage. We can easily do this by first going to the **Metrics** tab. Then, expand the query field and set the second value in the **FROM** field to **network/rx.** Now, we can specify the **WHERE** clause by clicking the **+** symbol at the end of the row and choosing **hostname** from the drop-down. After **hostname =**, click on **select tag value** and choose one of the minion nodes from the list.

Finally, leave **mean()** for the second **SELECT** field shown as follows:



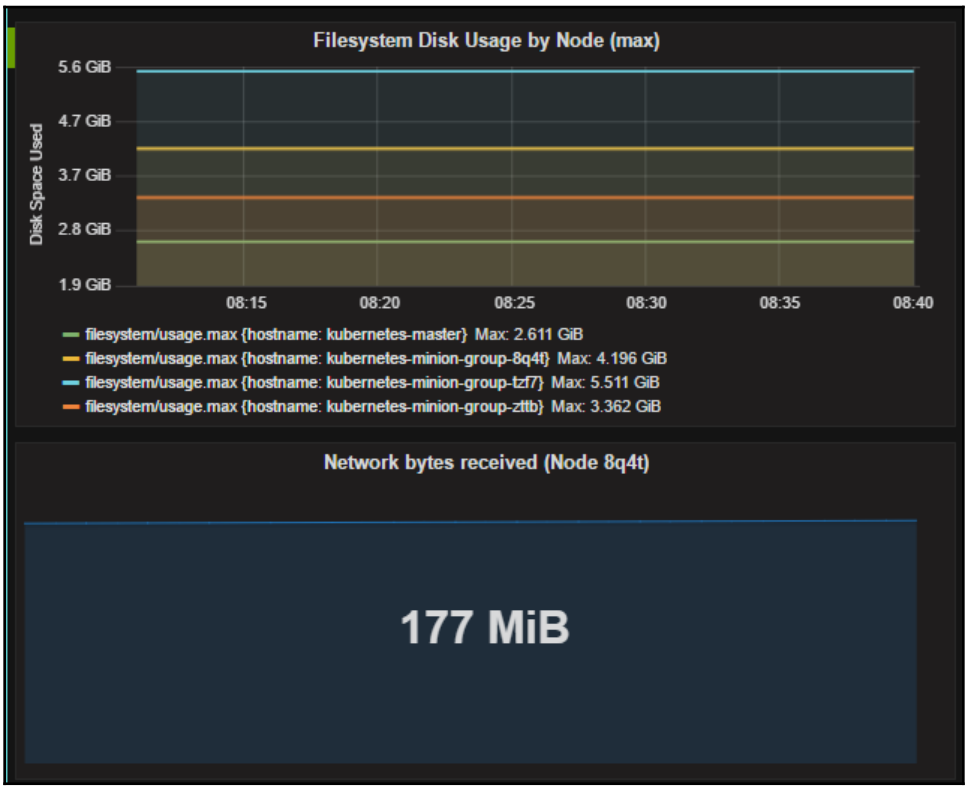
Singlestat options

In the **Options** tab, make sure that **Unit format** is set to **data | bytes** and check the **Show** box next to **Spark lines**. The spark line gives us a quick historical view of the recent variations in the value. We can use **Background mode** to take up the entire background; by default, it uses the area below the value.

In **Coloring**, we can optionally check the **Value** or **Background** box and choose **Thresholds** and **Colors.** This will allow us to choose different colors for the value based on the threshold tier we specify. Note that an unformatted version of the number must be used for threshold values.

Now, let's go back to the **General** tab and set the title as `Network bytes received (Node35ao)`. Use the identifier for your minion node.

Once again, let's save our work and return to the dashboard. We should now have a row that looks like the following screenshot:



Custom dashboard panels

Grafana has a number of other panel types that you can play with, such as **Dashboard list**, **Plugin list**, **Table**, and **Text**.

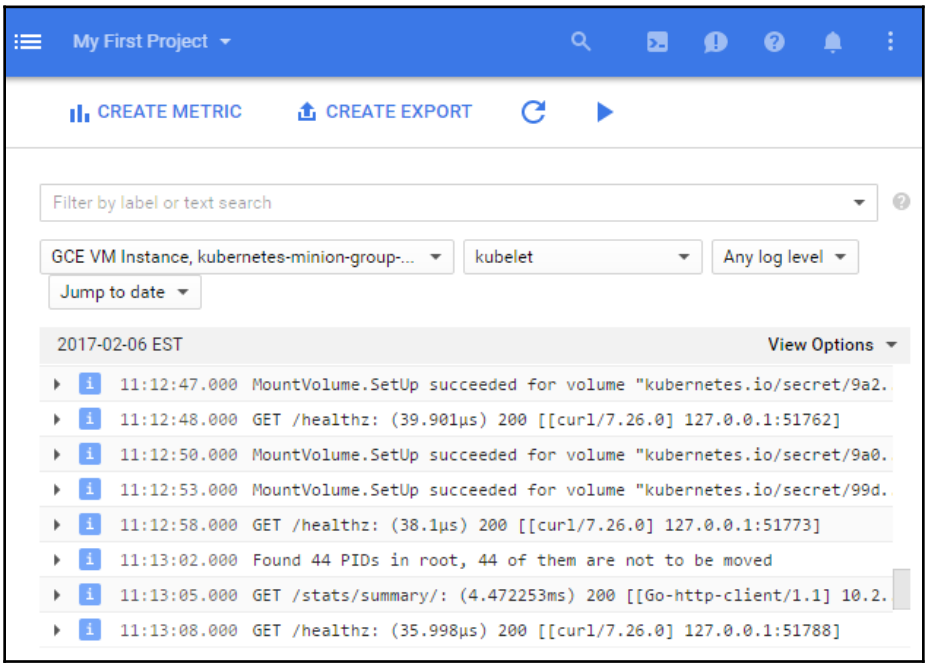
As we can see, it is pretty easy to build a custom dashboard and monitor the health of our cluster at a glance.

FluentD and Google Cloud Logging

Looking back at the *System pod listing* screenshot at the beginning of the chapter, you may have noted a number of pods starting with the words `fluentd-cloud-logging-kubernetes`. These pods appear when using the GCE provider for your K8s cluster.

A pod like this exists on every node in our cluster, and its sole purpose is to handle the processing of Kubernetes logs. If we log in to our Google Cloud Platform account, we can see some of the logs processed there. Simply use the left side, and under **Stackdriver**, select **Logging**. This will take us to a log listing page with a number of drop-down menus on the top. If this is your first time visiting the page, the first drop-down will likely be set to **Cloud HTTP Load Balancer**.

In this drop-down menu, we'll see a number of GCE types of entries. Select GCE VM instances and then the Kubernetes master or one of the nodes. In the second drop-down, we can choose various log groups, including **kubelet.** We can also filter by the event log level and date. Additionally, we can use the play button to watch events stream in live shown as follows:



The Google Cloud Logging filter

FluentD

Now we know that the `fluentd-cloud-logging-kubernetes` pods are sending the data to the Google Cloud, but why do we need FluentD? Simply put, FluentD is a collector.

It can be configured to have multiple sources to collect and tag logs, which are then sent to various output points for analysis, alerting, or archiving. We can even transform data using plugins before it is passed on to its destination.

Not all provider setups have FluentD installed by default, but it is one of the recommended approaches to give us greater flexibility for future monitoring operations. The AWS Kubernetes setup also uses FluentD, but instead forwards events to Elasticsearch.

Exploring FluentD: If you are curious about the inner workings of the FluentD setup or just want to customize the log collection, we can explore quite easily using the `kubectl exec` command and one of the pod names from the command we ran earlier in the chapter. First, let's see if we can find the FluentD `config` file: **`$ kubectl exec fluentd-cloud-logging-kubernetes-minion-group-r4qt --namespace=kube-system -- ls /etc/td-agent`**.

We will look in the `etc` folder and then `td-agent`, which is the `fluent` sub folder. While searching in this directory, we should see a `td-agent.conf` file. We can view that file with a simple `cat` command, as follows: **`$ kubectl exec fluentd-cloud-logging-kubernetes-minion-group-r4qt --namespace=kube-system -- cat /etc/td-agent/td-agent.conf`**.

We should see a number of sources, including the various Kubernetes components, Docker, and some GCP elements. While we can make changes here, remember that it is a running container and our changes won't be saved if the pod dies or is restarted. If we really want to customize, it's best to use this container as a base and build a new container, which we can push to a repository for later use.

Maturing our monitoring operations

While Grafana gives us a great start to monitoring our container operations, it is still a work in progress. In the real world of operations, having a complete dashboard view is great once we know there is a problem. However, in everyday scenarios, we'd prefer to be proactive and actually receive notifications when issues arise. This kind of alerting capability is a must to keep the operations team ahead of the curve and out of reactive mode.

There are many solutions available in this space, and we will take a look at two in particular: GCE monitoring (Stackdriver) and Sysdig.

GCE (Stackdriver)

Stackdriver is a great place to start for infrastructure in the public cloud. It is actually owned by Google, so it's integrated as the Google Cloud Platform monitoring service. Before your lock-in alarm bells start ringing, Stackdriver also has solid integration with AWS. In addition, Stackdriver has alerting capability with support for notification to a variety of platforms and webhooks for anything else.

Signing up for GCE monitoring

In the GCE console, in the **Stackdriver** section, click on **Monitoring.** This will open a new window, where we can sign up for a free trial of Stackdriver. We can then add our GCP project and optionally an AWS account as well. This requires a few more steps, but instructions are included on the page. Finally, we'll be given instructions on how to install the agents on our cluster nodes. We can skip this for now, but will come back to it in a minute.

Click on **Continue**, set up your daily alerts, and click on **Continue** again.

Click on **Launch Monitoring** to proceed. We'll be taken to the main dashboard page, where we will see some basic statistics on our node in the cluster. If we select **Resources** from the side menu and then **Instances**, we'll be taken to a page with all our nodes listed. By clicking on the individual node, we can again see some basic information even without an agent installed.

Stackdriver also offers monitoring and logging agents that can be installed on the nodes. However, it currently does not support the container OS that is used by default in the GCE `kube-up` script. You can still see the basic metrics for any nodes in GCE or AWS, but will need to use another OS if you want a detailed agent installation.

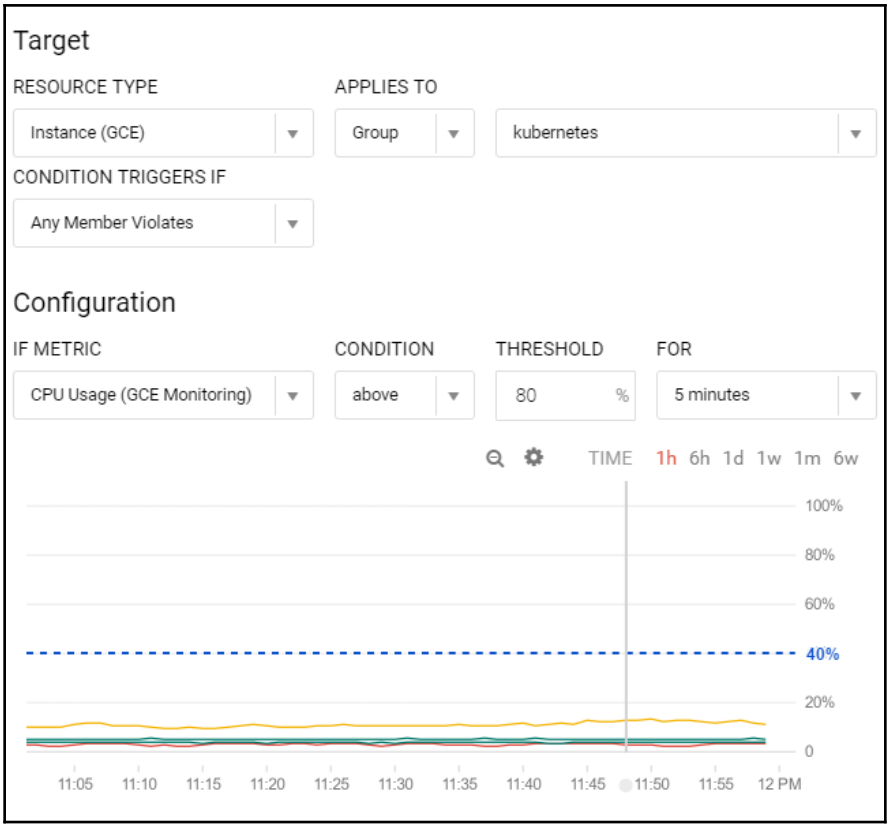
Alerts

Next, we can look at the alerting policies available as part of the monitoring service. From the instance details page, click on the **Create Alerting Policy** button in the **Incidents** section at the top of the page.

We will click on **Add Condition** and select a **Metric Threshold**. In the **Target** section, set **RESOURCE TYPE** to **Instance (GCE)**. Then, set **APPLIES TO** to **Group** and **kubernetes**. Leave **CONDITION TRIGGERS IF** set to **Any Member Violates**.

In the **Configuration** section, leave **IF METRIC** as **CPU Usage (GCE Monitoring)** and **CONDITION** as **above**. Now, set **THRESHOLD** to `80` and set the time in **FOR** to **5 minutes**.

Then click on **Save Condition**:



Google Cloud Monitoring alert policy

Next, we will add a notification. In the **Notification** section, leave **Method** as **Email** and enter your email address.

We can skip the **Documentation** section, but this is where we can add text and formatting to alert messages.

Finally, name the policy `Excessive CPU Load` and click on **Save Policy.**

Now, whenever the CPU from one of our instances goes above 80 percent, we will receive an email notification. If we ever need to review our policies, we can find them in the **Alerting** drop-down and then in **Policies Overview** in the menu on the left-hand side of the screen.

Beyond system monitoring with Sysdig

Monitoring our cloud systems is a great start, but what about the visibility of the containers themselves? Although there are a variety of cloud monitoring and visibility tools, Sysdig stands out for its ability to dive deep, not only into system operations, but specifically containers.

Sysdig is open source and is billed as a universal system visibility tool with native support for containers. It is a command line tool that provides insight into the areas we looked at earlier, such as storage, network, and system processes. What sets it apart is the level of detail and visibility it offers for these process and system activities. Furthermore, it has native support for containers, which gives us a full picture of our container operations. This is a highly recommended tool for your container operations arsenal. The main website of Sysdig is `http://www.sysdig.org/`.

Sysdig Cloud

We will take a look at the Sysdig tool and some of the useful command line-based UIs in a moment. However, the team at Sysdig has also built a commercial product, named **Sysdig Cloud**, which provides the advanced dashboard, alerting, and notification services we discussed earlier in the chapter. Also, the differentiator here has high visibility into containers, including some nice visualizations of our application topology.

If you'd rather skip the *Sysdig Cloud* section and just try out the command-line tool, simply skip to *The Sysdig command line* section later in this chapter.

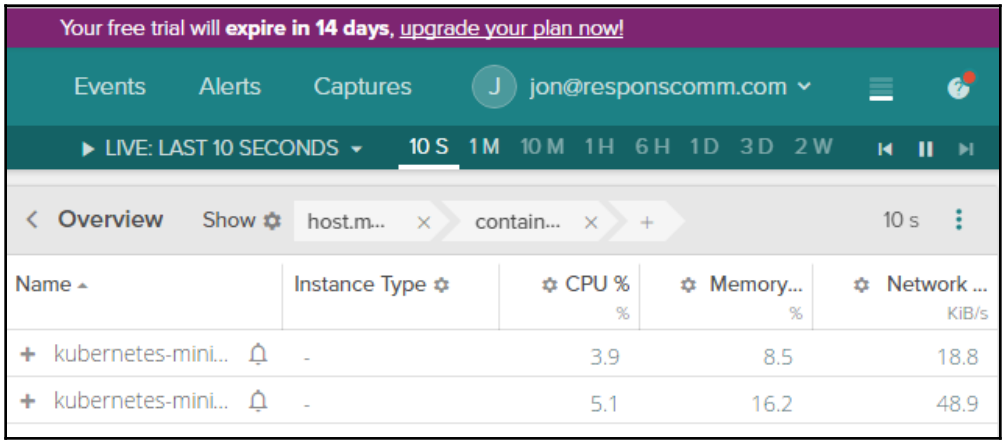
If you have not done so already, sign up for Sysdig Cloud at `http://www.sysdigcloud.com`.

After activating and logging in for the first time, we'll be taken to a welcome page. Clicking on **Next**, we are shown a page with various options to install the Sysdig agents. For our example environment, we will use the Kubernetes setup. Selecting Kubernetes will give you a page with your API key and a link to instructions. The instructions will walk you through how to create a Sysdig agent DaemonSet on your cluster. Don't forget to add the API key from the install page.

We will not be able to continue on the install page until the agents connect. After creating the DaemonSet and waiting a moment, the page should continue to the AWS integration page. You can fill this out if you like, but for this walk-through, we will click on **Skip**. Then, click on **Let's Get Started**.

As of this writing, Sysdig and Sysdig Cloud were not fully compatible with the latest container OS deployed by default in the GCE `kube-up` script, Container-optimized OS from Google: `https://cloud.google.com/container-optimized-os/docs`.

We'll be taken to the main **Sysdig Cloud** dashboard screen. We should see at least two minion nodes appear under the **Explore** tab. We should see something similar to the following screenshot with our minion nodes:



Sysdig Cloud Explore page

This page shows us a table view, and the links on the left let us explore some key metrics for CPU, memory, networking, and so on. Although this is a great start, the detailed views will give us a much deeper look at each node.

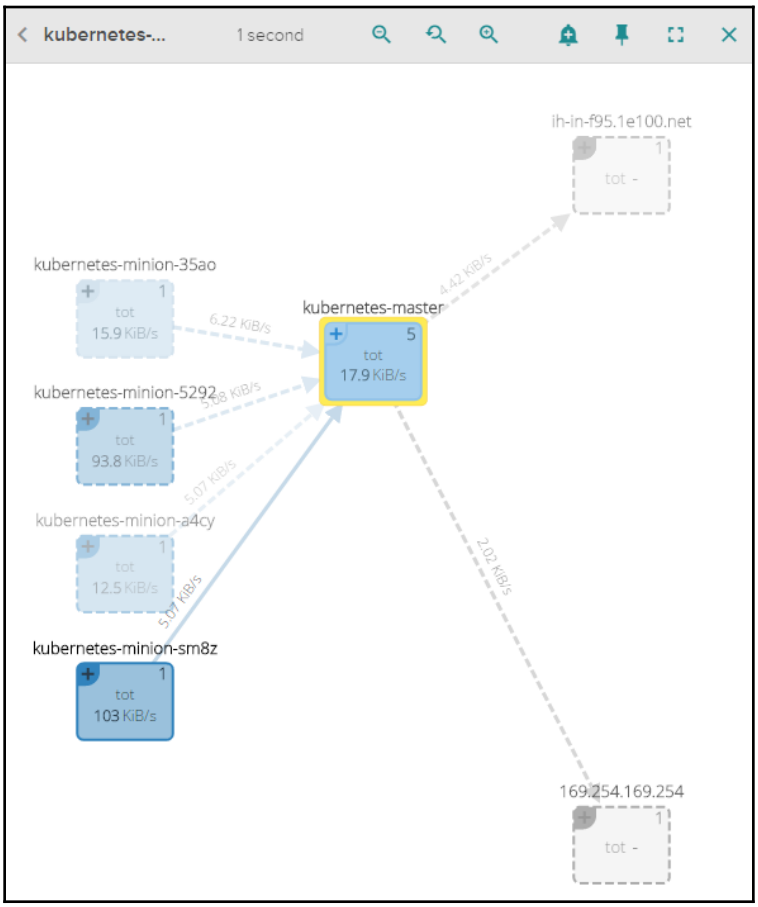
Detailed views

Let's take a look at these views. Select one of the minion nodes and then scroll down to the detail section that appears below. By default, we should see the **System: Overview by Process** view (if it's not selected, just click on it from the list on the left-hand side). If the chart is hard to read, simply use the maximize icon in the top-left corner of each graph for a larger view.

There are a variety of interesting views to explore. Just to call out a few others, **Services** | **HTTP Overview** and **Hosts & Containers** | **Overview by Container** give us some great charts for inspection. In the latter view, we can see stats for CPU, memory, network, and file usage by container.

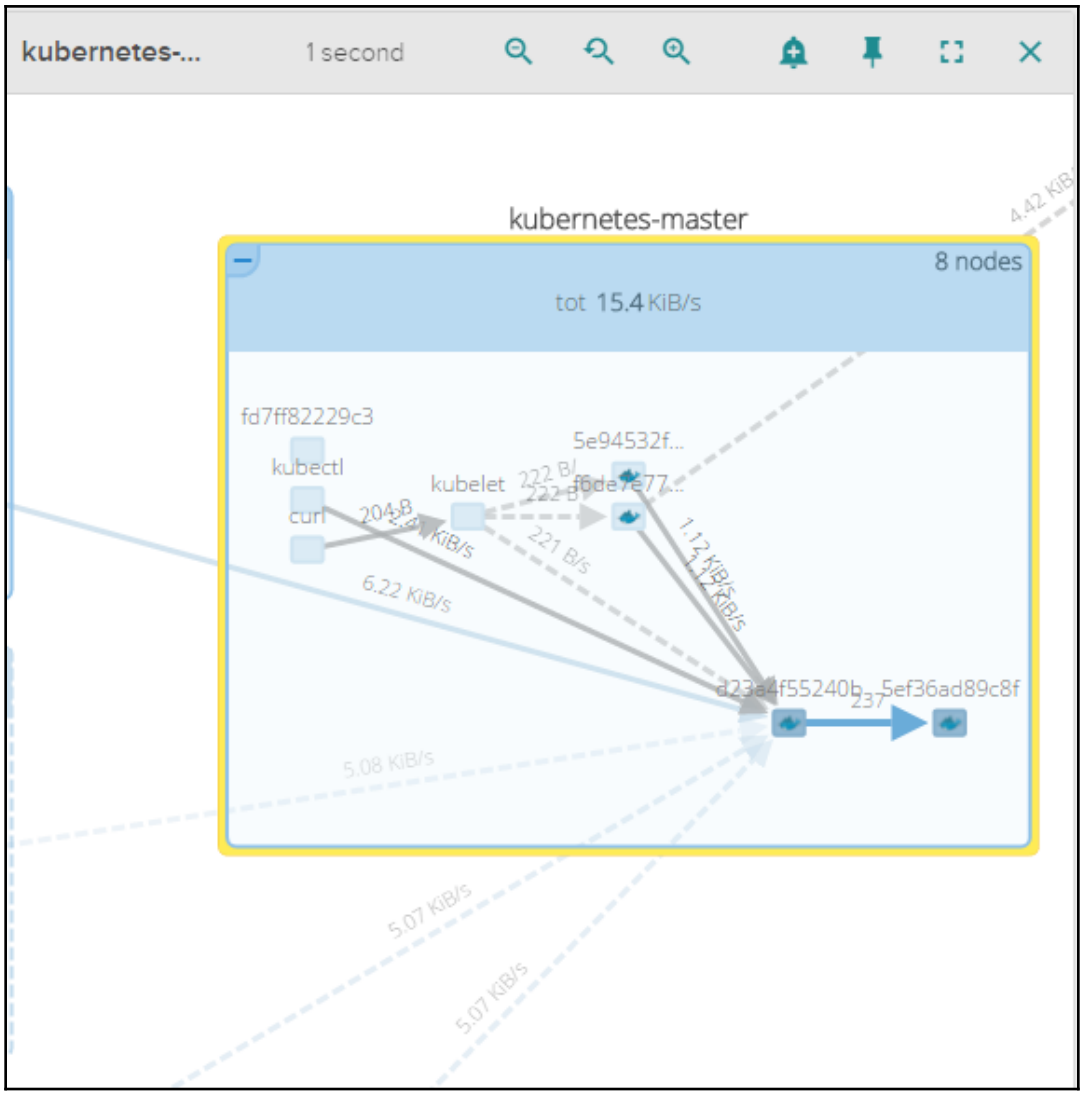
Topology views

In addition, there are three topology views at the bottom. These views are perfect for helping us understand how our application is communicating. Click on **Topology** | **Network Traffic** and wait a few seconds for the view to fully populate. It should look similar to the following screenshot:



Sysdig Cloud network topology view

Note that the view maps out the flow of communication between the minion nodes and the master in the cluster. You may also note a **+** symbol in the top corner of the node boxes. Click on that in one of the minion nodes and use the zoom tools at the top of the view area to zoom into the details, as shown in the following screenshot:



The Sysdig Cloud network topology detailed view

Note that we can now see all the components of Kubernetes running inside the master. We can see how the various components work together. We can see `kube-proxy` and the `kubelet` process running, as well as a number of boxes with the Docker whale, which indicate that they are containers. If we zoom in and use the plus icon, we can see that these are the containers for our pods and core Kubernetes processes, as we saw in the services running on the master section in `Chapter 1`, *Introduction to Kubernetes*.

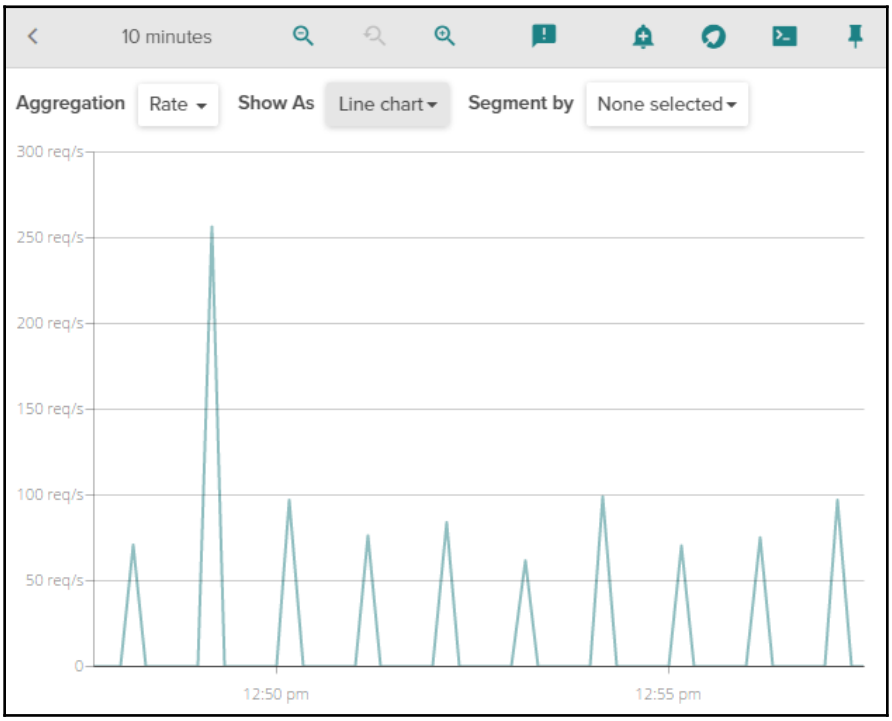
Also, if you have the master included in your monitored nodes, you can watch `kubelet` initiate communication from a minion and follow it all the way through the `kube-apiserver` container in the master.

We can even sometimes see the instance communicating with the GCE infrastructure to update metadata. This view is great in order to get a mental picture of how our infrastructure and underlying containers are talking to one another.

Metrics

Next, let's switch over to the **Metrics** tab in the left-hand menu next to **Views**. Here, there are also a variety of helpful views.

Let's look at **capacity.estimated.request.total.count** in **System**. This view shows us an estimate of how many requests a node is capable of handling when fully loaded. This can be really useful for infrastructure planning:

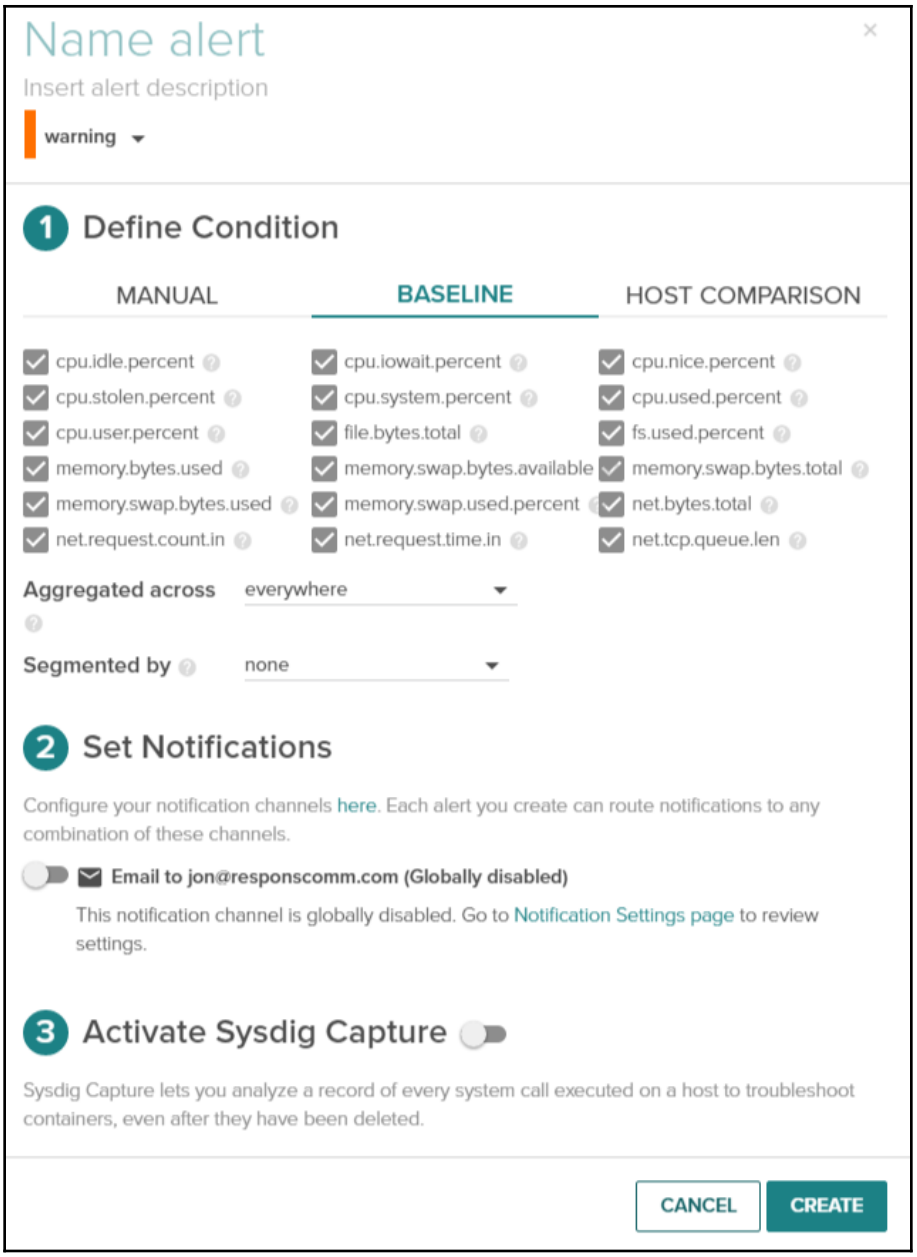


Sysdig Cloud capacity estimate view

Alerting

Now that we have all this great information, let's create some notifications. Scroll back up to the top of the page and find the bell icon next to one of your minion entries. This will open a **Create Alert** dialog. Here, we can set manual alerts similar to what we did earlier in the chapter. However, there is also the option to use **BASELINE** and **HOST COMPARISON**.

Using the **BASELINE** option is extremely helpful, as Sysdig will watch the historical patterns of the node and alert us whenever one of the metrics strays outside the expected metric thresholds. No manual settings are required, so this can really save time for the notification setup and help our operations team to be proactive before issues arise. Refer to the following screenshot:



Sysdig Cloud new alert

The **HOST COMPARISON** option is also a great help as it allows us to compare metrics with other hosts and alert whenever one host has a metric that differs significantly from the group. A great use case for this is monitoring resource usage across minion nodes to ensure that our scheduling constraints are not creating a bottleneck somewhere in the cluster.

You can choose whichever option you like and give it a name and warning level. Enable the notification method. Sysdig supports email, **SNS** (short for **Simple Notification Service**), and **PagerDuty** as notification methods. You can optionally enable **Sysdig Capture** to gain deeper insight into issues. Once you have everything set, just click on **Create** and you will start to receive alerts as issues come up.

The Sysdig command line

Whether you only use the open source tool or you are trying out the full Sysdig Cloud package, the command line utility is a great companion to have to track down issues or get a deeper understanding of your system.

In the core tool, there is the main `sysdig` utility and also a command line-style UI named `csysdig`. Let's take a look at a few useful commands.

Find the relevant installation instructions for your OS here: `http://www.sysdig.org/install/`.

Once installed, let's first look at the process with the most network activity by issuing the following command:

```
$ sudo sysdig -pc -c topprocs_net
```

The following screenshot is the result of the preceding command:

Bytes	Process	Host_pid	Container_pid	container.name
79.06KB	kube-apise	5152	15	host
58.10KB	etcd	5211	10	host
6.29KB	dragent	19284	19292	host
4.52KB	kube-contr	5164	11	host
4.11KB	etcd	5211	11	host
1.95KB	kube-sched	5227	13	host
1.72KB	sshd	18963	18963	host

A Sysdig top process by network activity

This is an interactive view that will show us a top process in terms of network activity. Also, there are a plethora of commands to use with `sysdig`. A few other useful commands to try out include the following:

```
$ sudo sysdig -pc -c topprocs_cpu
$ sudo sysdig -pc -c topprocs_file
$ sudo sysdig -pc -c topprocs_cpu container.name=<Container Name NOT ID>
```

More examples can be found at `http://www.sysdig.org/wiki/sysdig-examples/`.

The Csysdig command-line UI

Just because we are in a shell on one of our nodes doesn't mean we can't have a UI. Csysdig is a customizable UI for exploring all the metrics and insight that Sysdig provides. Simply type `csysdig` in the prompt:

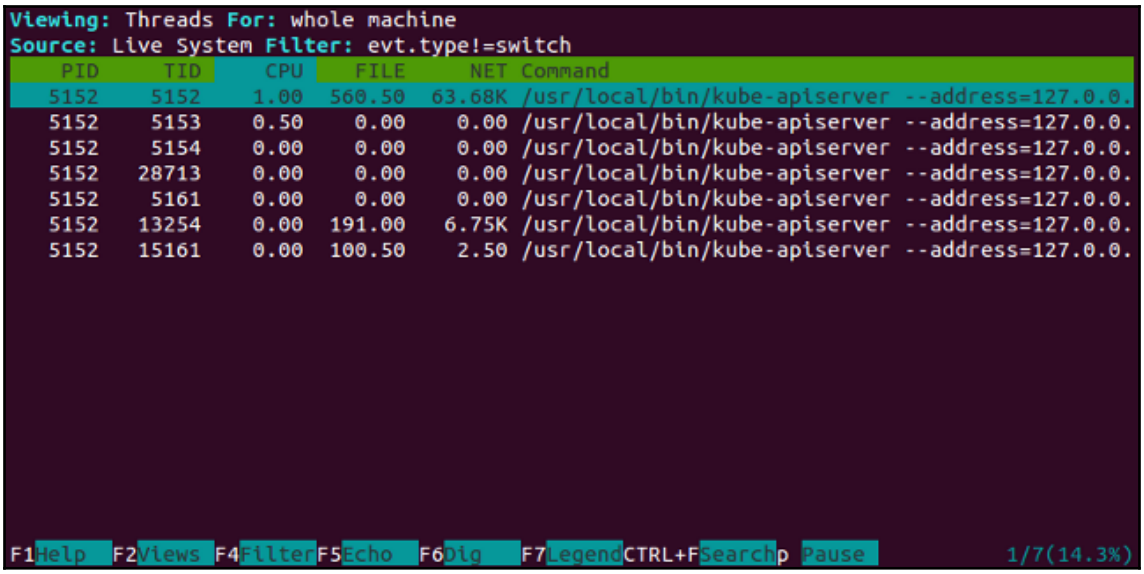
```
$ csysdig
```

After entering `csysdig`, we will see a real-time listing of all processes on the machine. At the bottom of the screen, you'll note a menu with various options. Click on **Views** or press *F2* if you love to use your keyboard. In the left-hand menu, there are a variety of options, but we'll look at threads. Double-click on **Threads**.

On some operating systems and with some SSH clients, you may have issues with the function keys. Check the settings on your terminal and make sure that the function keys are using the VT100+ sequences.

We can see all the threads currently running on the system and some information about the resource usage. By default, we see a big list that is updated often. If we click on the **Filter**, *F4* for the mouse-challenged, we can slim down the list.

Type `kube-apiserver`, if you are on the master, or `kube-proxy`, if you are on a node (minion), in the filter box and press *Enter*. The view now filters for only the threads in that command:



Csysdig threads

If we want to inspect this a little further, we can simply select one of the threads in the list and click on **Dig** or press *F6*. Now, we see a detailed listing of system calls from the command in real time. This can be a really useful tool to gain deep insight into the containers and processes running on our cluster.

Click on **Back** or press the *Backspace* key to go back to the previous screen. Then, go to **Views** once more. This time, we will look at the **Containers** view. Once again, we can filter and also use the **Dig** view to get more in-depth visibility into what is happening at the system call level.

Another menu item you might note here is **Actions**, which is available in the newest release. These features allow us to go from process monitoring to action and response. It gives us the ability to perform a variety of actions from the various process views in Csysdig. For example, the container view has actions to drop into a Bash shell, kill containers, inspect logs, and more. It's worth getting to know the various actions and hotkeys, and even add your own custom hotkeys for common operations.

Prometheus

A newcomer to the monitoring scene is an open source tool called Prometheus. Prometheus is an open source monitoring tool that was built by a team at SoundCloud. You can find more about the project at `https://prometheus.io`.

Their website offers the following features:

- A multi-dimensional data model (`https://prometheus.io/docs/concepts/data_model/`) (the time series are identified by their metric name and key/value pairs)
- A flexible query language (`https://prometheus.io/docs/prometheus/latest/querying/basics/`) to leverage this dimensionality
- No reliance on distributed storage; single-server nodes are autonomous
- Time series collection happens via a pull model over HTTP
- Pushing time series (`https://prometheus.io/docs/instrumenting/pushing/`) is supported via an intermediary gateway
- Targets are discovered via service discovery or static configuration
- Multiple modes of graphing and dashboard support

Prometheus summary

Prometheus offers a lot of value to the operators of a Kubernetes cluster. Let's look at some of the more important dimensions of the software:

- **Simple to operate**: It was built to run as individual servers using local storage for reliability
- **It's precise**: You can use a query language similar to JQL, DDL, DCL, or SQL queries to define alerts and provide a multi-dimensional view of status
- **Lots of libraries**: You can use more than ten languages and numerous client libraries in order to introspect your services and software
- **Efficient**: With data stored in an efficient, custom format both in memory and on disk, you can scale out easily with sharding and federation, creating a strong platform from which to issue powerful queries that can construct powerful data models and ad hoc tables, graphs, and alerts

Also, Promethus is 100% open source and is (as of July 2018) currently an incubating project in the CNCF. You can install it with Helm as we did with other software, or do a manual installation as we'll detail here. Part of the reason that we're going to look at Prometheus today is due to the overall complexity of the Kubernetes system. With lots of moving parts, many servers, and potentially differing geographic regions, we need a system that can cope with all of that complexity.

A nice part about Prometheus is the pull nature, which allows you to focus on exposing metrics on your nodes as plain text via HTTP, which Prometheus can then pull back to a central monitoring and logging location. It's also written in Go and inspired by the closed source Borgmon system, which makes it a perfect match for our Kubernetes cluster. Let's get started with an install!

Prometheus installation choices

As with previous examples, we'll need to either use our local Minikube install or the GCP cluster that we've spun up. Log in to your cluster of choice, and then let's get Prometheus set up. There's actually lots of options for installing Prometheus due to the fast moving nature of the software:

- The simplest, manual method; if you'd like to build the software from the getting started documents, you can jump in with `https://prometheus.io/docs/prometheus/latest/getting_started/` and get Prometheus monitoring itself.
- The middle ground, with Helm; if you'd like to take the middle road, you can install Prometheus on your cluster with Helm (`https://github.com/helm/charts/tree/master/stable/prometheus`).
- The advanced `Operator` method; if you want to use the latest and greatest, let's take a look at the Kubernetes `Operator` class of software, and use it to install Prometheus. The `Operator` was created by CoreOS, who have recently been acquired by Red Hat. That should mean interesting things for Project Atomic and Container Linux. We'll talk more about that later, however! We'll use the Operator model here.

The Operator is designed to build upon the Helm-style management of software in order to build additional human operational knowledge into the installation, maintenance, and recovery of applications. You can think of the Operator software just like an SRE Operator: someone who's an expert in running a piece of software.

An Operator is an application-specific controller that extends the Kubernetes API in order to manage complex stateful applications such as caches, monitoring systems, and relational or non-relational databases. The Operator uses the API in order to create, configure, and manage these stateful systems on behalf of the user. While Deployments are excellent in dealing with seamless management of stateless web applications, the Deployment object in Kubernetes struggles to orchestrate all of the moving pieces in a stateful application when it comes to scaling, upgrading, recovering from failure, and reconfiguring these systems.

You can read more about extending the Kubernetes API here: `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/`.

Operators leverage some core Kubernetes concepts that we've discussed in other chapters. Resources (ReplicaSets) and Controllers (for example Deployments, Services, and DaemonSets) are leverage with additional operational knowledge of the manual steps that are encoded in the Operator software. For example, when you scale up an etcd cluster manually, one of the key steps in the process is to create a DNS name for the new etcd member that can be used to route to the new member once it's been added to the cluster. With the Operator pattern being used, that systematized knowledge is built into the `Operator` class to provide the cluster administrator with seamless updates to the etcd software.

The difficulty in creating operators is understanding the underlying functionality of the stateful software in question, and then encoding that into a resource configuration and control loop. Keep in mind that Kubernetes can be thought of as simply being a large distributed messaging queue, with messages that exist in the form of a YAML blob of declarative state that the cluster operator defines, which the Kubernetes system puts into place.

Tips for creating an Operator

If you want to create your own `Operator` in the future, you can keep the following tips from CoreOS in mind. Given the nature of their application-specific domain, you'll need to keep a few things in mind when managing complex applications. First, you'll have a set of system flow activities that your `Operator` should be able to perform. This will be actions such as creating a user, creating a database, modifying user permissions and passwords, and deleting users (such as the default user installed when creating many systems).

You'll also need to manage your installation dependencies, which are the items that need to be present and configured for your system to work in the first place. CoreOS also recommends the following principles be followed when creating an `Operator`:

- **Single step to deploy**: Make sure your `Operator` can be initialized and run with a single command that takes no additional work to get running.
- **New third-party type**: Your `Operator` should leverage the third-party API types, which users will take advantage of when creating applications that use your software.
- **Use the basics**: Make sure that your `Operator` uses the core Kubernetes objects such as ReplicaSets, Services, and StatefulSets, in order to leverage all of the hard work being poured into the open source Kubernetes project.
- **Compatible and default working**: Make sure you build your `Operators` so that they exist in harmony with older versions, and design your system so that it still continues to run unaffected if the `Operator` is stopped or accidentally deleted from your cluster.
- **Version**: Make sure to facilitate the ability to version instances of your `Operator`, so cluster administrators don't shy away from updating your software.
- **Test**: Also, make sure to test your `Operator` against a destructive force such as a Chaos Monkey! Your `Operator` should be able to survive the failure of nodes, pods, storage, configuration, and networking outages.

Installing Prometheus

Let's run through an install of Prometheus using the new pattern that we've discovered. First, let's use the Prometheus definition file to create the deployment. We'll use Helm here to install the Operator!

Make sure you have Helm installed, and then make sure you've initialized it:

```
$ helm init
master $ helm init
Creating /root/.helm
...
Adding stable repo with URL:
https://kubernetes-charts.storage.googleapis.com
Adding local repo with URL: http://127.0.0.1:8879/charts
$HELM_HOME has been configured at /root/.helm.
...
Happy Helming!
$
```

Next, we can install the various `Operator` packages required for this demo:

```
$ helm repo add coreos
https://s3-eu-west-1.amazonaws.com/coreos-charts/stable/
"coreos" has been added to your repositories
```

Now, install the `Operator`:

```
$ helm install coreos/prometheus-operator --name prometheus-operator
```

You can see that it's installed and running by first checking the installation:

```
$ helm ls prometheus-operator
NAME                    REVISION UPDATED                         STATUS
CHART NAMESPACE
prometheus-operator     1 Mon Jul 23 02:10:18 2018        DEPLOYED
prometheus-operator-0.0.28 default
```

Then, look at the pods:

```
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
prometheus-operator-d75587d6-bmmvx 1/1 Running 0 2m
```

Now, we can install `kube-prometheus` to get all of our dependencies up and running:

```
$ helm install coreos/kube-prometheus --name kube-prometheus --set
global.rbacEnable=true
NAME:   kube-prometheus
LAST DEPLOYED: Mon Jul 23 02:15:59 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Alertmanager
NAME             AGE
kube-prometheus  1s

==> v1/Pod(related)
NAME                                                READY STATUS RESTARTS
AGE
kube-prometheus-exporter-node-45rwl                 0/1 ContainerCreating
0 1s
kube-prometheus-exporter-node-d84mp                 0/1 ContainerCreating
0 1s
kube-prometheus-exporter-kube-state-844bb6f589-z58b6  0/2 ContainerCreating
0 1s
kube-prometheus-grafana-57d5b4d79f-mgqw5            0/2 ContainerCreating
0 1s
```

```
==> v1beta1/ClusterRoleBinding
NAME                                        AGE
psp-kube-prometheus-alertmanager            1s
kube-prometheus-exporter-kube-state         1s
psp-kube-prometheus-exporter-kube-state     1s
psp-kube-prometheus-exporter-node           1s
psp-kube-prometheus-grafana                 1s
kube-prometheus                             1s
psp-kube-prometheus                         1s
...
```

We've truncated the output here as there's a lot of information. Let's look at the pods again:

```
$ kubectl get pods
NAME                                                  READY STATUS
RESTARTS AGE
alertmanager-kube-prometheus-0                        2/2 Running 0 3m
kube-prometheus-exporter-kube-state-85975c8577-vfl6t  2/2
Running 0 2m
kube-prometheus-exporter-node-45rwl                   1/1 Running 0 3m
kube-prometheus-exporter-node-d84mp                   1/1 Running 0 3m
kube-prometheus-grafana-57d5b4d79f-mgqw5              2/2 Running 0 3m
prometheus-kube-prometheus-0                          3/3 Running 1 3m
prometheus-operator-d75587d6-bmmvx                    1/1 Running 0 8m
```

Nicely done!

If you forward the port for `prometheus-kube-prometheus-0` to `8448`, you should be able to see the Prometheus dashboard, which we'll revisit in later chapters as we explore high availability and the productionalization of your Kubernetes cluster. You can check this out at `http://localhost:8449/alerts`.

Summary

We took a quick look at monitoring and logging with Kubernetes. You should now be familiar with how Kubernetes uses cAdvisor and Heapster to collect metrics on all the resources in a given cluster. Furthermore, we saw how Kubernetes saves us time by providing InfluxDB and Grafana set up and configured out of the box. Dashboards are easily customizable for our everyday operational needs.

In addition, we looked at the built-in logging capabilities with FluentD and the Google Cloud Logging service. Also, Kubernetes gives us great time savings by setting up the basics for us.

Finally, you learned about the various third-party options available to monitor our containers and clusters. Using these tools will allow us to gain even more insight into the health and status of our applications. All these tools combine to give us a solid toolset to manage day-to-day operations. Lastly, we explored different methods of installing Prometheus, with an eye on building more robust production systems.

In the next chapter, we will explore the new cluster federation capabilities. Still mostly in beta, this functionality will allow us to run multiple clusters in different data centers and even clouds, but manage and distribute applications from a single control plane.

Questions

1. Name two of the built-in monitoring tools for Kubernetes
2. What namespace do the built-in monitoring tools run in?
3. What graphing software is used by most of the monitoring tools?
4. What is FluentD referred to as?
5. What's Google's native monitoring system?
6. What are two good reasons to use Prometheus?

Further reading

If you'd like to read more about the Kubernetes Operator Framework, check out this blog post: `https://coreos.com/blog/introducing-operator-framework`.

If you'd like to check out a video on Kubernetes monitoring from Packt, see this video: `https://www.packtpub.com/mapt/video/virtualization_and_cloud/9781789130003/65553/65558/monitoring-your-infrastructure`.

9
Operating Systems, Platforms, and Cloud and Local Providers

The first half of this chapter will cover how open standards encourage a diverse ecosystem of container implementations. We'll look at the **Open Container Initiative** (**OCI**) and its mission to provide an open container specification as well. The second half of this chapter will cover the various operating systems available for running containerized workloads, such as CoreOS. We'll also look at its advantages as a host OS, including performance and support for various container implementations. Additionally, we'll take a brief look at the Tectonic Enterprise offering from CoreOS. We'll look at the various hosted platforms offered by the major **cloud service providers** (**CSPs**) and see how they stack up.

This chapter will discuss the following topics:

- Why do standards matter?
- The OCI and the **Cloud Native Computing Foundation** (**CNCF**)
- Container specifications versus implementations
- Various container-oriented operating systems
- Tectonic
- The CSP platforms available that can run Kubernetes workloads

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in, or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes online at `https://labs.play-with-k8s.com/`.

You'll also need GitHub credentials, which we'll go over setting up later in the chapter.

The GitHub repository for this chapter can be found at `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter09`.

The importance of standards

Over the past two years, containerization technology has had a tremendous growth in popularity. While Docker has been at the center of this ecosystem, there is an increasing number of players in the container space. There are already a number of alternatives to the containerization and Docker implementation itself (rkt, Garden, and so on). In addition, there is a rich ecosystem of third-party tools that enhance and complement your container infrastructure. While Kubernetes is designed to manage the state of a container and the orchestration, scheduling, and networking side of this ecosystem, the bottom line is that all of these tools form the basis to build cloud-native applications.

As we mentioned at the very beginning of this book, one of the most attractive things about containers is their ability to package our application for deployment across various environment tiers (that is, development, testing, and production) and various infrastructure providers (GCP, AWS, on-premises, and so on).

To truly support this type of deployment agility, we need not only the containers themselves to have a common platform, but also the underlying specifications to follow a common set of ground rules. This will allow for implementations that are both flexible and highly specialized. For example, some workloads may need to be run on a highly secure implementation. To provide this, the implementation will have to make more intentional decisions about some aspects of the implementation. In either case, we will have more agility and freedom if our containers are built on some common structures that all implementations agree on and support.

In the following pages, we'll explore the building blocks of the many competing standards in the Kubernetes ecosystem. We'll explain how they're changing and developing and what part they may play in the future.

One of the examples that we'll explore more deeply in this third edition is the CRI-O project, which came to be after the creation of the OCI Charter. Let's make sure we understand the importance of that mission.

The OCI Charter

The mission of the OCI Charter is to ensure that the open source community has a stable platform from which industry participants can contribute the portable, open, and vendor-neutral runtimes required to build container-powered applications. The Linux Foundation is the holder of the charter, which is a sister organization to the CNCF. We'll look more into the implications of a foundation in `Chapter 11`, *Kubernetes SIGs, Incubation Projects, and the CNCF*.

If you'd like to read more about these foundations, you can check out their websites here: `https://www.linuxfoundation.org/` and `https://www.cncf.io/`.

While the OCI Charter tries to standardize the building blocks of the ecosystem, it does not attempt to define the system at the macroscopic level, nor does it market a particular pathway or solution. There's also a process defined that helps technology mature in a responsible way through these foundations, to ensure that the best possible technology is reaching the end user. These are defined as the following stages:

1. Sandbox
2. Incubating
3. Graduated

For the specifics of this chapter as regards the OCI, let's look at what else they're trying to accomplish. Firstly, we're attempting to create a format specification. This specification will call out a few important dimensions in order to create a consensus:

- **Provide a format**: In order to ensure a specification that can be used across multiple runtimes, you need a standard container format and runtime specification. The container format is represented by the root filesystem that sits on the disk, with the necessary additional configuration that allows a given container to be run on the system. There is a push to categorize the standardization into the following layers: base, optional, and out of scope.
- **Provide a runtime**: This is more straightforward, as it's designed to provide an executable that can directly run a container via consumption of the aforementioned container format and runtime specification.

The Charter also incentivizes a number of projects, the first two of which are the runc projects, and the third of which involves the definition of its own specifications in the OCI Specification project. New projects are added by members through a review process that needs two-thirds approval from the current **Technical Oversight Board** (**TOB**). If we look deeper into the principles that govern the OCI, the website names six guiding principles:

- Technology leadership
- Influence through contribution
- Limited scope, limited politics
- Minimalist structure
- Representative leadership
- Adherence to anti-trust regulations

These items are a blend of philosophical and logical frameworks that encourage competition, collaboration, meritocracy, and the continuous improvement cycles that many Agile and DevOps practitioners have long utilized.

Let's dig more into the initiative itself now.

The OCI

One of the first initiatives to gain widespread industry engagement is the OCI. Among the 36 industry collaborators are Docker, Red Hat, VMware, IBM, Google, and AWS, as listed on the OCI website at `https://www.opencontainers.org/`.

The purpose of the OCI is to split implementations, such as Docker and rkt, from a standard specification for the format and runtime of containerized workloads. According to their own terms, the goal of the OCI specifications has three basic tenets (you can refer to more details about this in the *Further reading* section at the end of the chapter):

- Creating a formal specification for container image formats and runtime, which will allow a compliant container to be portable across all major, compliant operating systems and platforms without artificial technical barriers.
- Accepting, maintaining, and advancing the projects associated with these standards. It will look to agree on a standard set of container actions (start, exec, pause, and so on), as well as a runtime environment associated with a container runtime.
- Harmonizing the previously referenced standard with other proposed standards, including the appc specification.

By following these principals, the OCI hopes to bolster a collaborative and inclusive ecosystem that provides a rich and evolving toolset to meet the needs of today's complex application workloads, be they cloud-native or traditional.

There are additionally some guiding principles for the development of standards in this space. These principles were integrated from the founding beliefs of the folks who created appc, and are as follows:

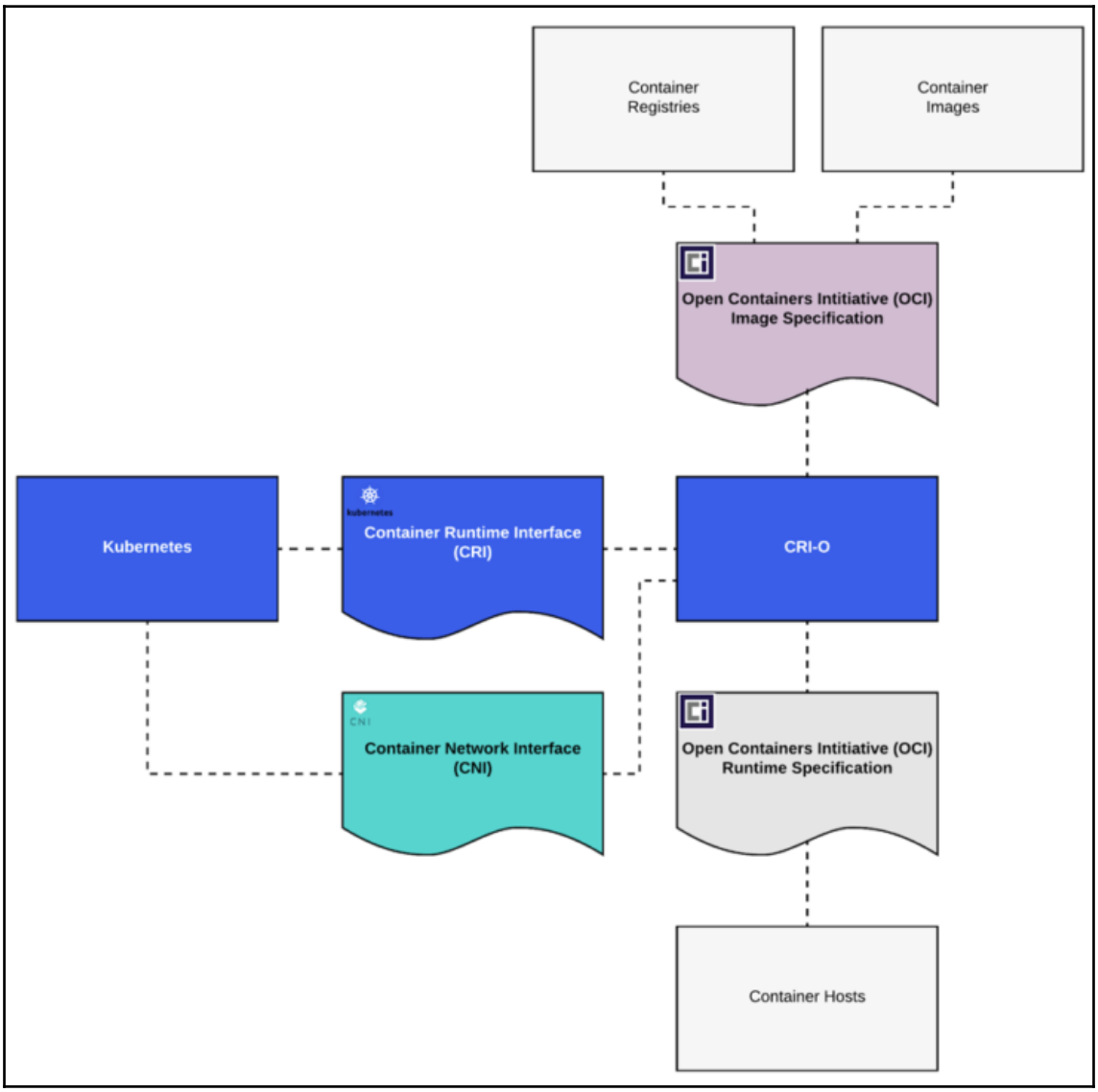
- **Security**: Isolate containers via pluggable interfaces using secure cryptographic principles, and a chain of custody for both images and application code.
- **Portability**: Ensure that containers continue to be portable across a wide variety software, clouds, and hardware.
- **Decentralized**: Container images should be straightforward and should take advantage of federation and namespacing.
- **Open**: The runtime and formats should be community-built, with multiple interchangeable parts.
- **Backward compatible**: Given the popularity of Docker and containers with nearly 9 billion downloads, backward compatibility should be given high priority.
- **Composable**: Tools for the operation of containers should be well integrated, but modular.
- **Code**: Consensus should be built from running, working code that follows principles of minimalism that adhere to domain-driven design. It should be stable and extensible.

Container Runtime Interface

Let's look at one of the newer and Kubernetes-specific OCI-based initiatives, CRI-O. CRI-O is currently part of the Kubernetes incubator, but it may move out to its own project as it matures. One of the compelling parts of the CRI-O design is that it never breaks Kubernetes. This is different because other runtimes are designed to do many things, such as building images, managing security, orchestration, and inspecting images. CRI-O is only designed to help Kubernetes orchestrate and schedule containers.

You can get the code for the CRI-O project and read the documentation at `https://github.com/kubernetes-incubator/cri-o/`.

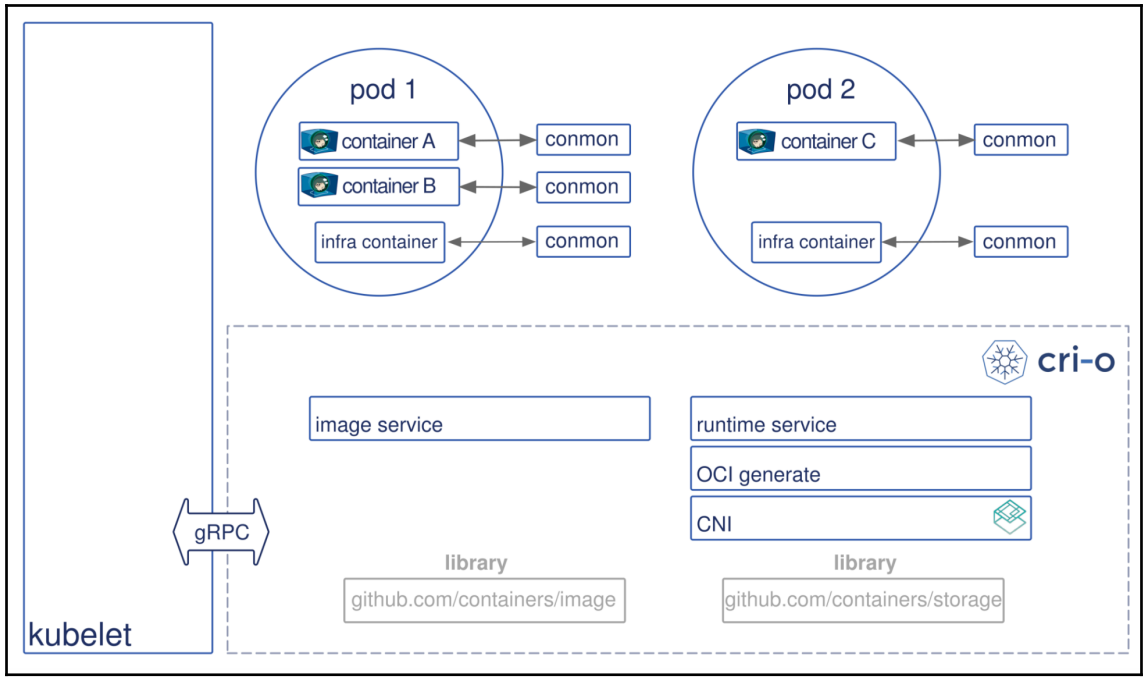
To this end, CRI-O is developed congruently with the CRI itself, and aligns itself with upstream releases of the Kubernetes system. The following diagram shows how the CRI-O works with the OCI:



In order to achieve this workflow, the following happens:

1. The operator decides to start a pod, which causes Kubernetes to use the `kubelet` to start a pod. That `kubelet` talks through the CRI to the CRI-O daemon.
2. CRI-O then uses several libraries, built with the OCI standard, to pull and unpack the given container image from a registry. From these operations, CRI-O generates a JSON blob that is used in the next step to run the container.
3. CRI-O kicks off an OCI-compatible runtime, which then runs the container process. This could be runc or the new Kata Container runtime (which has absorbed Intel's clear containers initiative).

You'll notice here that the CRI-O is acting as an interleaving layer between the libraries and runtimes, such that it's using standard formats to accomplish most its goals. This ensures the goal is making Kubernetes work at all times. Here's a diagram showing the system of the flow that was described in this section:



For networking, CRI-O would leverage the **Container Networking Interface** (**CNI**), which is similar to the CRI, but deals with the networking stack. You should begin to see a pattern emerge here.

CRI-O is an implementation that helps to implement the OCI specification. This allows users to take for granted the container runtime being used as an implementation detail, and to focus instead on how the application is interacting with the objects and abstractions of the Kubernetes system.

Trying out CRI-O

Let's look at some installation methods so you can give CRI-O a try on your own. In order to get started, you'll need a few things, including runc or another OCI compatible runtime, as well as socat, iproute, and iptables. There's a few options for running CRI-O in Kubernetes:

- In a full-scale cluster, using `kube-adm` and `systemd` to leverage the CRI-O socket with `--container-runtime-endpoint /var/run/crio/crio.sock`
- With Minikube, by starting it up with specific command-line options
- On atomic with atomic install `--system-package=no -n cri-o --storage ostree registry.centos.org/projectatomic/cri-o:latest`

If you'd like to build CRI-O from source, you can run the following on your laptop. You need some dependencies installed in order to make this build phase work. First, run the following commands to get your dependencies installed.

The following commands are for Fedora, CentOS, and RHEL distributions:

```
yum install -y \
  btrfs-progs-devel \
  device-mapper-devel \
  git \
  glib2-devel \
  glibc-devel \
  glibc-static \
  go \
  golang-github-cpuguy83-go-md2man \
  gpgme-devel \
  libassuan-devel \
  libgpg-error-devel \
  libseccomp-devel \
  libselinux-devel \
  ostree-devel \
  pkgconfig \
  runc \
  skopeo-containers
```

These commands are to be used for Debian, Ubuntu, and related distributions:

```
apt-get install -y \
  btrfs-tools \
  git \
  golang-go \
  libassuan-dev \
  libdevmapper-dev \
  libglib2.0-dev \
  libc6-dev \
  libgpgme11-dev \
  libgpg-error-dev \
  libseccomp-dev \
  libselinux1-dev \
  pkg-config \
  go-md2man \
  runc \
  skopeo-containers
```

Secondly, you'll need to grab the source code like so:

```
git clone https://github.com/kubernetes-incubator/cri-o # or your fork
cd cri-o
```

Once you have the code, go ahead and build it:

```
make install.tools
make
sudo make install
```

You can use additional build flags to add thing such as `seccomp`, SELinux, and `apparmor` with this format: `make BUILDTAGS='seccomp apparmor'`.

You can run Kubernetes locally with the `local-up-cluster.sh` script in Kubernetes. I'll also show you how to run this on Minikube.

First, clone the Kubernetes repository:

```
git clone https://github.com/kubernetes/kubernetes.git
```

Next, you'll need to start the CRI-O daemon and run the following command to get spin up your cluster using CRI-O:

```
CGROUP_DRIVER=systemd \
 CONTAINER_RUNTIME=remote \
 CONTAINER_RUNTIME_ENDPOINT='unix:///var/run/crio/crio.sock  --runtime-
request-timeout=15m' \
 ./hack/local-up-cluster.sh
```

If you have a running cluster, you can also use the instructions, available at the following URL, to switch the runtime from Docker to CRI-O: `https://github.com/kubernetes-incubator/cri-o/blob/master/kubernetes.md/`.

Let's also check how to use CRI-O on Minikube, which is one of the easiest ways to get experimenting:

```
minikube start \
 --network-plugin=cni \
 --extra-config=kubelet.container-runtime=remote \
 --extra-config=kubelet.container-runtime-endpoint=/var/run/crio/crio.sock
\
 --extra-config=kubelet.image-service-endpoint=/var/run/crio/crio.sock \
 --bootstrapper=kubeadm
```

Lastly, we can use our GCP platform to spin up a cluster with CRI-O and start experimenting:

```
gcloud compute instances create cri-o \
  --machine-type n1-standard-2 \
  --image-family ubuntu-1610 \
  --image-project ubuntu-os-cloud
```

Let's use these machines to run through a quick tutorial. SSH into the machine using `gcloud compute ssh cri-o`.

Once you're on the server, we'll need to install the `cri-o`, `crioctl`, `cni`, and `runc` programs. Grab the `runc` binary first:

```
wget
https://github.com/opencontainers/runc/releases/download/v1.0.0-rc4/runc.am
d64
```

Set it executable and move it to your path as follows:

```
chmod +x runc.amd64
sudo mv runc.amd64 /usr/bin/runc
```

You can see it's working by checking the version:

```
$ runc -version
runc version 1.0.0-rc4
commit: 2e7cfe036e2c6dc51ccca6eb7fa3ee6b63976dcd
spec: 1.0.0
```

You'll need to install the CRI-O binary from source, as it's not currently shipping any binaries.

First, download the latest binary release and install Go:

```
wget https://storage.googleapis.com/golang/go1.8.5.linux-amd64.tar.gz
sudo tar -xvf go1.8.5.linux-amd64.tar.gz -C /usr/local/
mkdir -p $HOME/go/src
export GOPATH=$HOME/go
export PATH=$PATH:/usr/local/go/bin:$GOPATH/bin
```

This should feel familiar, as you would install Go the same way for any other project. Check your version:

```
go version
go version go1.8.5 linux/amd64
```

Next up, get `crictl` using the following commands:

```
go get github.com/kubernetes-incubator/cri-tools/cmd/crictl
cd $GOPATH/src/github.com/kubernetes-incubator/cri-tools
make
make install
```

After that's downloaded, you'll need to build CRI-O from source:

```
sudo apt-get update && apt-get install -y libglib2.0-dev \
 libseccomp-dev \
 libgpgme11-dev \
 libdevmapper-dev \
 make \
 git
```

Now, get CRI-O and install it:

```
go get -d github.com/kubernetes-incubator/cri-o
cd $GOPATH/src/github.com/kubernetes-incubator/cri-o
make install.tools
Make
sudo make install
```

After this is complete, you'll need to create configuration files with `sudo make install.config`. You need to ensure that you're using a valid registry option in the `/etc/crio/cirio.conf` file. An example of this looks like the following:

```
registries = ['registry.access..com', 'registry.fedoraproject.org',
'docker.io']
```

At this point, we're ready to start the CRI-O system daemon, which we can do by leveraging `systemctl`. Let's create a `crio.service`:

```
$ vim /etc/systemd/system/crio.service
```

Add the following text:

```
[Unit]
Description=OCI-based implementation of Kubernetes Container Runtime
Interface
Documentation=https://github.com/kubernetes-incubator/cri-o

[Service]
ExecStart=/usr/local/bin/crio
Restart=on-failure
RestartSec=5

[Install]
WantedBy=multi-user.target
```

Once that's complete, we can reload `systemctl` and enable CRI-O:

```
$ sudo systemctl daemon-reload && \
 sudo systemctl enable crio && \
 sudo systemctl start crio
```

After this is complete, we can validate whether or not we have a working install of CRI-O by checking the version of the endpoint as follows:

```
$ sudo crictl --runtime-endpoint unix:///var/run/crio/crio.sock version
Version:  0.1.0
RuntimeName:  cri-o
RuntimeVersion:  1.10.0-dev
RuntimeApiVersion:  v1alpha1
```

Next up, we'll need to grab the latest version of the CNI plugin, so we can build and use it from source. Let's use Go to grab our source code:

```
go get -d github.com/containernetworking/plugins
cd $GOPATH/src/github.com/containernetworking/plugins
./build.sh
```

Next, install the CNI plugins into your cluster:

```
sudo mkdir -p /opt/cni/bin
sudo cp bin/* /opt/cni/bin/
```

Now, we can configure the CNI so that CRI-O can use it. First, make a directory to store the configuration, then we'll set two configuration files as follows:

```
sudo mkdir -p /etc/cni/net.d
```

Next, you'll want to create and compose `10-mynet.conf`:

```
sudo sh -c 'cat >/etc/cni/net.d/10-mynet.conf <<-EOF
{
"cniVersion": "0.2.0",
   "name": "mynet",
   "type": "bridge",
   "bridge": "cni0",
   "isGateway": true,
   "ipMasq": true,
   "ipam": {
       "type": "host-local",
       "subnet": "10.88.0.0/16",
       "routes": [
           { "dst": "0.0.0.0/0"  }
       ]
   }
}
EOF'
```

And then, compose the `loopback` interface as follows:

```
sudo sh -c 'cat >/etc/cni/net.d/99-loopback.conf <<-EOF
{
   "cniVersion": "0.2.0",
   "type": "loopback"
}
EOF'
```

Next up, we'll need some special containers from Project Atomic to get this working. `skopeo` is a command-line utility that is OCI-compliant and can perform various operations on container images and image repositories. Install the containers as follows:

```
sudo add-apt-repository ppa:projectatomic/ppa
sudo apt-get update
sudo apt-get install skopeo-containers -y
```

Restart CRI-O to pick up the CNI configuration with `sudo systemctl restart crio`. Great! Now that we have these components installed, let's build something!

First off, we'll create a sandbox using a template policy from the Kubernetes incubator.

This template is NOT production ready!

Change first to the CRI-O source tree with the template, as follows:

```
cd $GOPATH/src/github.com/kubernetes-incubator/cri-o
```

Next, you'll need to create and capture the pod ID:

```
sudo mkdir /etc/containers/
sudo cp test/policy.json /etc/containers
```

You can use `critcl` to get the status of the pod as follows:

```
sudo crictl inspectp --output table $POD_ID
ID: cd6c0883663c6f4f99697aaa15af8219e351e03696bd866bc3ac055ef289702a
Name: podsandbox1
UID: redhat-test-crio
Namespace: redhat.test.crio
Attempt: 1
Status: SANDBOX_READY
Created: 2016-12-14 15:59:04.373680832 +0000 UTC
Network namespace: /var/run/netns/cni-bc37b858-fb4d-41e6-58b0-9905d0ba23f8
IP Address: 10.88.0.2
Labels:
group -> test
Annotations:
owner -> jwhite
security.alpha.kubernetes.io/seccomp/pod -> unconfined
security.alpha.kubernetes.io/sysctls ->
kernel.shm_rmid_forced=1,net.ipv4.ip_local_port_range=1024 65000
security.alpha.kubernetes.io/unsafe-sysctls -> kernel.msgmax=8192
```

We'll use the `crictl` tool again to pull a container image for a Redis server:

```
sudo crictl pull quay.io/crio/redis:alpine
CONTAINER_ID=$(sudo crictl create $POD_ID
test/testdata/container_redis.json test/testdata/sandbox_config.json)
```

Next, we'll start and check the status of the Redis container as follows:

```
sudo crictl start $CONTAINER_ID
sudo crictl inspect $CONTAINER_ID
```

At this point, you should be able to `telnet` into the Redis container to test its functionality:

```
telnet 10.88.0.2 6379
Trying 10.88.0.2...
Connected to 10.88.0.2.
Escape character is '^]'.
```

Nicely done—you've now created a pod and container manually, using some of the core abstractions of the Kubernetes system! You can stop the container and shut down the pod with the following commands:

```
sudo crictl stop $CONTAINER_ID
sudo crictl rm $CONTAINER_ID
sudo crictl stopp $POD_ID
sudo crictl rmp $POD_ID
sudo crictl pods
sudo crictl ps
```

More on container runtimes

There's a number of container- and VM-based options for OCI-compliant implementations. We know of runc, which is the standard reference implementation of the OCI runtime. This is what the container uses. There's also the following available:

- `projectatomic/bwrap-oci` (`https://github.com/projectatomic/bwrap-oci`): Converts the OCI spec file to a command line for `projectatomic/bubblewrap` (`https://github.com/projectatomic/bubblewrap`)
- `giuseppe/crun` (`https://github.com/giuseppe/crun`): Runtime implementation in C

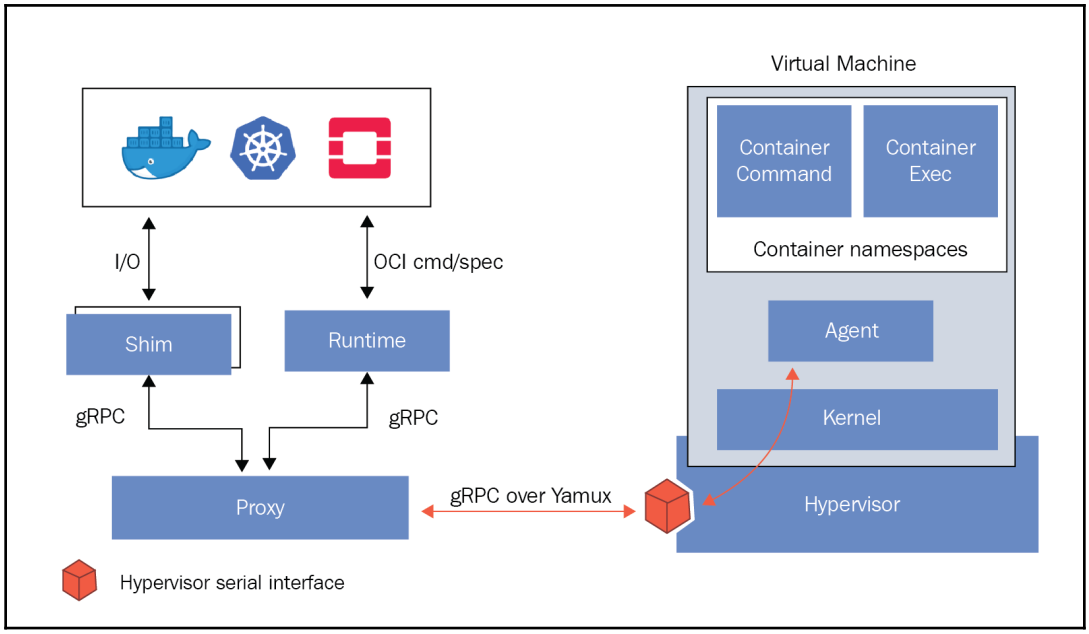
There are also VM-based implementations that take a different path towards security:

- `hyperhq/runv` (`https://github.com/hyperhq/runv`)—hypervisor-based runtime for OCI
- `clearcontainers/runtime` (`https://github.com/clearcontainers/runtime`)—hypervisor-based OCI runtime utilizing `containers/virtcontainers` (`https://github.com/containers/virtcontainers`) by Intel
- `google/gvisor` (`https://github.com/google/gvisor`)—gVisor is a user-space kernel, which contains runsc to run sandboxed containers

- `kata-containers/runtime` (`https://github.com/kata-containers/runtime`)—hypervisor-based OCI runtime combining technology from `clearcontainers/runtime` (`https://github.com/clearcontainers/runtime`) and `hyperhq/runv` (`https://github.com/hyperhq/runv`)

The most interesting project of these is the last in the list, Kata containers, which combines clear container and runV into a cohesive package. These foundational pieces are already in production use at scale in the enterprises, and Kata is looking to provide a secure, lightweight VM for containerized environments. By utilizing runV, Kata containers can run inside of any KVM-compatible VM, such as Xen, KVM, and vSphere, while still remaining compatible with CRI-O, which is important! Kata hopes to offer the speed of a container with the security surface of a VM.

Here's a diagram from Kata's site, explaining the architecture in visual detail:



CNCF

A second initiative that also has widespread industry acceptance is the CNCF. While still focused on containerized workloads, the CNCF operates a bit higher up the stack, at the application design level.

Its purpose is to provide a standard set of tools and technologies to build, operate, and orchestrate cloud-native application stacks. Cloud has given us access to a variety of new technologies and practices that can improve and evolve our classic software designs. The CNCF is also particularly focused on the new paradigm of microservice-oriented development.

As a founding participant in the CNCF, Google has donated the Kubernetes open source project. The goal will be to increase interoperability in the ecosystem and support better integration with projects. The CNCF already hosts a variety of projects on orchestration, logging, monitoring, tracing, and application resilience.

For more information on CNCF, refer to `https://cncf.io/`.

We'll talk more about the CNCF, **Special Interest Groups** (**SIGs**), and the landscape therein in the following chapters.

For now, here's a landscape and trail map to consider: `https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/`.

Standard container specification

A core result of the OCI effort is the creation and development of the overarching container specification. The specification has five core principles that all containers should follow, which I will briefly paraphrase:

- The container must have *standard operations* to create, start, and stop containers across all implementations.
- The container must be *content-agnostic*, which means that type of application inside the container does not alter the standard operations or publishing of the container itself.
- The container must be *infrastructure-agnostic* as well. Portability is paramount; therefore, the container must be able to operate just as easily in GCE as in your company's data center or on a developer's laptop.

- A container must also be *designed for automation*, which allows us to automate across the build, as well as for updates and the deployment pipelines. While this rule is a bit vague, the container implementation should not require onerous manual steps for creation and release.
- Finally, the implementation must support *industrial-grade delivery*. Once again, this means speaking to the build and deployment pipelines and requiring streamlined efficiency in the portability and transit of the containers between infrastructure and deployment tiers.

The specification also defines core principles for container formats and runtimes. You can read more about the specifications on the open containers GitHub page at `https://github.com/opencontainers/specs`.

While the core specification can be a bit abstract, the runc implementation is a concrete example of the OCI specs, in the form of a container runtime and image format. Again, you can read more of the technical details on GitHub at `https://github.com/opencontainers/runc`.

The backing format and runtime for a variety of popular container tools is runc. It was donated to OCI by Docker and was created from the same plumbing work used in the Docker platform. Since its release, it has received a welcome uptake by numerous projects.

Even the popular open source PaaS Cloud Foundry announced that it will use runc in Garden. Garden provides the containerization plumbing for Diego, which acts as an orchestration layer similar to Kubernetes.

The rkt implementation was originally based on the appc specification. The appc specification was actually an earlier attempt by the folks at CoreOS to form a common specification around containerization. Now that CoreOS is participating in OCI, they are working to help merge the appc specification into OCI; this should result in a higher level of compatibility across the container ecosystem.

CoreOS

While the specifications provide us with a common ground, there are also some trends evolving around the choice of OS for our containers. There are several tailored-fit OSes that are being developed specifically to run container workloads. Although implementations vary, they all have similar characteristics. The focus is on a slim installation base, atomic OS updating, and signed applications for efficient and secure operations.

One OS that is gaining popularity is CoreOS. CoreOS offers major benefits for both security and resource utilization. It provides resource utilization by completely removing package dependencies from the picture. Instead, CoreOS runs all applications and services in containers. By providing only a small set of services required to support running containers and bypassing the need for hypervisor usage, CoreOS lets us use a larger portion of the resource pool to run our containerized applications. This allows users to gain higher performance from their infrastructure and better container-to-node (server) usage ratios.

Recently, CoreOS was purchased by Red Hat, which means that the current version of container Linux will evolve against Red Hat's container OS offering, Project Atomic. These two products will eventually turn into Red Hat CoreOS. If you consider the upstream community approach that Fedora takes to Red Hat Enterprise Linux, it seems likely that there will be something similar for Red Hat CoreOS.

This also means that Red Hat will be integration Tectonic, which we'll explore later in the chapter, and the Quay, the enterprise container registry that CoreOS acquired. It's important to note that the rkt container standard will not be part of the acquisition, and will instead become a community supported project.

If you'd like to see the relevant official announcements for the news discussed in the preceding section, you can check out these posts:

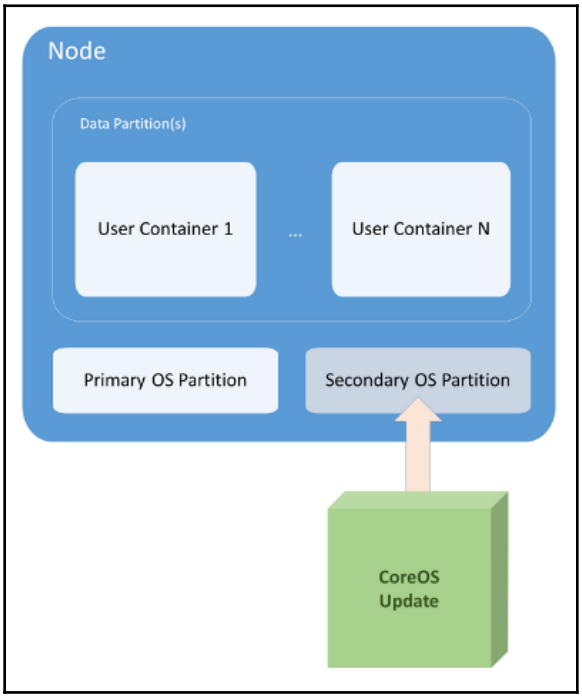
- **Press release**: `https://www.redhat.com/en/about/press-releases/red-hat-acquire-coreos-expanding-its-kubernetes-and-containers-leadership`
- **Red Hat blog**: `https://www.redhat.com/en/blog/coreos-bet`
- **CoreOS blog**: `https://coreos.com/blog/coreos-agrees-to-join-red-hat/`

Here's a brief overview of the various container OSes. There are several other container-optimized OSes that have emerged recently:

- *Red Hat Enterprise Linux Atomic Host* focuses on security with SELinux enabled by default and atomic updates to the OS similar to what we saw with CoreOS. Refer to the following link: `https://access.redhat.com/articles/rhel-atomic-getting-started`.
- *Ubuntu Snappy* also capitalizes on the efficiency and security gains of separating the OS components from the frameworks and applications. Using application images and verification signatures, we get an efficient Ubuntu-based OS for our container workloads at `http://www.ubuntu.com/cloud/tools/snappy`.

- *Ubuntu LXD* runs a container hypervisor and provides a path for migrating Linux-based VMs to containers with ease: `https://www.ubuntu.com/cloud/lxd`.
- *VMware Photon* is another lightweight container OS that is optimized specifically for vSphere and the VMware platform. It runs Docker, rkt, and Garden and also has some images that you can run on the popular public cloud providers. Refer to the following link: `https://vmware.github.io/photon/`.

Using the isolated nature of containers, we increase reliability and decrease the complexity of updates for each application. Now, applications can be updated along with supporting libraries whenever a new container release is ready, as shown in the following diagram:



CoreOS update procedure

Finally, CoreOS has some added advantages in the realm of security. For starters, the OS can be updated as one whole unit, instead of via individual packages (refer to the preceding diagram). This avoids many issues that arise from partial updates. To achieve this, CoreOS uses two partitions: one as the active OS partition, and a secondary one to receive a full update. Once updates are completed successfully, a reboot promotes the secondary partition. If anything goes wrong, the original partition is available as a fallback.

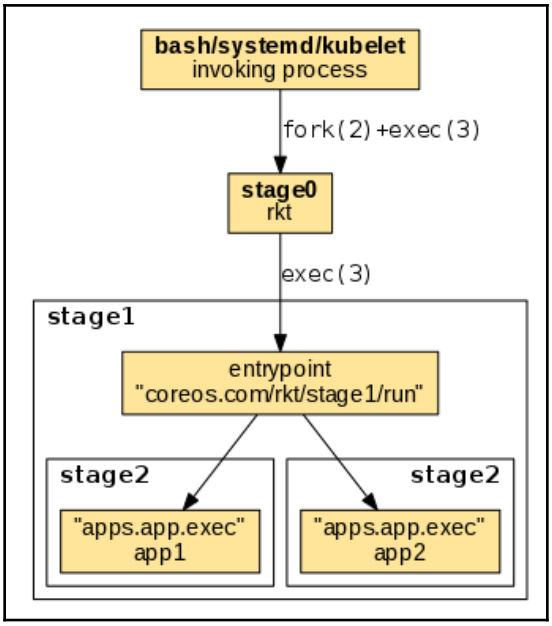
The system owners can also control when those updates are applied. This gives us the flexibility to prioritize critical updates, while working with real-world scheduling for the more common updates. In addition, the entire update is signed and transmitted via SSL for added security across the entire process.

rkt

As mentioned previously, rkt will be continuing on as a community driven project. rkt is another implementation with a specific focus on security. The main advantage of rkt is that it runs the engine without a daemon as root, the way Docker does today. Initially, rkt also had an advantage in the establishment of trust for container images. However, recent updates to Docker have made great strides, especially the new content trust feature.

The bottom line is that rkt is still an implementation, with a focus on security, for running containers in production. rkt uses an image format named ACI, but it also supports Docker-based images. Over the past year, rkt has undergone significant updates and is now at version 1.24.0. It has gained much momentum as a means to run Docker images securely in production.

Here's a diagram showing how the rkt execution chain works:



In addition, CoreOS is working with Intel® to integrate the new Intel® Virtualization Technology, which allows containers to run in higher levels of isolation. This hardware-enhanced security allows the containers to be run inside a **Kernel-based Virtual Machine** (**KVM**) process, providing isolation from the kernel in a similar fashion to what we see with hypervisors today.

etcd

Another central piece in the CoreOS ecosystem worth mentioning is their open source etcd project. etcd is a distributed and consistent key-value store. A RESTful API is used to interface with etcd, so it's easy to integrate with your project.

If it sounds familiar, it's because we saw this process running in `Chapter 1`, *Introduction to Kubernetes*, in the section entitled *Services running on the master*. Kubernetes actually utilizes etcd to keep track of cluster configuration and current state. K8s uses it for its service discovery capabilities as well. For more details, refer to `https://github.com/coreos/etcd`.

Kubernetes with CoreOS

Now that we understand the benefits, let's take a look at a Kubernetes cluster using CoreOS. The documentation supports a number of platforms, but one of the easiest to spin up is AWS with the CoreOS CloudFormation and CLI scripts.

If you are interested in running Kubernetes with CoreOS on other platforms, you can find more details in the CoreOS documentation at `https://coreos.com/kubernetes/docs/latest/`. You can find the latest instructions for AWS at `https://coreos.com/kubernetes/docs/latest/kubernetes-on-aws.html`.

You can follow the instructions covered previously in this chapter to spin up Kubernetes on CoreOS. You'll need to create a key pair on AWS, and also specify a region, cluster name, cluster size, and DNS to proceed.

In addition, we will need to create a DNS entry, and will require a service such as Route 53 or a production DNS service. When following the instructions, you'll want to set the DNS to a domain or sub-domain on which you have permission to set up a record. We will need to update the record after the cluster is up and running and has a dynamic endpoint defined.

There you have it! We now have a cluster running CoreOS. The script creates all the necessary AWS resources, such as **Virtual Private Clouds** (**VPCs**), security groups, and IAM roles. Now that the cluster is up and running, we can get the endpoint with the `status` command and update our DNS record as follows:

```
$ kube-aws status
```

Copy the entry listed next to `Controller DNS Name` in the output from the preceding command, and then edit your DNS records to get the domain or sub-domain you specified earlier to point to this load balancer.

If you forget which domain you specified or need to check on the configuration, you can look in the generated `kubeconfig` file with your favorite editor. It will look something like this:

```
apiVersion: v1
kind: Config
clusters:
- cluster:
    certificate-authority: credentials/ca.pem
    server: https://coreos.mydomain.com
  name: kube-aws-my-coreos-cluster-cluster
contexts:
- context:
    cluster: kube-aws-my-coreos-cluster-cluster
    namespace: default
    user: kube-aws-my-coreos-cluster-admin
  name: kube-aws-my-coreos-cluster-context
users:
- name: kube-aws-my-coreos-cluster-admin
  user:
    client-certificate: credentials/admin.pem
    client-key: credentials/admin-key.pem
current-context: kube-aws-my-coreos-cluster-context
```

In this case, the `server` line will have your domain name.

If this is a fresh box, you will need to download `kubectl` separately, as it is not bundled with `kube-aws`:

```
$ wget
https://storage.googleapis.com/kubernetes-release/release
/v1.0.6/bin/linux/amd64/kubectl
```

We can now use `kubectl` to see our new cluster:

```
$ ./kubectl --kubeconfig=kubeconfig get nodes
```

We should see a single node listed with the EC2 internal DNS as the name. Note `kubeconfig`, this tells Kubernetes the path to use the configuration file for the cluster that was just created instead. This is also useful if we want to manage multiple clusters from the same machine.

Tectonic

Running Kubernetes on CoreOS is a great start, but you may find that you want a higher level of support. Enter Tectonic, the CoreOS enterprise offering for running Kubernetes with CoreOS. Tectonic uses many of the components we already discussed. Both Docker and rkt runtimes are supported. In addition, Kubernetes, etcd, and flannel are packaged together to give a full stack of cluster orchestration. We discussed flannel briefly in `Chapter 3`, *Working with Networking, Load Balancers, and Ingress*. It is an overlay network that uses a model similar to the native Kubernetes model, and uses etcd as a backend.

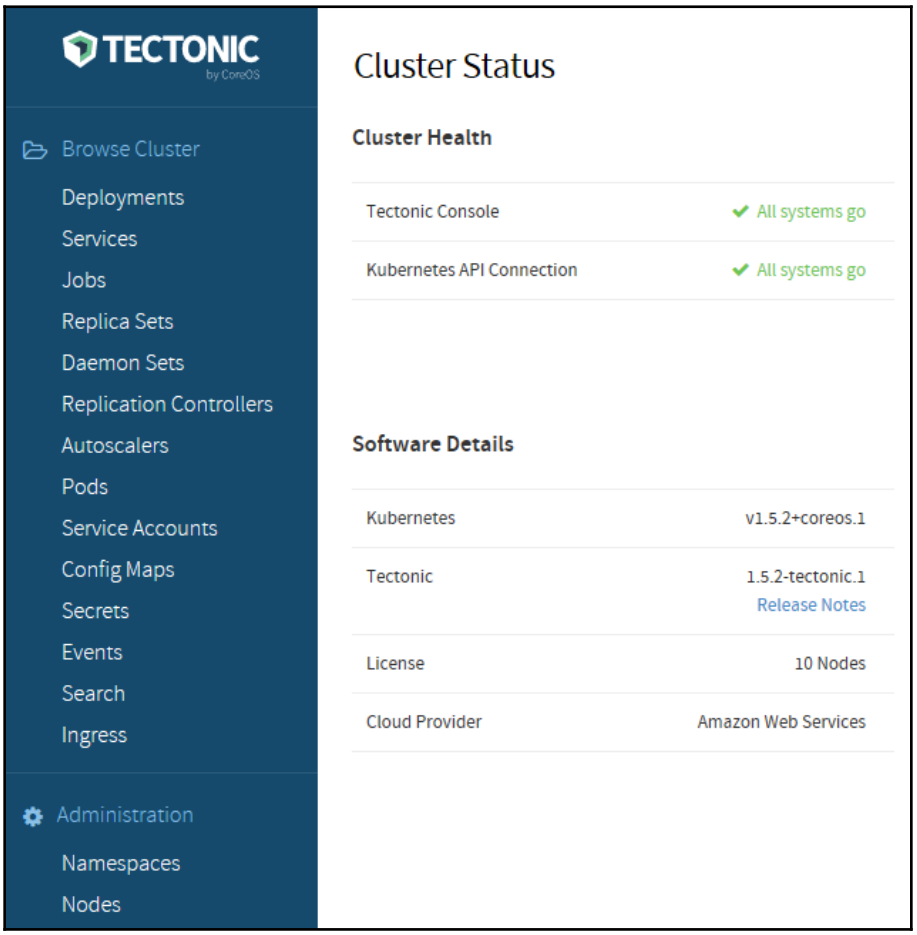
Offering a support package similar to Red Hat, CoreOS also provides 24/7 support for the open source software that Tectonic is built on. Tectonic also provides regular cluster updates and a nice dashboard with views for all of the components of Kubernetes. **CoreUpdate** allows users to have more control of the automatic update process. In addition, it ships with modules for monitoring, SSO, and other security features.

As CoreOS is integrated into Red Hat, this offering will be replaced over time with a Red Hat approach.

You can find more information and the latest instructions to install at `https://coreos.com/tectonic/docs/latest/install/aws/index.html`.

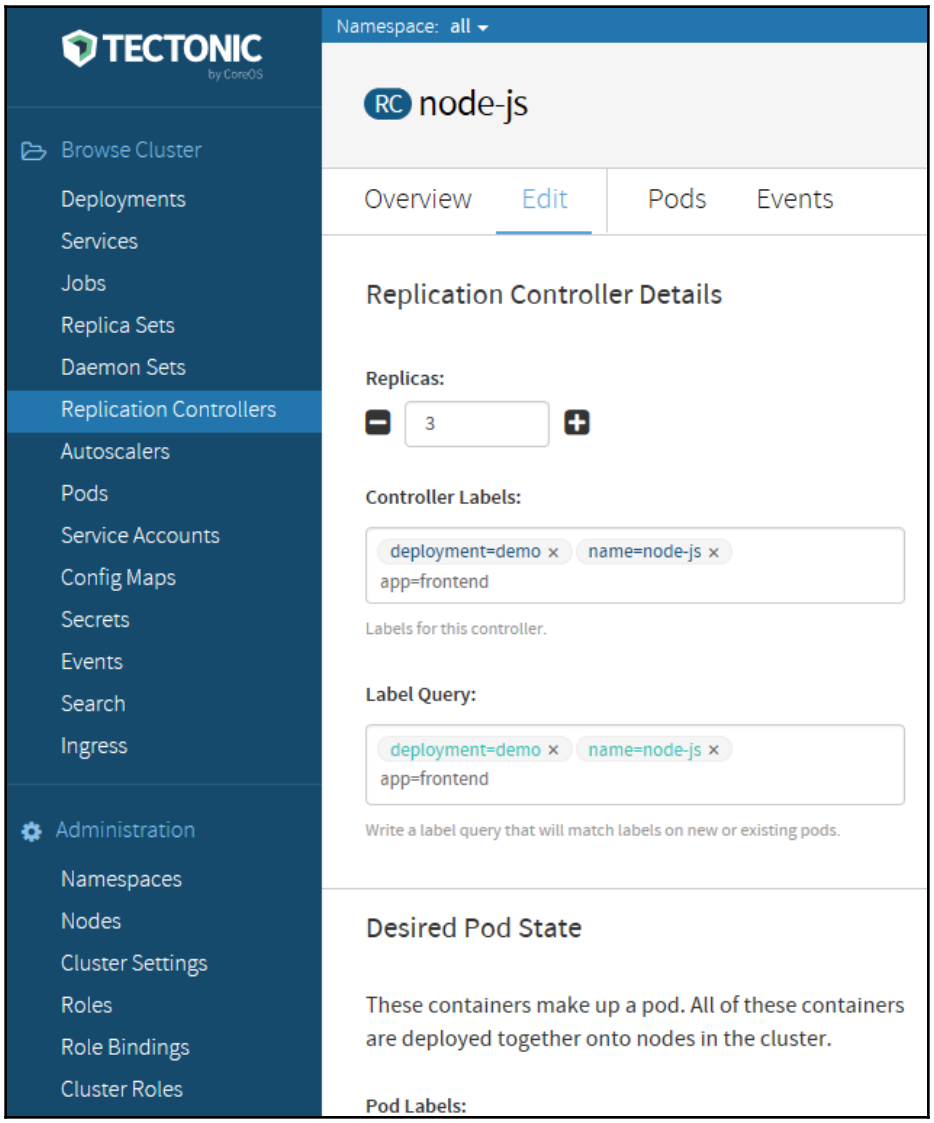
Dashboard highlights

Some highlights of the Tectonic dashboard are shown in the following screenshot:



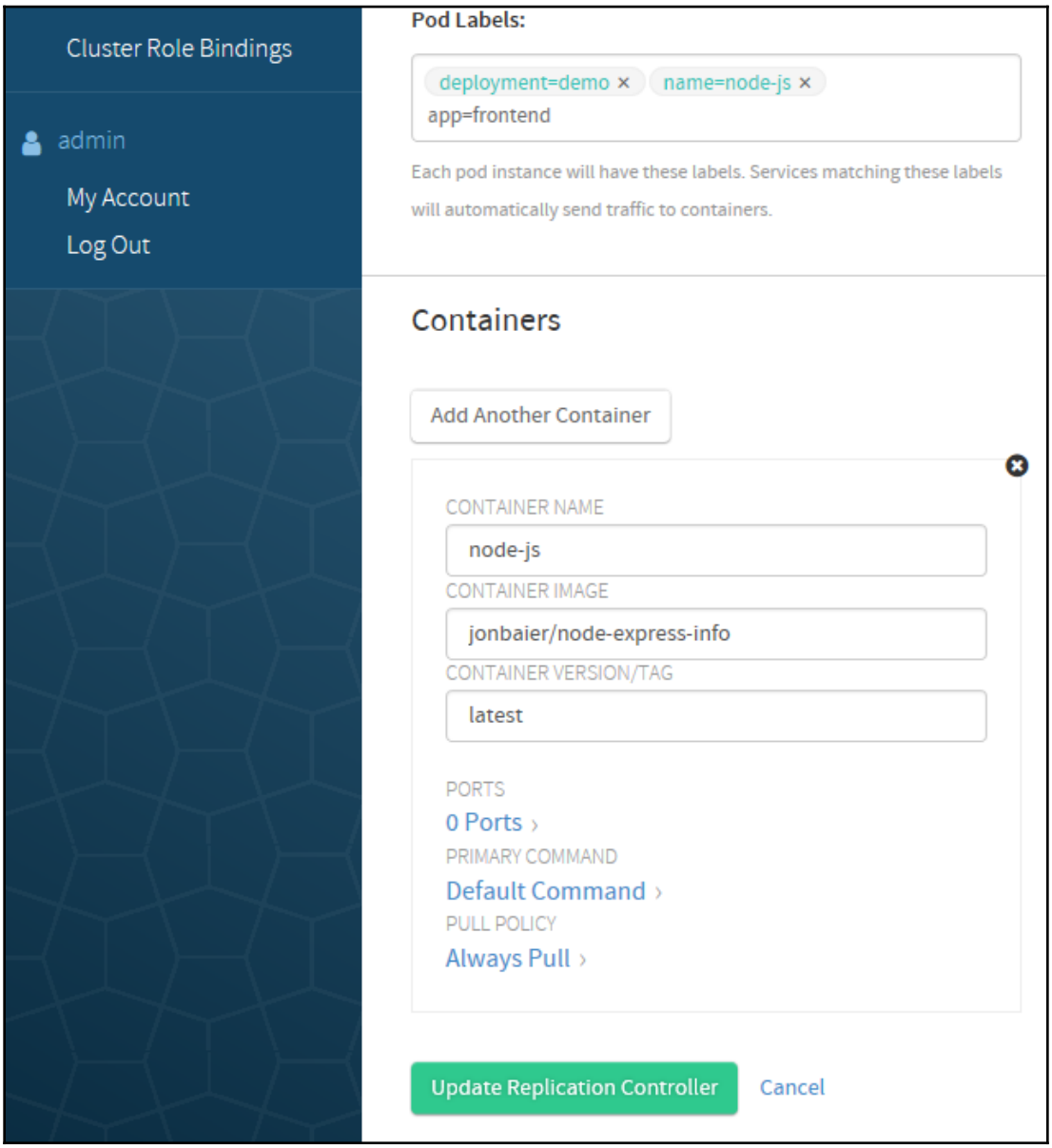
The Tectonic main dashboard

Tectonic is now generally available and the dashboard already has some nice features. As you can see in the following screenshot, we can see a lot of detail about our replication controller, and can even use the GUI to scale up and down with the click of a button:

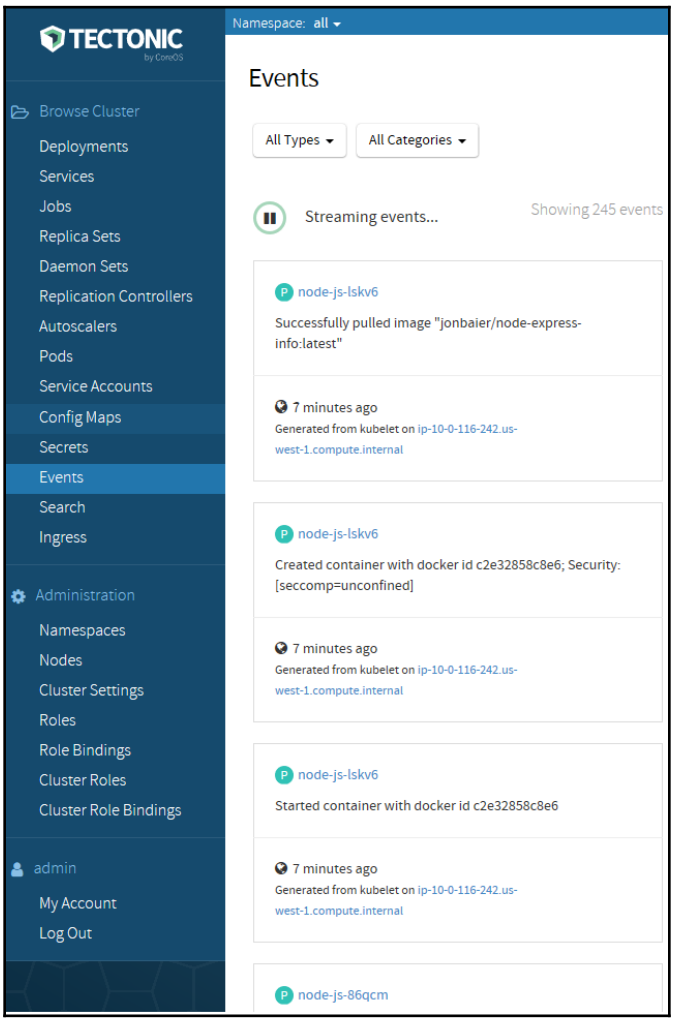


Tectonic replication controller detail

This graphic is quite large, so it's broken across two pages. The following screenshot continues from the preceding screenshot:

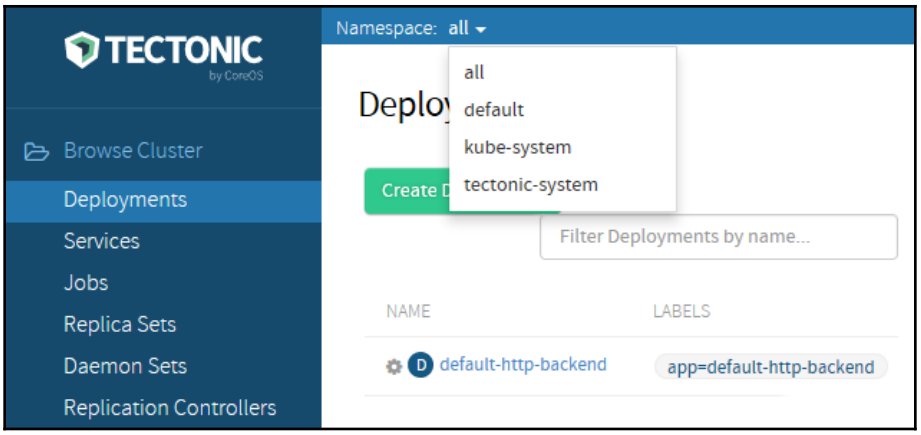


Another nice feature is the **Events** page. Here, we can watch the events live, pause them, and filter them based on event severity and resource type:



Events stream

A useful feature to browse anywhere in the dashboard system is the **Namespace:** filtering option. Simply click on the drop-down menu next to the word **Namespace:** at the top of any page that shows resources, and we can filter our views by namespace. This can be helpful if we want to filter out the Kubernetes system pods, or just look at a particular collection of resources:



Namespace filtering

Hosted platforms

There are several options available for hosted Kubernetes in the cloud. These **Platforms as a service** (**PaaS**) can provide a stable operating model as you push towards production. Here's an overview of the major PaaSes provided by Amazon, Microsoft, and Google.

Amazon Web Services

Elastic Container Service (**ECS**) has just been launched as of the time of this chapter's writing. AWS is preparing a networking plugin to differentiate itself from other offerings, called the vpc-cni. This allows for pod networking in Kubernetes to use **Elastic Network Interfaces** (**ENIs**) on AWS. With ECS, you do have to pay for manager nodes, which is a different path to that taken by Microsoft and Google. ECS' startup procedure is also currently more complex and doesn't have single-command creation via the CLI.

Microsoft Azure

The Azure Container Service is the second longest running hosted Kubernetes service in the cloud after the Google Kubernetes Engine. You can use Azure templates and the Resource Manager to spin up clusters with Terraform. Microsoft offers advanced networking features, integration with Azure Active Directory, and monitoring as its standout features.

Google Kubernetes Engine

The Google Kubernetes Engine is another excellent option for running your containerized workloads. At the time of writing, it's considered to be one of the most robust offerings. GKE is able to autoscale the cluster size, while AWS and Azure offer manual scaling. GKE offers a one-command start, and is the fastest to provision a Kubernetes cluster. It also offers an *Alpha Mode* where you can try bleeding edge features in the alpha channel releases. GKE provides high availability in zones and regions, the latter of which spreads out master node zones to provide best-in-class high availability.

Summary

In this chapter, we looked at the emerging standards bodies in the container community and how they are using open specifications to shape the technology for the better. We looked at various container frameworks and runtimes. We dipped our toes into the CNCF, and tried out CRI-O.

We also took a closer look at CoreOS, a key player in both the container and Kubernetes community. We explored the technology that CoreOS is developing in order to enhance and complement container orchestration, and saw first-hand how to use some of it with Kubernetes. Finally, we looked at the supported enterprise offering of Tectonic and some of the features that are available now.

We also looked at some of the major PaaS offered by cloud service providers.

In the next chapter, we will explore the broader Kubernetes ecosystem and the tools available to move your cluster from development and testing into full-blown production.

Further reading

- `https://www.opencontainers.org/faq/` (under **How broad is the mission of the OCI?**)
- `https://github.com/opencontainers/specs/blob/master/principles.md`

10
Designing for High Availability and Scalability

This chapter will cover advanced concepts such as high availability, scalability, and the requirements that Kubernetes operators will need to cover in order to begin to explore the topic of running Kubernetes in production. We'll take a look at the **Platform as a Service** (**PaaS**) offerings from Google and Azure and we'll use the familiar principles of running production workloads in a cloud environment.

We'll cover the following topics in this chapter:

- Introduction to high availability
- High availability best practices
- Multi-region setups
- Security best practices
- Setting up high availability on the hosted Kubernetes PaaS
- Cluster life cycle events
- How to use admission controllers
- Getting involved with the workloads API
- What is a **custom resource definition** (**CRD**)?

Technical requirements

You'll need to have access to your Google Cloud Platform account in order to explore some of these options. You can also use a local Minikube setup to test some of these features, but many of the principles and approaches we'll discuss here require servers in the cloud.

Introduction to high availability

In order to understand our goals for this chapter, we first need to talk about the more general terms of high availability and scalability. Let's look at each individually to understand how the pieces work together.

We'll discuss the required terminology and begin to understand the building blocks that we'll use to conceptualize, construct, and run a Kubernetes cluster in the cloud.

Let's dig into high availability, uptime, and downtime.

How do we measure availability?

High availability (**HA**) is the idea that your application is available, meaning reachable, to your end users. In order to create *highly available* applications, your application code and the frontend that users interact with needs to be available the majority of the time. This term comes from the system design field, which defines the architecture, interface, data, and modules of a system in order to satisfy a given set of requirements. There are many examples of system design in disciplines from product development all the way to distributed systems theory. In HA, system design helps us understand the logical and physical design requirements to achieve a reliable and performant system.

In the industry, we refer to excellence in availability as five nines of availability. This *99.999* availability translates into specific amounts of downtime per day, week, month, and year.

If you'd like to read more about the math behind the five nine's availability equation, you can read about floor and ceiling functions here: `https://en.wikipedia.org/wiki/Floor_and_ceiling_functions`.

We can also look at the general availability formula, which you can use to understand a given system's availability:

```
Downtime per year in hours = (1 - Uptime Availability) x 365 x 24
```

Uptime and downtime

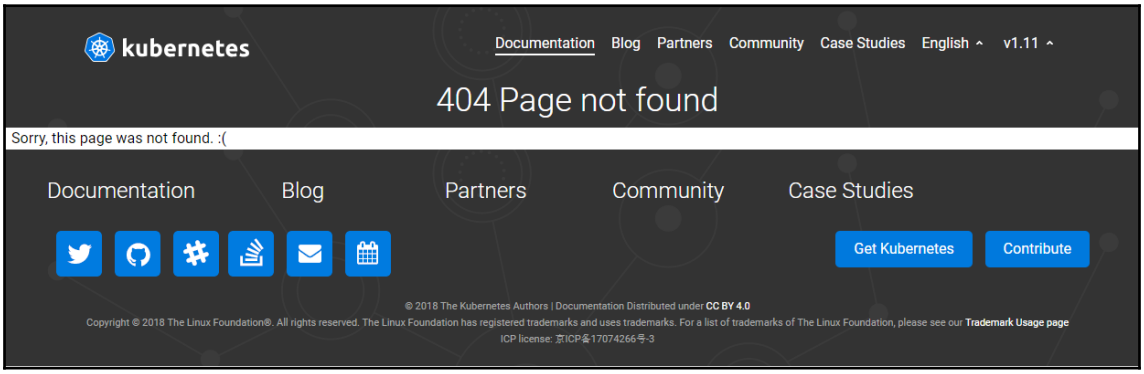
Let's dig into what it means to be up or down before we look at net availability over a daily, weekly, and yearly period. We should also establish a few key terms in order to understand what availability means to our business.

Uptime

Uptime is the measure of time a given system, application, network, or other logical and physical object has been up and available to be used by the appropriate end user. This can be an internally facing system, an external item, or something that's only interacted with via other computer systems.

Downtime

Downtime is similar to uptime, but measures the time in which a given system, application, network, or other logical and physical object is not available to the end user. Downtime is subject to some interpretation, as it's defined as a period where the system is not performing its primary function as originally intended. The most ubiquitous example of downtime is the infamous 404 page, which you may have seen before:



In order to understand the availability of your system with the preceding concepts, we can calculate using available uptime and downtime figures:

```
Availability Percentage = (Uptime / (Uptime + Downtime) x 100
```

There is a more complex calculation for systems that have redundant pieces that increase the overall stability of a system, but let's stick with our concrete example for now. We'll investigate the redundant pieces of Kubernetes later on in this chapter.

Given these equations, which you can use on your own in order to measure the uptime of your Kubernetes cluster, let's look at a few examples.

Let's look at some of the math behind these concepts. To get started, uptime availability is a function of **Mean Time Between Failures** (**MTBF**), divided by the sum of **Mean Time to Repair** (**MTTR**) and MTBF combined.

We can calculate MTBF as follows:

```
MTBF = 'Total hours in a year' / 'Number of yearly failures'
```

And MTTR is represented as follows:

```
MTTR = ('Amount of failure' x 'Time to repair the system') / 'Total number of
failures'
```

This is represented with the following formula:

```
Uptime Availability = MTBF/(MTTR + MTBF)
Downtime per Year (Hours) = (1 – Uptime Ratio) x 365 x 24
```

The five nines of availability

We can look more deeply at the industry standard of five nines of availability against fewer nines. We can use the term **Service Level Agreement** (**SLA**) to understand the contract between the end user and the Kubernetes operator that guarantees the availability of the underlying hardware and Kubernetes software to your application owners.

A SLA is a guaranteed level of availability. It's important to note that the availability gets very expensive as it increases.

Here are a few SLA levels:

- With an SLA of 99.9% availability, you can have a downtime of:
    - **Daily**: 1 minute, 26.4 seconds
    - **Weekly**: 10 minutes, 4.8 seconds
    - **Monthly**: 43 minutes, 49.7 seconds
    - **Yearly**: 8 hours 45 minutes, 57.0 seconds
- With an SLA of 99.99% availability, you can have a downtime of:
    - **Daily**: 8.6 seconds
    - **Weekly**: 1 minutes, 0.5 seconds
    - **Monthly**: 4 minutes, 23.0 seconds
    - **Yearly**: 52 minutes, 35.7 seconds
- With an SLA of 99.999% availability, you can have downtime of:
    - **Daily**: 0.9 seconds
    - **Weekly**: 6.0 seconds
    - **Monthly**: 26.3 seconds
    - **Yearly**: 5 minutes, 15.6 seconds

As you can see, with five nines of availability, you don't have a lot of room to breathe with your Kubernetes cluster. It's also important to note that the availability of your cluster is a function of the application's availability.

What does that mean? Well, the application itself will also have problems and code errors that are outside of the domain and control of the Kubernetes cluster. So, the uptime and availability of a given application is going to be equal to (and rarely if ever equal, given human error) or less than your cluster's general availability.

So, let's figure out the pieces of HA in Kubernetes.

HA best practices

In order to build HA Kubernetes systems, it's important to note that availability is as often a function of people and process as it is a failure in technology. While hardware and software fails often, humans and their involvement in the process is a very predictable drag on the availability of all systems.

It's important to note that this book won't get into how to design a microservices architecture for failure, which is a huge part of coping with some (or all) system failures in a cluster scheduling and networking system such as Kubernetes.

There's another important concept that's important to consider: graceful degradation.

Graceful degradation is the idea that you build functionality in layers and modules, so even with the catastrophic failure of some pieces of the system, you're still able to provide some level of availability. There is a corresponding term for the progressive enhancement that's followed in web design, but we won't be using that pattern here. Graceful degradation is an outcome of the condition of a system having fault tolerance, which is very desirable for mission critical and customer-facing systems.

In Kubernetes, there are two methods of graceful degradation:

- **Infrastructure degradation**: This kind of degradation relies on complex algorithms and software in order to handle unpredictable failure of hardware, or software-defined hardware (think virtual machines, **Software-Defined Networking** (**SDN**), and so on). We'll explore how to make the essential components of Kubernetes highly available in order to provide graceful degradation in this form.
- **Application degradation**: While this is largely determined by the aforementioned strategies of microservice best practice architectures, we'll explore several patterns here that will enable your users to be successful.

In each of these scenarios, we're aiming to provide as much full functionality as possible to the end user, but if we have a failure of application, Kubernetes components, or underlying infrastructure, the goal should be to give some level of access and availability to the users. We'll strive to abstract away completely underlying infrastructure failure using core Kubernetes strategies, while we'll build caching, failover, and rollback mechanisms in order to deal with application failure. Lastly, we'll build out Kubernetes components in a highly available fashion.

Anti-fragility

Before we dig into these items, it makes sense to step back and consider the larger concept of anti-fragility, which Nassim Nicholas Taleb discusses in his book *Antifragility*.

To read more about Taleb's book, check out his book's home page at `https://www.penguinrandomhouse.com/books/176227/antifragile-by-nassim-nicholas-taleb/9780812979688/`.

There are a number of key concepts that are important to reinforce as we cope with the complexity of the Kubernetes system, and in how we leverage the greater Kubernetes ecosystem in order to survive and strive.

First, redundancy is key. In order to cope with system failure across the many layers of a system, it's important to build redundant and failure tolerant parts into the system. These redundant layers can utilize algorithms such as Raft consensus, which aims to provide a control plane for multiple objects to agree in a fault-tolerant distributed system. Redundancy of this type relies on N+1 redundancy in order to cope with physical or logical object loss.

We'll take a look at etcd in a bit to explore redundancy.

Second, triggering, coping with, exploring, and remediating failure scenarios is key. You'll need to forcefully cause your Kubernetes system to fail in order to understand how it behaves at the limit, or in corner cases. Netflix's Chaos Monkey is a standard and well-worn approach to testing complex system reliability.

You can read more about Netflix's Chaos Monkey here: `https://github.com/Netflix/chaosmonkey`.

Third, we'll need to make sure that the correct patterns are available to our systems, and that we implement the correct patterns in order to build anti-fragility into Kubernetes. Retry logic, load balancing, circuit breakers, timeouts, health checks, and concurrent connection checks are key items for this dimension of anti-fragility. Istio and other service meshes are advanced players in this topic.

You can read more about Istio and how to manage traffic here: `https://istio.io/docs/concepts/traffic-management/`.

HA clusters

In order to create Kubernetes clusters which can fight against the patterns of anti-fragility and to increase the uptime of our cluster, we can create highly available clusters using the core components of the system. Let's explore the two main methods of setting up highly available Kubernetes clusters. Let's look at what you get from the major cloud service providers when you spin up a Kubernetes cluster with them first.

HA features of the major cloud service providers

What are the pieces of Kubernetes that need to be high availability in order to achieve the five nines of uptime for your infrastructure? For one, you should consider how much the **cloud service provider** (**CSP**) does for you on the backend.

For **Google Kubernetes Engine** (**GKE**), nearly all of the components are managed out of the box. You don't have to worry about the manager nodes or any cost associated with them. GKE also has the most robust autoscaling functionality currently. **Azure Kubernetes Service** (**AKS**) and Amazon **Elastic Kubernetes Service** (**EKS**) both use a self-managed autoscaling function, which means that you're in charge of managing the scale out of your cluster by using autoscaling groups.

GKE is also able to handle automatic updates to the management nodes without user intervention, but also offers a turnkey automatic update along with AKS so that the operator can choose when seamless upgrade happens. EKS is still working out those details.

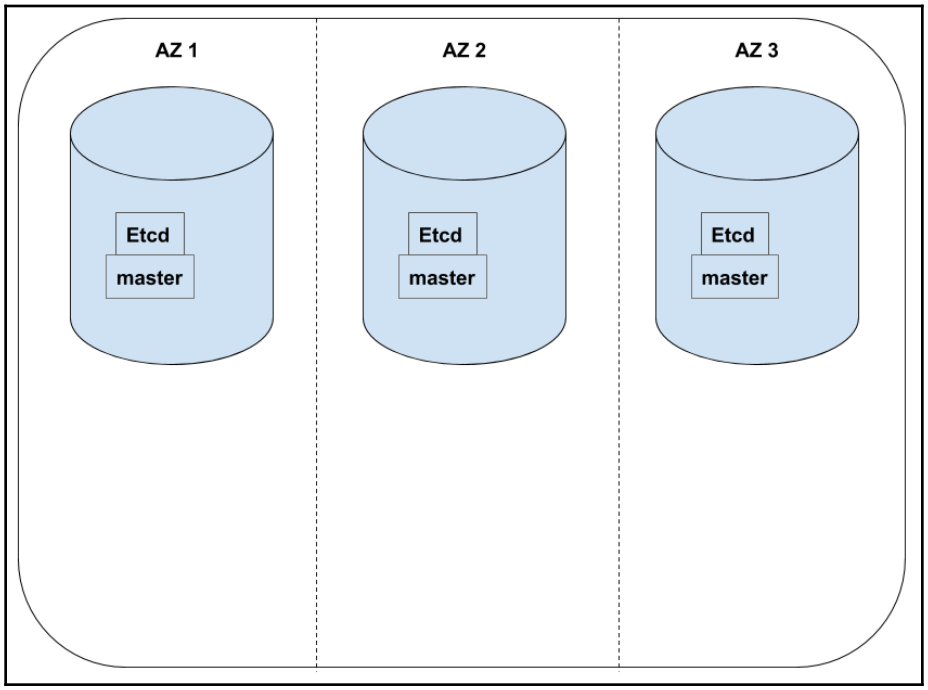
EKS provides highly available master/worker nodes across multiple **Availability Zones** (**AZ**), while GKE offers something similar in their regional mode, which is akin to AWS's regions. AKS currently does not provide HA for the master nodes, but the worker nodes in the cluster are spread across multiple AZ in order to provide HA.

HA approaches for Kubernetes

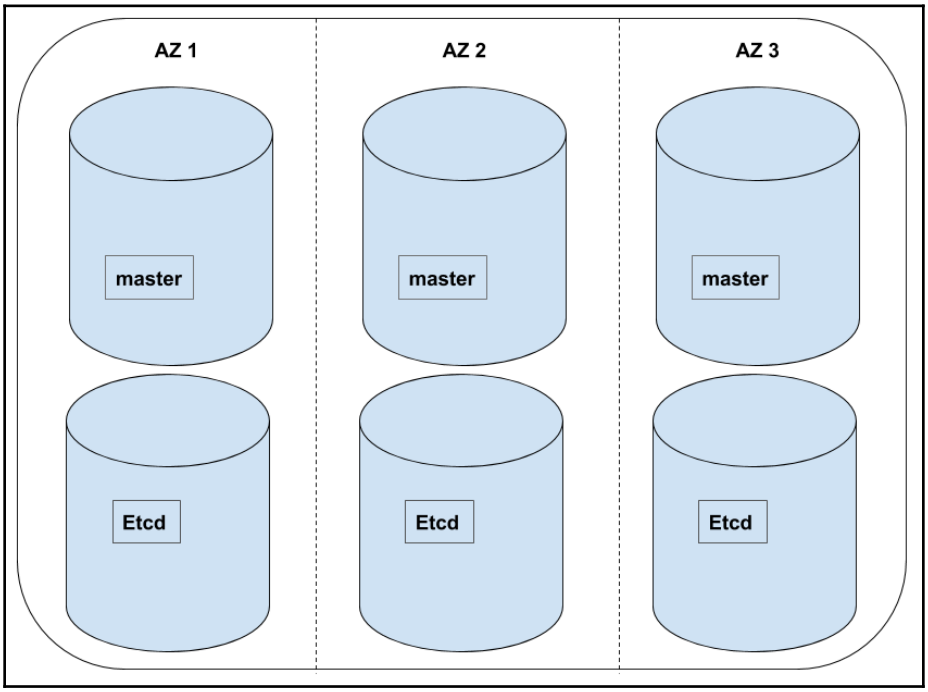
If you're going to be running Kubernetes outside of a hosted PaaS, you'll need to adopt one of two strategies for running an HA cluster for Kubernetes. In this chapter, we'll go through an example with Stacked masters, and will describe the more complex external etcd cluster method.

In this method, you'll combine etcd and manager (control plane) nodes in order to reduce the amount of infrastructure required to run your cluster. This means that you'll need at least three machines in order to achieve HA. If you're running in the cloud, that also means you'll need to spread your instances across three availability zones in order to take advantage of the uptime provided by spreading your machines across zones.

Stacked masters is going to look like this in your architectural diagrams:



The second option you have builds in more potential availability in exchange for infrastructure complexity. You can use an external etcd cluster in order to create separation for the control plane and the ectd members, further increasing your potential availability. A setup in this manner will require a bare minimum of six servers, also spread across availability zones, as in the first example:



In order to achieve either of these methods, you'll need some prerequisites.

Prerequisites

As mentioned in the preceding section, you'll need three machines for the masters, three machines for the workers, and an extra three machines for the external etcd cluster if you're going to go down that route.

Here are the minimum requirements for the machines – you should have one of the following operating systems:

- Ubuntu 16.04+
- Debian 9
- CentOS 7
- RHEL 7
- Fedora 25/26 (best-effort)
- Container Linux (tested with 1576.4.0)

On each of the machines, you'll need 2 GB or more of RAM per machine, two or more CPUs, and full network connectivity between all machines in the cluster (a public or private network is fine). You'll also need a unique hostname, MAC address, and a `product_uuid` for every node.

If you're running in a managed network of any sort (datacenter, cloud, or otherwise), you'll also need to ensure that the required security groups and ports are open on your machines. Lastly, you'll need to disable swap in order to get a working `kubelet`.

For a list of required open ports, check out `https://kubernetes.io/docs/setup/independent/install-kubeadm/#check-required-ports`.

In some cloud providers, virtual machines may share identical `product_uuids`, though it's unlikely that they'll share identical MAC addresses. It's important to check what these are, because Kubernetes networking and Calico will use these as unique identifiers, and we'll see errors if they're the same. You can check both with the following commands:

```
LANG=C ifconfig -a | grep -Po 'HWaddr \K.*$'
```

The preceding command will get you the MAC address, while the following command will tell you the `uuid`:

```
sudo cat /sys/class/dmi/id/product_uuid
```

Setting up

Now, let's start setting up the machines.

You'll need to run all of the commands here on a control plane node, and as root.

First, you'll need to set up SSH. Calico will be setting up your networking, so we'll use the IP address of your machine in order to get started with this process. Keep in mind that Kubernetes networking has three basic layers:

- The containers and pods that run on your nodes, which are either virtual machines or hardware servers.
- Services, which are an aggregation and abstraction layer that lets you use the various Kubernetes controllers to set up your applications and ensure that your pods are scheduled according to its availability needs.
- Ingress, which allows traffic from outside of your cluster and are routed to the right container.

So, we need to set up Calico in order to deal with these different layers. You'll need to get your node's CIDR address, which we recommend being installed as Calico for this example.

You can find more information on the CNI network documentation at `https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network`.

You'll need to make sure that the SSH agent on the configuration machine has access to all of the other nodes in the cluster. Turn on the agent, and then add our identity to the session:

```
eval $(ssh-agent)
ssh-add ~/.ssh/id_rsa
```

You can test to make sure that this is working correctly by using the `-A` flag, which preserves your identity across an SSH tunnel. Once you're on another node, you can use the `-E` flag to preserve the environment:

```
sudo -E -s
```

Next, we'll need to put a load balancer from our cloud environment in front of the `kube-apiserver`. This will allow your cluster's API server remain reachable in the case of one of the machines going down or becoming unresponsive. For this example, you should use a TCP capable load balancer such as an Elastic Load Balancer (AWS), Azure Load Balancer (Azure), or a TCP/UDP Load Balancer (GCE).

Make sure that your load balancer is resolvable via DNS, and that you set a health check that listens on the `kube-apiserver` port at `6443`. You can test the connection to the API server once the load balancer is in place with `nc -v LB_DNS_NAME PORT`. Once you have the cloud load balancer set up, make sure that all of the control plane nodes are added to it.

Stacked nodes

In order to run a set of stack nodes, you'll need to bootstrap the first control plane node with a `kubeadm-conf-01.yaml` template. Again, this example is using Calico, but you can configure the networking as you please. You'll need to substitute the following values with your own in order to make the example work:

- `LB_DNS`
- `LB_PORT`
- `CONTROL01_IP`
- `CONTROL01_HOSTNAME`

Open up a new file, `kubeadm-conf-01.yaml`, with your favorite IDE:

```
apiVersion: kubeadm.k8s.io/v1alpha2
kind: MasterConfiguration
kubernetesVersion: v1.11.0
apiServerCertSANs:
- "LB_DNS"
api:
   controlPlaneEndpoint: "LB_DNS:LB_PORT"
etcd:
 local:
   extraArgs:
     listen-client-urls: "https://127.0.0.1:2379,https://CONTROL01_IP:2379"
     advertise-client-urls: "https://CONTROL01_IP:2379"
     listen-peer-urls: "https://CONTROL01_IP:2380"
     initial-advertise-peer-urls: "https://CONTROL01_IP:2380"
     initial-cluster: "CONTROL01_HOSTNAME=https://CONTROL01_IP:2380"
   serverCertSANs:
     - CONTROL01_HOSTNAME
     - CONTROL01_IP
```

```
    peerCertSANs:
      - CONTROL01_HOSTNAME
      - CONTROL01_IP
networking:
    podSubnet: "192.168.0.0/16"
```

Once you have this file, execute it with the following command:

```
kubeadm init --config kubeadm-conf-01.yaml
```

Once this command is complete, you'll need to copy the following list of certificates and files to the other control plane nodes:

```
/etc/kubernetes/pki/ca.crt
/etc/kubernetes/pki/ca.key
/etc/kubernetes/pki/sa.key
/etc/kubernetes/pki/sa.pub
/etc/kubernetes/pki/front-proxy-ca.crt
/etc/kubernetes/pki/front-proxy-ca.key
/etc/kubernetes/pki/etcd/ca.crt
/etc/kubernetes/pki/etcd/ca.key
/etc/kubernetes/admin.conf
```

In order to move forward, we'll need to add another template file on our second node to create the second stacked node under `kubeadm-conf-02.yaml`. Like we did previously, you'll need to replace the following values with your own:

- `LB_DNS`
- `LB_PORT`
- `CONTROL02_IP`
- `CONTROL02_HOSTNAME`

Open up a new file, `kubeadm-conf-02.yaml`, with your favorite IDE:

```
apiVersion: kubeadm.k8s.io/v1alpha2
kind: MasterConfiguration
kubernetesVersion: v1.11.0
apiServerCertSANs:
- "LOAD_BALANCER_DNS"
api:
    controlPlaneEndpoint: "LB_DNS:LB_PORT"
etcd:
 local:
    extraArgs:
      listen-client-urls: "https://127.0.0.1:2379,https://CONTROL02_IP:2379"
      advertise-client-urls: "https://CONTROL02_IP:2379"
      listen-peer-urls: "https://CONTROL02_IP:2380"
```

```
      initial-advertise-peer-urls: "https://CONTROL01_IP:2380"
      initial-cluster:
"CONTROL01_HOSTNAME=https://CONTROL01_IP:2380,CONTROL02_HOSTNAME=https://CO
NTROL02_IP:2380"
      initial-cluster-state: existing
   serverCertSANs:
     - CONTROL02_HOSTNAME
     - CONTROL02_IP
   peerCertSANs:
     - CONTROL02_HOSTNAME
     - CONTROL02_IP
networking:
   podSubnet: "192.168.0.0/16"
```

Before running this template, you'll need to move the copied files over to the correct directories. Here's an example that should be similar on your system:

```
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
mv /home/${USER}/admin.conf /etc/kubernetes/admin.conf
```

Once you've copied those files over, you can run a series of `kubeadm` commands to absorb the certificates, and then bootstrap the second node:

```
kubeadm alpha phase certs all --config kubeadm-conf-02.yaml
kubeadm alpha phase kubelet config write-to-disk --config kubeadm-
conf-02.yaml
kubeadm alpha phase kubelet write-env-file --config kubeadm-conf-02.yaml
kubeadm alpha phase kubeconfig kubelet --config kubeadm-conf-02.yaml
systemctl start kubelet
```

Once that's complete, you can add the node to the etcd as well. You'll need to set some variables first, along with the IPs of the virtual machines running your nodes:

```
export CONTROL01_IP=<YOUR_IP_HERE>
export CONTROL01_HOSTNAME=cp01H
export CONTROL02_IP=<YOUR_IP_HERE>
export CONTROL02_HOSTNAME=cp02H
```

Once you've set up those variables, run the following `kubectl` and `kubeadm` commands. First, add the certificates:

```
export KUBECONFIG=/etc/kubernetes/admin.conf
kubectl exec -n kube-system etcd-${CONTROL01_HOSTNAME} -- etcdctl --ca-file
/etc/kubernetes/pki/etcd/ca.crt --cert-file
/etc/kubernetes/pki/etcd/peer.crt --key-file
/etc/kubernetes/pki/etcd/peer.key --endpoints=https://${CONTROL01_IP}:2379
member add ${CONTROL02_HOSTNAME} https://${CP1_IP}:2380
```

Next, phase in the configuration for `etcd`:

```
kubeadm alpha phase etcd local --config kubeadm-config-02.yaml
```

This command will cause the etcd cluster to become unavailable for a short period of time, but that is by design. You can then deploy the remaining components in the `kubeconfig` and `controlplane`, and then mark the node as a master:

```
kubeadm alpha phase kubeconfig all --config kubeadm-conf-02.yaml
kubeadm alpha phase controlplane all --config kubeadm-conf-02.yaml
kubeadm alpha phase mark-master --config kubeadm-conf-02.yaml
```

We'll run through this once more with the third node, adding more value to the initial cluster under etcd's `extraArgs`.

You'll need to create a third `kubeadm-conf-03.yaml` file on the third machine. Follow this template and substitute the variables, like we did previously:

```
apiVersion: kubeadm.k8s.io/v1alpha2
kind: MasterConfiguration
kubernetesVersion: v1.11.0
apiServerCertSANs:
- "LB_DNS"
api:
   controlPlaneEndpoint: "LB_DNS:LB_PORT"
etcd:
 local:
   extraArgs:
     listen-client-urls: "https://127.0.0.1:2379,https://CONTROL03_IP:2379"
     advertise-client-urls: "https://CONTROL03_IP:2379"
     listen-peer-urls: "https://CONTROL03_IP:2380"
     initial-advertise-peer-urls: "https://CONTROL03_IP:2380"
     initial-cluster:
"CONTRL01_HOSTNAME=https://CONTROL01_IP:2380,CONTROL02_HOSTNAME=https://CON
TROL02_IP:2380,CONTRL03_HOSTNAME=https://CONTROL03_IP:2380"
     initial-cluster-state: existing
   serverCertSANs:
```

```
      - CONTRL03_HOSTNAME
      - CONTROL03_IP
    peerCertSANs:
      - CONTRL03_HOSTNAME
      - CONTROL03_IP
networking:
    podSubnet: "192.168.0.0/16"
```

You'll need to move the files again:

```
mkdir -p /etc/kubernetes/pki/etcd
mv /home/${USER}/ca.crt /etc/kubernetes/pki/
mv /home/${USER}/ca.key /etc/kubernetes/pki/
mv /home/${USER}/sa.pub /etc/kubernetes/pki/
mv /home/${USER}/sa.key /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.crt /etc/kubernetes/pki/
mv /home/${USER}/front-proxy-ca.key /etc/kubernetes/pki/
mv /home/${USER}/etcd-ca.crt /etc/kubernetes/pki/etcd/ca.crt
mv /home/${USER}/etcd-ca.key /etc/kubernetes/pki/etcd/ca.key
mv /home/${USER}/admin.conf /etc/kubernetes/admin.conf
```

And, once again you'll need to run the following commands in order bootstrap them:

```
kubeadm alpha phase certs all --config kubeadm-conf-03.yaml
kubeadm alpha phase kubelet config write-to-disk --config kubeadm-
conf-03.yaml
kubeadm alpha phase kubelet write-env-file --config kubeadm-conf-03.yaml
kubeadm alpha phase kubeconfig kubelet --config kubeadm-conf-03.yaml
systemctl start kubelet
```

And then, add the nodes to the etcd cluster once more:

```
export CONTROL01_IP=<YOUR_IP_HERE>
export CONTROL01_HOSTNAME=cp01H
export CONTROL03_IP=<YOUR_IP_HERE>
export CONTROL03_HOSTNAME=cp03H
```

Next, we can set up the etcd system:

```
export KUBECONFIG=/etc/kubernetes/admin.conf

kubectl exec -n kube-system etcd-${CONTROL01_HOSTNAME} -- etcdctl --ca-file
/etc/kubernetes/pki/etcd/ca.crt --cert-file
/etc/kubernetes/pki/etcd/peer.crt --key-file
/etc/kubernetes/pki/etcd/peer.key --endpoints=https://${CONTROL01_IP}:2379
member add ${CONTROL03_HOSTNAME} https://${CONTROL03_IP}:2380

kubeadm alpha phase etcd local --config kubeadm-conf-03.yaml
```

After that's complete, we can once again deploy the rest of the components of the control plane and mark the node as a master. Run the following commands:

```
kubeadm alpha phase kubeconfig all --config kubeadm-conf-03.yaml

kubeadm alpha phase controlplane all --config kubeadm-conf-03.yaml

kubeadm alpha phase mark-master --config kubeadm-conf-03.yaml
```

Great work!

Installing workers

Once you've configure the masters, you can join the worker nodes to the cluster. You can only do this once you've installed networking, the container, and any other prerequisites you've added to your clusters such as DNS, though. However, before you add the worker nodes, you'll need to configure a pod network. You can find more information about the pod network add-on here: `https://kubernetes.io/docs/setup/independent/create-cluster-kubeadm/#pod-network`.

Cluster life cycle

There are a few more key items that we should cover so that you're armed with full knowledge about the items that can help you with creating highly available Kubernetes clusters. Let's discuss how you can use admission controllers, workloads, and custom resource definitions to extend your cluster.

Admission controllers

Admission controllers are Kubernetes code that allows you to intercept a call to the Kubernetes API server after it has been authenticated and authorized. There are standard admission controllers that are included with the core Kubernetes system, and people also write their own. There are two controllers that are more important than the rest:

- The `MutatingAdmissionWebhook` is responsible for calling `Webhooks` that mutate, in serial, a given request. This controller only runs during the mutating phase of cluster operating. You can use a controller like this in order to build business logic into your cluster to customize admission logic with operations such as `CREATE`, `DELETE`, and `UPDATE`. You can also do things like automate the provisioning of storage with the `StorageClass`. Say that a deployment creates a `PersistentVolumeClaim`; a webhoook can automate the provisioning of the `StorageClass` in response. With the `MutatingAdmissionWebhook`, you can also do things such as injecting a sidecar into a container prior to it being built.
- The `ValidatingAdmissionWebhook` is what the admission controller runs in the validation phase, and calls any webhooks that will validate a given request. Here, webhooks are called in parallel, in contrast to the serial nature of the `MutatingAdmissionWebhook`. It is key to understand that none of the webhooks that it calls are allowed to mutate the original object. An example of a validating webhook such as this is incrementing a quota.

Admission controllers and their mutating and validating webhooks are very powerful, and importantly provide Kubernetes operators with additional control without having to recompile binaries such as the `kube-apiserver`. The most powerful example is Istio, which uses webhooks to inject its Envoy sidecar in order to implement load balancing, circuit breaking, and deployment capabilities. You can also use webhooks to restrict namespaces that are created in multi-tenant systems.

You can think of mutation as a change and validation as a check in the Kubernetes system. As the associated ecosystem of software grows, it will become increasingly important from a security and validation standpoint to use these types of capabilities. You can use controllers, with their change and check capabilities to do things such as override image pull policies in order to enable or prevent certain images from being used on your cluster.

These admission controllers are essentially part of the cluster control plane, and can only be run by cluster administrators.

Here's a very simple example where we'll check that a namespace exists in the admission controller.

NamespaceExists: This admission controller checks all requests on namespaced resources other than Namespace itself. If the namespace referenced from a request doesn't exist, the request is rejected. You can read more about this at `https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/#namespaceexists`.

First, let's grab Minikube for our cluster and check which namespaces exist:

```
master $ kubectl get namespaces
 NAME STATUS AGE
 default Active 23m
 kube-public Active 23m
 kube-system Active 23m
```

Great! Now, let's try and create a simple deployment, where we put it into a namespace that doesn't exist. What do you think will happen?

```
master $ kubectl run nodejs --image nodej2 --namespace not-here
 Error from server (NotFound): namespaces "not-here" not found
```

So, why did that happen? If you guessed that our `ValidatingAdmissionWebhook` picked up on that request and blocked it, you'd be correct!

Using admission controllers

You can turn admission controllers on and off in your server with two different commands. Depending on how your server was configured and how you started `kube-apiserver`, you may need to make changes against `systemd`, or against a manifest that you created to start up the API server in the first place.

Generally, to enable the server, you'll execute the following:

```
kube-apiserver --enable-admission-plugins
```

And to disable it, you'll change that to the following:

```
kube-apiserver --disable-admission-plugins=
```

If you're running Kubernetes 1.10 or later, there is a set of recommended admission controllers for you. You can enable them with the following:

```
kube-apiserver --enable-admission-
plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,D
efaultTolerationSeconds,MutatingAdmissionWebhook,ValidatingAdmissionWebhook
,ResourceQuota
```

In earlier version of Kubernetes, there weren't separate concepts of mutating and validating, so you'll have to read the documentation to understand the implication of using admission controllers on earlier versions of the software.

The workloads API

The workloads API is an important concept to grasp in order to understand how managing objects has stabilized over the course of many releases in Kubernetes. In the early days of Kubernetes, pods and their workloads were tightly coupled with containers that shared the CPU, networking, storage, and life cycle events. Kubernetes introduced concepts such as replication, then deployment, and then labels, and helped manage 12-factor applications. StatefulSets were introduced as Kubernetes operators moved into stateful workloads.

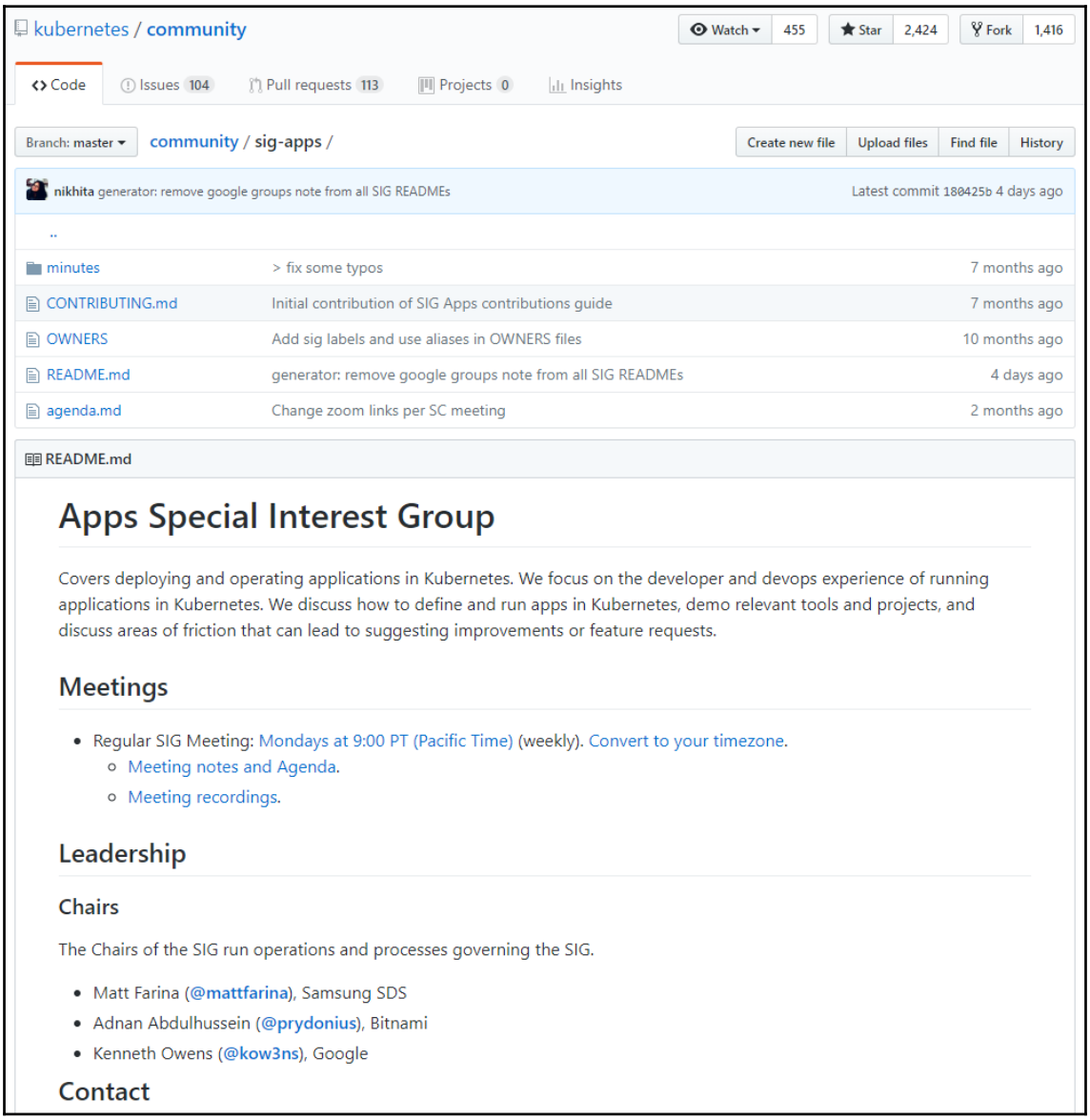
Over time, the concept of the Kubernetes workload became a collective of several parts:

- Pods
- ReplicationController
- ReplicaSet
- Deployment
- DaemonSet
- StatefulSet

These pieces are the current state of the art for orchestrating a reasonable swath of workload types in Kubernetes, but unfortunately the API was spread across many different parts of the Kubernetes codebase. The solution to this was many months of hard work to centralize all of this code, after making many backwards compatibility breaking changes, into `apps/v1` API. Several key decisions were made when making the move to `apps/v1`:

- **Default selector behavior**: Unspecified label selectors are used to default to an auto-generated selector culled from the template labels
- **Immutable selectors**: While changing selectors is useful in some cases for deployment, it has always been against Kubernetes recommendations to mutate a selector, so the change was made to enable promoted canary-type deployments and pod relabeling, which is orchestrated by Kubernetes
- **Default rolling updates**: The Kubernetes programmers wanted RollingUpdate to be the default form, and now it is
- **Garbage collection**: In 1.9 and `apps/v1`, garbage collection is more aggressive, and you won't see pods hanging around any more after DaemonSets, ReplicaSets, StatefulSets, or Deployments are deleted

If you'd like more input into these decisions, you can join the *Apps Special Interest Group*, which can be found here: `https://github.com/kubernetes/community/tree/master/sig-apps`:



For now, you can consider the workloads API to be stable and backwards compatible.

Custom resource definitions

The last piece we'll touch on in our HA chapter is custom resources. These are an extension of the Kubernetes API, and are compliment with the admission controllers we discussed previously. There are several methods for adding custom resources to your cluster, and we'll discuss those here.

As a refresher, keep in mind that a non-custom resource in Kubernetes is an endpoint in the Kubernetes API that stores a collection of similar API objects. You can use custom resources to enhance a particular Kubernetes installation. We'll see examples of this with Istio in later chapters, which uses CRDs to put prerequisites into place. Custom resources can be modified, changed, and removed with `kubectl`.

When you pair custom resources with controllers, you have the ability to create a declarative API, which allows you to set the state for your gathered resources outside of the cluster's own life cycle. We touched on an example of the custom controller and custom resource pattern earlier in this book with the operator pattern. You have a couple of options when deciding whether or not to create a custom resource with Kubernetes. The documentation recommends the following decision table when choosing:

Consider API aggregation if:	Prefer a stand-alone API if:
Your API is Declarative.	Your API does not fit the Declarative model.
You want your new types to be readable and writable using `kubectl` .	`kubectl` support is not required
You want to view your new types in a Kubernetes UI, such as dashboard, alongside built-in types.	Kubernetes UI support is not required.
You are developing a new API.	You already have a program that serves your API and works well.
You are willing to accept the format restriction that Kubernetes puts on REST resource paths, such as API Groups and Namespaces. (See the API Overview.)	You need to have specific REST paths to be compatible with an already defined REST API.
Your resources are naturally scoped to a cluster or to namespaces of a cluster.	Cluster or namespace scoped resources are a poor fit; you need control over the specifics of resource paths.
You want to reuse Kubernetes API support features.	You don't need those features.

Image credit: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/#should-i-add-a-custom-resource-to-my-kubernetes-cluster

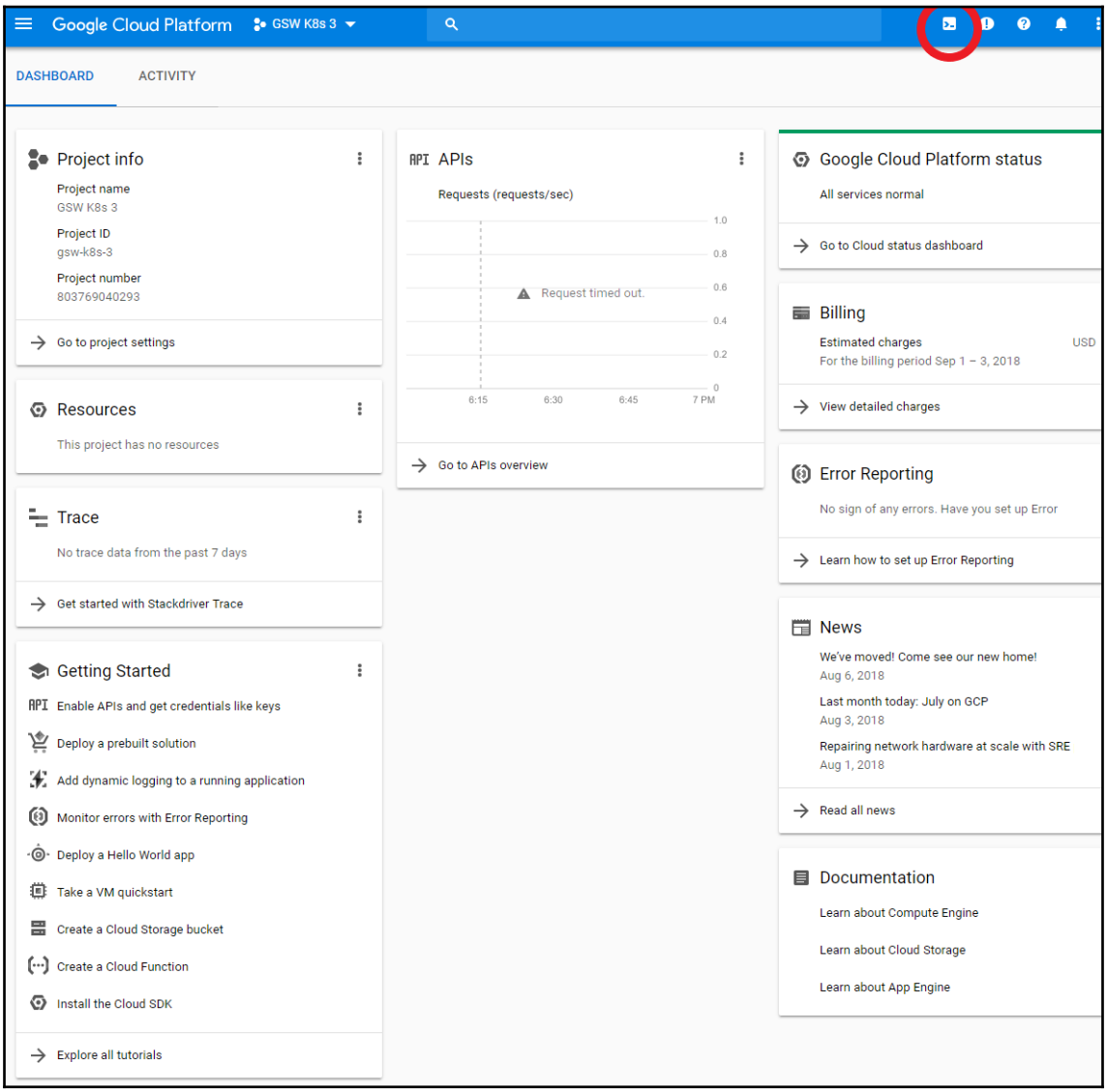
A key point in deciding to write a custom resource is to ensure that your API is declarative. If it's declarative, it's a good fit for a custom resource. You can write custom resources in two ways, with custom resource definitions or through API aggregation. API aggregation requires programming, and we won't be getting into that topic for this chapter, but you can read more about it here: `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/apiserver-aggregation/`.

Using CRDs

While Aggregated APIs are more flexible, CRDs are easier to user. Let's try and create the example CRD from the Kubernetes documentation.

First, you'll need to spin up your Minikube cluster and the GKE cluster on GCP, which will be one of your own clusters or a playground such as Katacoda. Let's jump into a Google Cloud Shell and give this a try.

Once on your GCP home page, click the CLI icon, which is circled in red in the following screenshot:



Once you're in the shell, create a quick Kubernetes cluster. You may need to modify the cluster version in case older versions aren't supported:

```
gcloud container clusters create gswk8s \
  --cluster-version 1.10.6-gke.2 \
  --zone us-east1-b \
  --num-nodes 1 \
  --machine-type n1-standard-1
<lots of text>
...
Creating cluster gsk8s...done.
Created
[https://container.googleapis.com/v1/projects/gsw-k8s-3/zones/us-east1-b/cl
usters/gsk8s].
To inspect the contents of your cluster, go to:
https://console.cloud.google.com/kubernetes/workload_/gcloud/us-east1-b/gsk
8s?project=gsw-k8s-3
kubeconfig entry generated for gsk8s.
NAME LOCATION MASTER_VERSION MASTER_IP MACHINE_TYPE NODE_VERSION NUM_NODES
STATUS
gsk8s us-east1-b 1.10.6-gke.2 35.196.63.146 n1-standard-1 1.10.6-gke.2 1
RUNNING
```

Next, add the following text to `resourcedefinition.yaml`:

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form:
<plural>.<group>
  name: crontabs.stable.example.com
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: stable.example.com
  # list of versions supported by this CustomResourceDefinition
  version: v1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: crontabs
    # singular name to be used as an alias on the CLI and for display
    singular: crontab
    # kind is normally the CamelCased singular type. Your resource
```

```
manifests use this.
    kind: CronTab
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - cront
```

Once you've added that, we can create it:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl apply -f
resourcedefinition.yaml
customresourcedefinition "crontabs.stable.example.com" created
```

Great! Now, this means that our RESTful endpoint will be available at the following URI: `/apis/stable.example.com/v1/namespaces/*/crontabs/`. We can now use this endpoint to manage custom objects, which is the other half of our key CRD value.

Let's create a custom object called `os-crontab.yaml` so that we can insert some arbitrary JSON data into the object. In our case, we're going to add the OS metadata for cron and the crontab interval.

Add the following:

```
apiVersion: "stable.example.com/v1"
kind: CronTab
metadata:
  name: cron-object-os-01
spec:
  intervalSpec: "* * 8 * *"
  os: ubuntu
```

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl create -f os-crontab.yaml
crontab "cron-object-os-01" created
```

Once you've created the resource, you can get it as you would any other Deployment, StatefulSet, or other Kubernetes object:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl get crontab
NAME                AGE
cron-object-os-01   38s
```

If we inspect the object, we would expect to see a bunch of standard configuration, plus the `intervalSpec` and OS data that we encoded into the CRD. Let's check and see if it's there.

We can use the alternative name, `cront`, that we gave in the CRD in order to look it up. I've highlighted the data as follows—nice work!

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl get cront-o yaml
apiVersion: v1
items:
- apiVersion: stable.example.com/v1
  kind: CronTab
  metadata:
    clusterName: ""
    creationTimestamp: 2018-09-03T23:27:27Z
    generation: 1
    name: cron-object-os-01
    namespace: default
    resourceVersion: "2449"
    selfLink: /apis/stable.example.com/v1/namespaces/default/crontabs/cron-
object-os-01
    uid: eb5dd081-afd0-11e8-b133-42010a8e0095
  spec:
    intervalSpec: '* * 8 * *'
    os: Ubuntu
kind: List
metadata:
  resourceVersion: ""
  selfLink: ""
```

Summary

In this chapter, we looked into the core components of HA. We explored the ideas of availability, uptime, and fragility. We took those concepts and explored how we could achieve five nines of uptime.

Additionally, we explored the key components of a highly available cluster, the etcd and control plane nodes, and left room to imagine the other ways that we'd build HA into our clusters using hosted PaaS offerings from the major cloud providers.

Later, we looked at the cluster life cycle and dug into advanced capabilities with a number of key features of the Kubernetes system: admission controllers, the workload API, and CRS.

Lastly, we created a CRD on a GKE cluster within GCP in order to understand how to begin building these custom pieces of software.

Questions

1. What are some ways to measure the quality of an application?
2. What is the definition of uptime?
3. How many nines of availability should a Kubernetes system strive for?
4. What does it mean for a system to fail in predefined ways, while still providing reduced functionality?
5. Which PaaS provides highly available master and worker nodes across multiple availability zones?
6. What's a stacked node?
7. What's the name of the API that collects all of the controllers in a single, unified API?

Further reading

If you'd like to read more about high availability and mastering Kubernetes, check out the following Packt resources:

- `https://www.packtpub.com/virtualization-and-cloud/kubernetes-cookbook-second-edition`
- `https://www.packtpub.com/virtualization-and-cloud/mastering-kubernetes`

11
Kubernetes SIGs, Incubation Projects, and the CNCF

In this chapter, we're going to discuss how to get involved in the softer, social side of the Kubernetes ecosystem. We'll go into detail on how the **Cloud Native Computing Foundation** (**CNCF**) works, and the various efforts being made to orchestrate open source software at a global level. There's interest in our ecosystem at every level, from the individual contributor all the way up to the Fortune 100 mega-corporation.

We'll explore how the CNCF and its predecessors, the Linux and Apache Foundations, guide interest and contributions into the people and software economy. Some of the key areas will manage governance, tracking, and processes that are designed to keep people, process, and technology evolving in a sustainable, reliable model. In this chapter, we'll explore several key areas:

- How is the community around the Kubernetes ecosystem constructed? How is it different from the traditional **Free and Open Source Software** (**FOSS**) or **Open Source Software** (**OSS**) movements?
- How can you get involved with the discussion in order to understand and participate in the evolution of the ecosystem?
- What are the major projects, and how are they categorized?
- How can you chose the right tools for the job, given all of the change?
- How can you get involved with open source software in general?

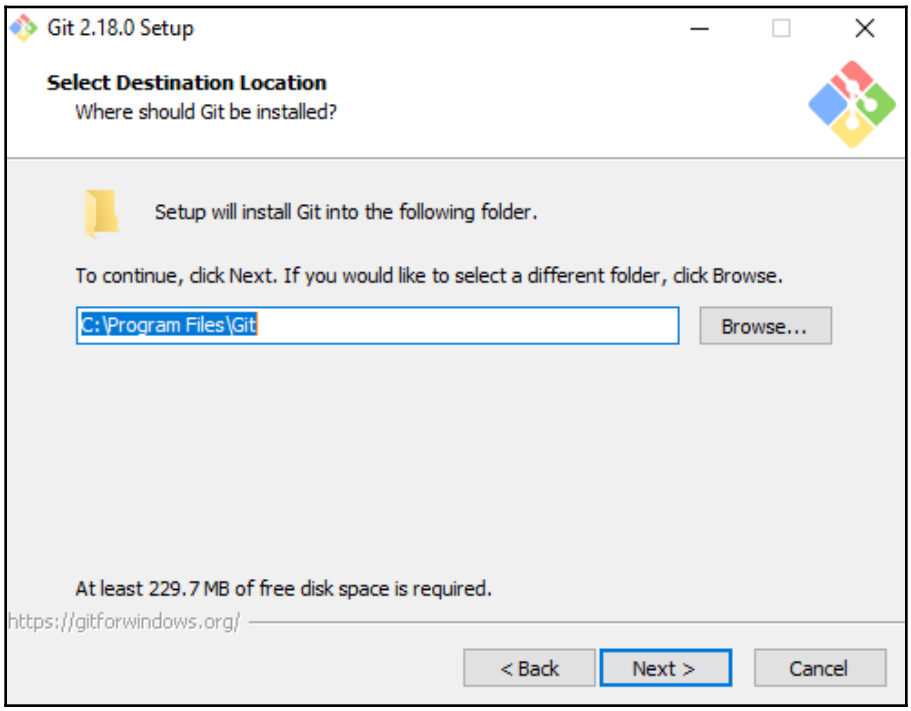
Technical requirements

In order to move quickly through this chapter, you should make sure that you have a GitHub account set up, with SSH key and account details configured correctly. Why is this important, you may ask? Well, to get involved with the CNCF, and the Linux or Apache Foundations, you'll need a way to browse, consume, and contribute to code. Git is the underlying tool and process that's used to participate, so we'll make sure here that our toolset is correctly set up before proceeding to the higher level topics.

You can sign up for GitHub and once you've added the account, you can review the help area in the **GitHub Guides** section of the website at `https://guides.github.com/`. For our purposes in this chapter, you'll need to set up an SSH key in order to start cloning, signing, and committing code.

If you're on Windows, you'll need to use Git Bash, or something similar, to generate a key. You can download Git Bash from `https://gitforwindows.org/`.

Install the software first, and then we'll set up your environment. The installation looks as follows:



Setting up Git for contributions

Type the following command, using your email address in place of mine:

```
$ ssh-keygen -t rsa -b 4096 -C "jesse@gsw-k8s-3rd.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/jesse/.ssh/id_rsa):
Created directory '/c/Users/jesse/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/jesse/.ssh/id_rsa.
Your public key has been saved in /c/Users/jesse/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:AtDI+/yPNxi8y6WzdTecvd6U/ir6Q8pBtg0dv/ZhHlY jesse@gsw-k8s-3rd.com
The key's randomart image is:
+---[RSA 4096]----+
| ..o             |
|  o..        . |
|   ..     . o |
|  . .    + . . E|
|   o .. So +   ..|
|    o o. o.ooo=.|
|     . +o..+=.=o+|
|     .==o.o.o..=.|
|      **...o.++o+|
+----[SHA256]-----+
$ ~/Documents/Code
```

This will generate a key pair that you can add to your `ssh-agent`. You can also use GitHub Desktop if you'd prefer to avoid SSH keys, but we would recommend that you use native CLI tools.

Ensure that the agent is running with the following command:

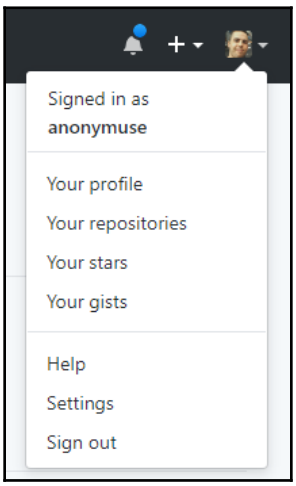
```
$ eval $(ssh-agent -s)
Agent pid 11684
```

You can then add your key to the agent as follows:

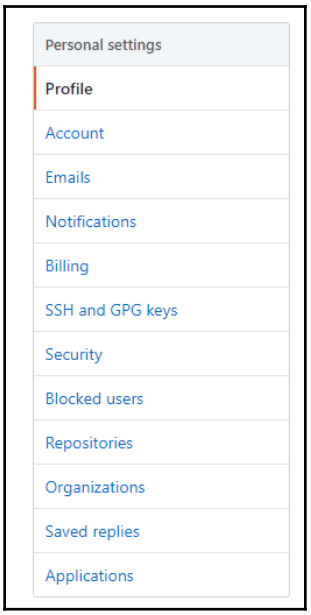
```
$ ssh-add ~/.ssh/id_rsa
Identity added: /c/Users/jesse/.ssh/id_rsa (/c/Users/jesse/.ssh/id_rsa)
```

We won't go over the instructions in detail here, but you can find the macOS and Linux instructions here `https://help.github.com/articles/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent/`.

Next up, we'll add your public key to your GitHub account so we can get going with the rest of the chapter. Navigate to `https://github.com`, click on your profile, and bring up your **Settings** page:



Then, we'll click on **SSH and GPG keys** and add in the key that you created on your machine:



Click **New SSH key** and then add your generated `id_rsa.pub` key. Importantly, do not add your `id_rsa` key, as that's private and should be kept safe and offline!

You can copy your public SSH key to your clipboard with the following command in Windows:

```
$ clip < ~/.ssh/id_rsa.pub
```

You can test it out once you've configured it with this command:

```
$ ssh -vT git@github.com
OpenSSH_7.7p1, OpenSSL 1.0.2o 27 Mar 2018
debug1: Reading configuration data /etc/ssh/ssh_config
debug1: Connecting to github.com [192.30.253.113] port 22.
debug1: Connection established.
....SNIP...
Hi anonymuse! You've successfully authenticated, but GitHub does not
provide shell access.
debug1: channel 0: free: client-session, nchannels 1
Transferred: sent 3328, received 2048 bytes, in 0.1 seconds
Bytes per second: sent 36660.1, received 22560.1
debug1: Exit status 1
```

If you see a message welcoming you by your username, you're all set!

Git's benefits

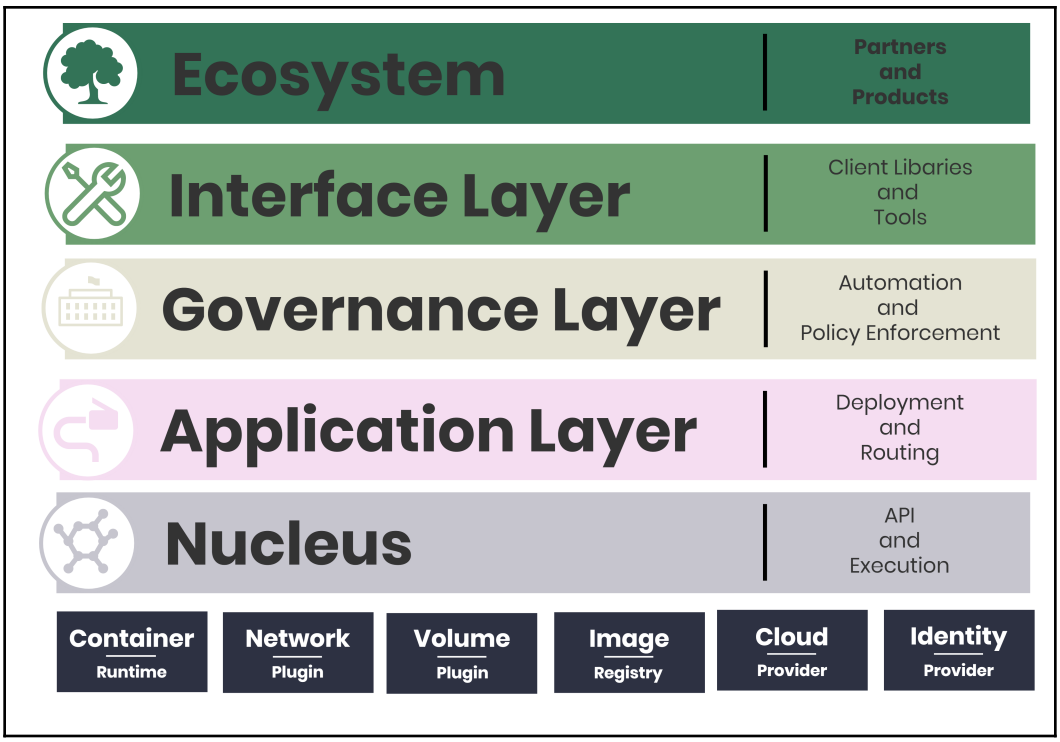
Once you have your keys, add them to your GitHub account in order to accomplish two important things:

- **Forking, pull requests, and contributions**: You'll be able to create private forks and pull requests using your own repositories, which allows you to begin contributing to the projects in the container ecosystem. You'll need the SSH key and the aforementioned programs in order to interact with Git, which is the underlying technology that powers this collaboration. There's a similar setup for GitLab and Bitbucket, but GitHub is currently the most popular tool and happens to be where all of the CNCF projects reside.

- **Digital chain of custody**: You'll be able to sign your commits. In order to participate in many of the cutting-edge Kubernetes ecosystem projects, you'll need to digitally sign your commits such that they're able to be attributed back to you. Many of the technologies that we've touched on in these books are used to power large infrastructure at the world's most advanced companies, and it's important for OSS to establish a strong chain of custody for highly distributed code development. The fingerprint of SSL and your machine is an essential piece of authentication and authorization.

CNCF structure

As a refresher, let's remind ourselves about the entire Kubernetes system, so we can understand conceptually where the ecosystem referred to in this chapter sits:



In this chapter, we're talking about the top, greenest layer in the preceding diagram. This layer is made up of hundreds of companies and products that power the software and frameworks needed to run Kubernetes at scale. You can find the highest level of grouping of this layer in a couple of places:

- The first place to check is the Kubernetes Community GitHub repository:
    - You can find the repository at `https://github.com/kubernetes/community`, and it's a good starting point for anyone who's interested in joining the code-powered portions of the Kubernetes system. In the preceding diagram, consider the layers nucleus through interface, that is, layers one through four. Here's where we'll find the **Special Interest Groups** (**SIGs**), which will allow us to branch us out into the ecosystem layer where we can explore the supporting technologies that enable Kubernetes to stay focused on its core functionality.
- The second place you can investigate to dig deeper into the ecosystem is the CNCF landscape:
    - The landscape is actually broken up into a few useful parts that can help anyone from individual users, all the way up to large enterprises, make decisions on what technology to adopt, what to wait on, and what to leave behind. Here's where we'll really dig into the supporting ecosystem in order to understand what's meant to be in Kubernetes core, and what's meant to be provided by the ecosystem.

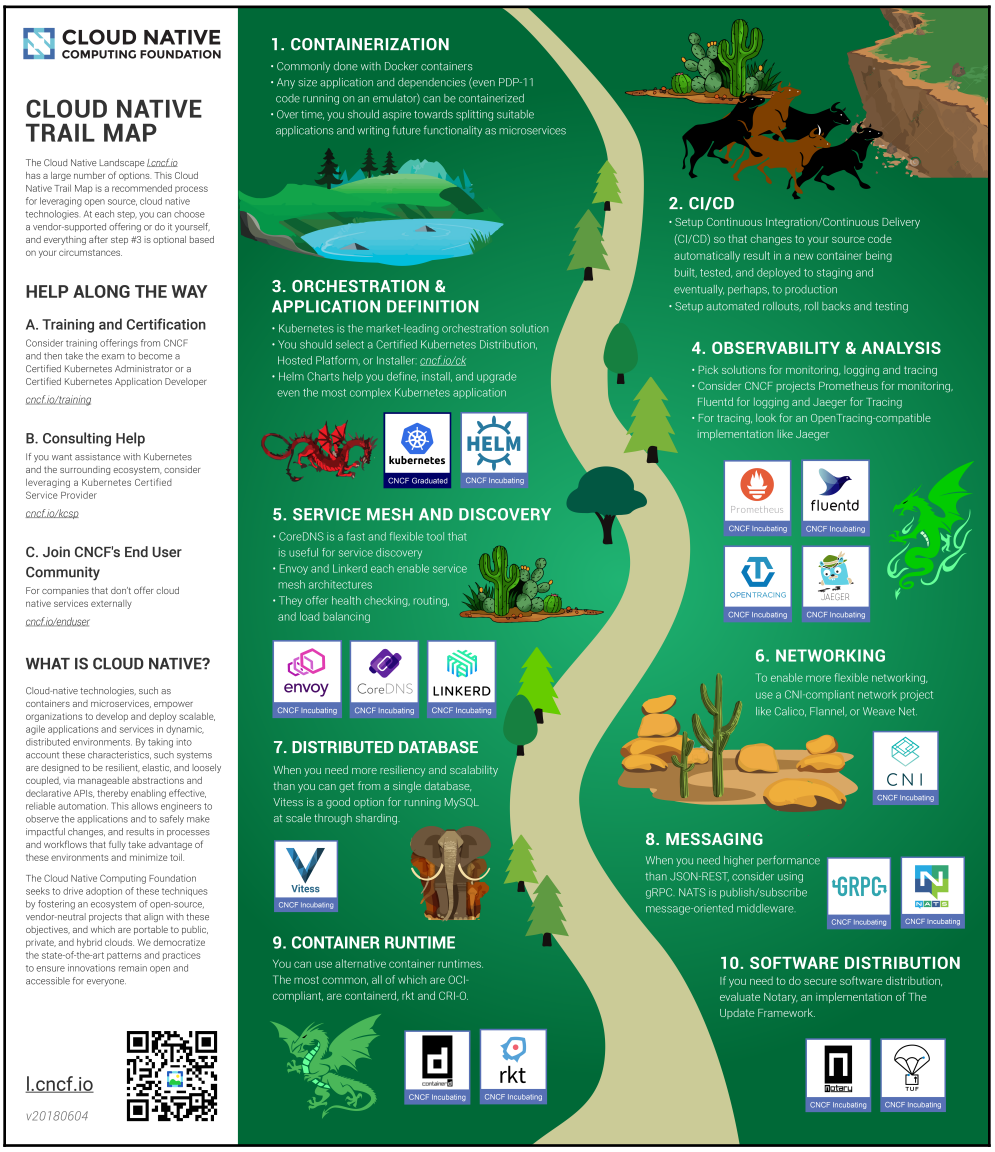
The Kubernetes documentation neatly answers the question, what is Kubernetes with the following quote:

> *Kubernetes provides a container-centric management environment. It orchestrates computing, networking, and storage infrastructure on behalf of user workloads. This provides much of the simplicity of Platform as a Service (PaaS) with the flexibility of Infrastructure as a Service (IaaS), and enables portability across infrastructure providers.*
>
> *For more information visit* `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`.

So, if that's what Kubernetes is, what isn't it?

What Kubernetes isn't

The most succinct—and currently, the best—viewpoint on the Kubernetes ecosystem, at a level that's digestible both for individuals running their own small scale clusters and executives looking to understand the massive scope of the Kubernetes ecosystem, is the **Cloud Native Trail Map**, shown here:



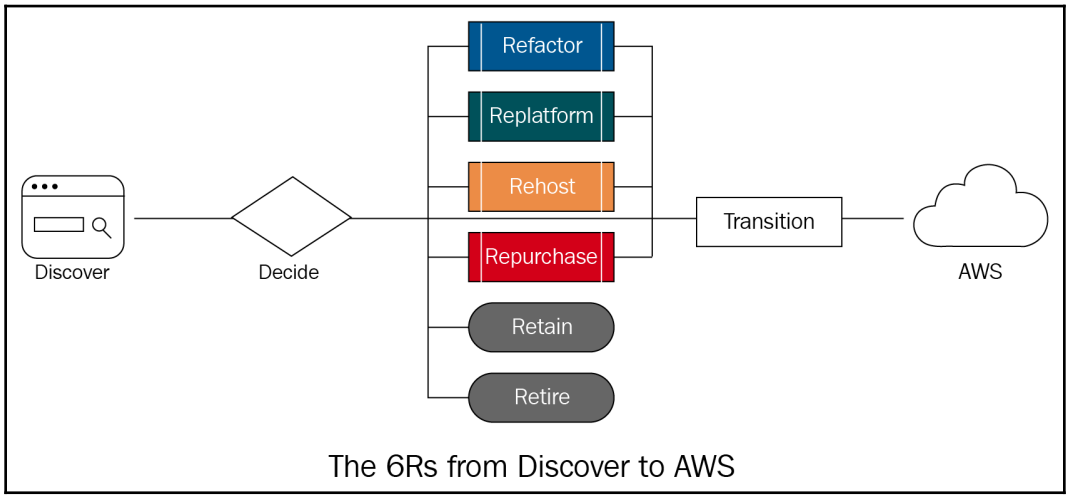
The trail map helps us to break down all of the efforts to support Kubernetes currently going on outside of the core container-centric management environment that we alluded to in the preceding section. Outside of networking, storage, and compute, there are a lot of moving pieces that need to work in order for complex, microservice-based, cloud-native applications to run at scale. What else is needed to support the Kubernetes PaaS system?

You should treat each of these layers as a choice; choose one technology (or multiple, to render a proof-of-concept and decision) and see how it works.

For example, let's take containerization: at this point, it's table stakes to run your application as a containerized workload, but it may take your organization time to re-architect your applications, or to learn how to use Dockerfiles and build cloud-native applications.

There are traditionally *6Rs* involved in moving your application to the cloud or container orchestration and scheduling platform.

Here's a diagram demonstrating the 6Rs referenced in the preceding tip box that you can utilize to update your applications:



The 6Rs from Discover to AWS

While this 6Rs formula was intended for considering a move to the cloud, it's also very useful when migrating to containers as well. Keep in mind here that not all of your applications will be well suited to running in containers (**Retain**), while some of them should be deprecated for OSS (**Retire**). A good way to start moving into containerized workloads is to simply drop a large monolithic application, such as a `Java .war` file or Python program, directly into the container and let it run as is (**Rehost**). In order to achieve the maximum benefits of containerization, and to take advantage of the cutting edge features of Kubernetes, you'll most likely need to explore refactoring, re-imagining, and rearchitecting your application (**Refactor**).

The next area of focus for anyone running a platform is **Continuous Integration and Continuous Delivery** (**CICD**). You'll need to manage both infrastructure and application-as-code in order to provide seamless rollouts, updates, and testing. In this new world, infrastructure and application are both first-class citizens when it comes to software.

Observability and analysis are also important in this realm of highly complex software systems that control both infrastructure and application. The CNCF breaks down solutions into sandbox, graduated, and incubation areas:

- **Sandbox**: OpenMetrics is designed to create a common standard, building from Prometheus, to transmit metrics at scale. OpenMetrics uses standard text formats, as well as protocol buffers in order to serialize structured data in a language and platform-neutral manner.
- **Incubating**: Here, we see Fluentd, Jaeger, and OpenTracing. Fluentd has been around for some time now, for those folks who've used the **Elasticsearch, Logstash, Kibana** (**ELK**) stack to collect metrics. It's an open source data aggregator that allows you to unify a set of logs from disparate sources. Jaeger helps operators to monitor and resolve issues in complex, distributed systems by providing tracing that can help unearth problems in modern microservice systems. Similarly to OpenMetrics, OpenTracing is an effort to build a standard for distributed tracing in microservices and OSS. As our systems become more deeply interconnected with know-nothing APIs, it is ever more important to introspect the connections of these systems.
- **Graduated**: Along with Kubernetes, Prometheus remains the only other project currently graduated within the CNCF. Prometheus is a monitoring and alerting system that can use a number of different time series databases to display system status.

Service mesh and discovery is the next step along the Cloud Native Trail Map. This tier can be thought of as an additional capability set on top of the base functionality of Kubernetes, which can be seen as a set of the following capabilities:

- A single Kubernetes API control plane
- An authentication and authorization model
- A namespaced, predictable, cluster-scoped resource description scheme
- A container scheduling and orchestration domain
- A pod-to-pod and ingress network routing domain

The three products in this portion of the map are CoreDNS, Envoy, and Linkerd. CoreDNS replaces `kube-dns` in your cluster, and provides the ability to chain multiple plugins together to create deeper functionality for looking up customer providers. CoreDNS will soon replace `kube-dns` as the default DNS provider for Kubernetes. Envoy is a service proxy that is built into the popular Istio product. Istio is a control that uses the Envoy binary as a data plane to provide common capabilities to a homogeneous set of software or services. Envoy provides the foundational capabilities for a service mesh that runs on top of the application that runs on Kubernetes, which provides an additional layer of resilience in the form of circuit breaking, rate limiting, load balancing, service discovery, routing, and application introspection in the form of metrics and logging. Linkerd has nearly all the same functionality as Envoy, as it's also a data plane for the service mesh.

Networking is the next building block that we can add to the Kubernetes ecosystem. The **Container Network Interface** (**CNI**) is one of several interfaces that are currently being developed from within the CNCF ecosystem. Multiple options for Kubernetes cluster networking are being developed in order to cope with the complex feature requirements that applications have these days. Current options include the following:

- Calico
- Flannel
- Weave Net
- Cilium
- Contiv
- SR-IOV
- Knitter

The Kubernetes team also provides a core set of plugins for the system that manage IP address allocation and interface creation.

Read more about the standard plugins at `https://github.com/containernetworking/plugins/`.

Reading from the GitHub project homepage, the CNI is described as follows:

> *CNI (Container Network Interface), a Cloud Native Computing Foundation project, consists of a specification and libraries for writing plugins to configure network interfaces in Linux containers, along with a number of supported plugins. CNI concerns itself only with network connectivity of containers and removing allocated resources when the container is deleted. Because of this focus, CNI has a wide range of support and the specification is simple to implement.*
>
> *For more information on Cloud Native Computing Foundation visit* `https://www.cncf.io/`.

There isn't currently a lot of activity in the distributed database portion of the trail map, simply because most of the workloads that currently run on Kubernetes tend to be stateless. There is a project incubating currently, named Vitess, which is attempting to provide a horizontal scaling model for the ever-popular MySQL database system. In order to scale MySQL across the pod-structured infrastructure of Kubernetes, the makers of Vitess are focusing on sharding out MySQL's data store in order to distribute it among the nodes of the cluster. It is similar to other NoSQL systems that, in this fashion, rely on data being replicated and spread out over several nodes. Vitess has been used at scale at YouTube since 2011, and is a promising technology for those looking to venture deeper into stateful workloads on Kubernetes.

For those operators who are pushing the limits of the Kubernetes system, there are several high-performance options for increasing the speed of your system. gRPC is a **Remote Procedure Call** (**RPC**) framework that was developed by Google to help clients and servers communicate transparently. gRPC is available in many languages, including C++, Java, Python, Go, Ruby, C#, Node.js, and more. gRPC uses ProtoBufs and is based on the simple concept that a service should have methods that can be called from another remote service. By defining these methods and parameters within the code, gRPC allows for large, complex applications to be built in pieces. NATS is a message queue that implements a distributed queue system that provides publish/subscribe and request/reply functionality, allowing the implementation of a highly scalable and secure foundation for **inter-process communication** (**IPC**).

The container runtime portion of the trail map is an area where there's been some contention. There are currently two options in the CNCF: `containerd` and `rkt`. These two technologies do not currently conform to the **Container Runtime Interface** (**CRI**), which is a new standard that attempts to create a shared understanding of what a container runtime should do. There are a few examples outside of the CNCF that currently conform to CRI standards:

- CRI-O
- Docker CRI shim
- Frakti
- rkt

There are also interesting players, such as Kata Containers, which are compliant with **Open Container Initiative** (**OCI**) standards and seek to offer containers running on lightweight virtual machines using technology from Hyper's runV and Intel's Clear Containers. Here, Kata replaces the traditional runC runtime in order to provide a container with a lightweight VM that contains its own mini-kernel.

The last piece of the trail map puzzle is software distribution, which is covered by Notary and the TUF framework. These are tools designed to aid in the secure distribution of software. Notary is a client/server framework that allows people to build trust over discretionary collections of data. In short, publishers can sign data content and then send that to consumers who have access to public key cryptographic systems, which allow them to validate the publisher's identity and the data.

The TUF framework is used by Notary, which is a framework that allows for the secure update of a software system. TUF is used in delivery secure updates **over-the-air** (**OTA**) to automobiles.

Kubernetes SIGs

In addition to all the players mentioned previously, there is a set of complementary SIGs that meet regularly to discuss issues and opportunities from within a given focus area of the Kubernetes ecosystem. From within those SIGs, there are sub-bounded working groups that aim to accomplish a specific goal. There are also sub-projects that further cut up the interest space, and committees, which are there to define meta-standards and address community-wide issues.

Here's a list of the current SIGs in operation, with the current chairs and meeting schedules:

Name	Chairs	Meetings
API Machinery (https://github.com/kubernetes/community/blob/master/sig-api-machinery/README.md)	• Daniel Smith (https://github.com/lavalamp), Google • David Eads (https://github.com/deads2k), Red Hat	Regular SIG Meeting: Wednesdays at 11:00 PT (Pacific Time) (biweekly) (https://docs.google.com/document/d/1FQx0BPlkkl1Bn0c9ocVBxYIKojpmrS1CFP5h0DI68AE/edit)
Apps (https://github.com/kubernetes/community/blob/master/sig-apps/README.md)	• Matt Farina (https://github.com/mattfarina), Samsung SDS • Adnan Abdulhussein (https://github.com/prydonius), Bitnami • Kenneth Owens (https://github.com/kow3ns), Google	Regular SIG Meeting: Mondays at 9:00 PT (Pacific Time) (weekly)
Architecture (https://github.com/kubernetes/community/blob/master/sig-architecture/README.md)	• Brian Grant (https://github.com/bgrant0607), Google • Jaice Singer DuMars (https://github.com/jdumars), Google	Regular SIG Meeting: Thursdays at 19:00 UTC (weekly)

Auth (https://github.com/kubernetes/community/blob/master/sig-auth/README.md)	• Jordan Liggitt (https://github.com/liggitt), Red Hat • Mike Danese (https://github.com/mikedanese), Google • Tim Allclair (https://github.com/tallclair), Google	Regular SIG Meeting: Wednesdays at 11:00 PT (Pacific Time) (biweekly)
AWS (https://github.com/kubernetes/community/blob/master/sig-aws/README.md)	• Justin Santa Barbara (https://github.com/justinsb) • Kris Nova (https://github.com/kris-nova), Heptio • Nishi Davidson (https://github.com/d-nishi), AWS	Regular SIG Meeting: Fridays at 9:00 PT (Pacific Time) (biweekly)
Azure (https://github.com/kubernetes/community/blob/master/sig-azure/README.md)	• Stephen Augustus (https://github.com/justaugustus), Red Hat • Shubheksha Jalan (https://github.com/shubheksha), Microsoft	Regular SIG Meeting: Wednesdays at 16:00 UTC (biweekly)

Big Data (`https://github.com/kubernetes/community/blob/master/sig-big-data/README.md`)	• Anirudh Ramanathan (`https://github.com/foxish`), Rockset • Erik Erlandson (`https://github.com/erikerlandson`), Red Hat • Yinan Li (`https://github.com/liyinan926`), Google	Regular SIG Meeting: Wednesdays at 17:00 UTC (weekly)

If you'd like to join one of the meetings, check out the master list here: `https://github.com/kubernetes/community/blob/master/sig-list.md`.

How to get involved

The last thing we wanted to share with you is intended to point you in the right direction so you can start to contribute directly to Kubernetes or other related software. Kubernetes has a great contributor guide, and you should consider contributing for several reasons:

- It's a great way to understand the core concepts and inner workings of Kubernetes. Writing the software of the system will give you, as an operator or developer, a unique understanding of how everything works.
- It's a fun way to meet other motivated, smart people. The world is becoming more and more interconnected, and OSS is powering some of the biggest companies in the world. Working directly on this technology will introduce you to engineers at the world's most advanced companies, and may even open to the door to significant career opportunities.
- Kubernetes, at its essence, is a community project, and relies on the contributions of its members and users. Getting involved with direct contribution of documentation updates, bug fixes, and feature creation evolves the ecosystem and provides a better experience for everyone.

If you'd like to read more about becoming a Kubernetes contributor, read more at `https://github.com/kubernetes/community/tree/master/contributors/guide/`.

Summary

In this chapter, you learned more about the Kubernetes ecosystem surrounding the Kubernetes system that we've been learning about. You've read about the core pieces of the CNCF, and we've explored the Cloud Native Trail Map to understand all of the supporting technology. We also looked at the SIGs, along with how you can start contributing to Kubernetes itself and why that's important!

Questions

1. Name at least one graduated project in the CNCF
2. Name at least three projects that are incubating in the CNCF
3. Name at least one project in the CNCF sandbox
4. What is the goal of the committee in the CNCF?
5. Why is it important to get involved with OSS development?
6. What kind of cipher material does Git contribution require?

Further reading

If you'd like to read more about how to master Git, check out the following resource from Packt Publishing: `https://www.packtpub.com/application-development/mastering-git`.

12
Cluster Federation and Multi-Tenancy

This chapter will discuss the new federation capabilities and how to use them to manage multiple clusters across cloud providers. We will also cover the federated version of the core constructs. We will walk you through federated Deployments, ReplicaSets, ConfigMaps, and Events.

This chapter will discuss the following topics:

- Federating clusters
- Federating multiple clusters
- Inspecting and controlling resources across multiple clusters
- Launching resources across multiple clusters

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in, or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes over the web: `https://labs.play-with-k8s.com/`. There's also the Katacoda playground at `https://www.katacoda.com/courses/kubernetes/playground`.

You'll also need GitHub credentials, the setting up of which we'll go over later in this chapter. Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter12`.

Introduction to federation

While federation is still very new in Kubernetes, it lays the groundwork for a highly sought after cross-cloud provider solution. Using federation, we can run multiple Kubernetes clusters on-premises and in one or more public cloud providers and manage applications utilizing the entire set of all our organizational resources.

This begins to create a path for avoiding cloud provider lock-in and highly available deployment that can place application servers in multiple clusters and allow for communication to other services located in single points among our federated clusters. We can improve isolation on outages at a particular provider or geographic location while providing greater flexibility for scaling and utilizing total infrastructure.

Currently, the federation plane supports these resources: ConfigMap, DaemonSets, Deployment, Events, Ingress, Namespaces, ReplicaSets, Secrets, and Services. Note that federation and its components are in alpha and beta phases of release, so functionality may still be a bit temperamental.

Why federation?

There are several major advantages to taking on Kubernetes cluster federation. As mentioned previously, federation allows you increase the availability and tenancy capabilities of your Kubernetes clusters. By scaling across availability zones or regions of a single **cloud service provider** (**CSP**), or by scaling across multiple CSPs, federation takes the concept of high availability to the next level. Some term this global scheduling, which will could enable you to direct traffic in order to maximize an inexpensive CSP resource that becomes available in the spot market. You could also use global scheduling to relocate workloads cluster to end use populations, improving the performance of your applications.

There is also the opportunity to treat entire clusters as if they were Kubernetes objects, and deal with failure on a per-cluster basis instead of per machine. Cluster federation could allow operators to automatically recover from entire clusters failing by routing traffic to redundant, available clusters.

It should be noted that, while federation increases the potential for high availability on your cluster, it's clear that the significant increase in complexity also lowers your potential reliability if your clusters aren't managed well. You can manage some of this complexity by using a hosted PaaS version of Kubernetes such as GKE, where leaving the cluster management to GCP will drastically lower the operational load on your teams.

Federation can also enable your team to support a hybrid environment, with on-premises clusters pairing with your resources in the cloud. Depending on your traffic routing requirements, this may require additional engineering in the form of a service mesh.

There's a number of technical features that federation supplies, which enable higher potential availability.

The building blocks of federation

Federation makes it easy to manage resources across clusters by providing two distinct types of building blocks. The first is resources and the second is service discovery:

- **Resource synchronization across clusters**: Federation is the glue that allows you to keep track of the many resources needed to run sets of applications. When you're running a lot of applications, with many resources and object types, across many clusters, federation is key to keeping your clusters organized and managed well. You may find yourself needing to keep an application deployment running in multiple clusters with a single pane of glass view.
- **Multi-cluster service discovery**: There are a number of resources that share well between clusters such as DNS, load balancers, object storage, and ingress. Federation gives you the ability to automatically configure those services with multi-cluster awareness, so you can route application traffic and manage the control plane across several clusters.

As we'll learn next, Kubernetes federation is managed by a tool named `kubefed`, which has a number of command-line flags that allow you to manage many clusters and the building blocks we discussed previously. The major building blocks of `kubefed` that we'll use are as follows:

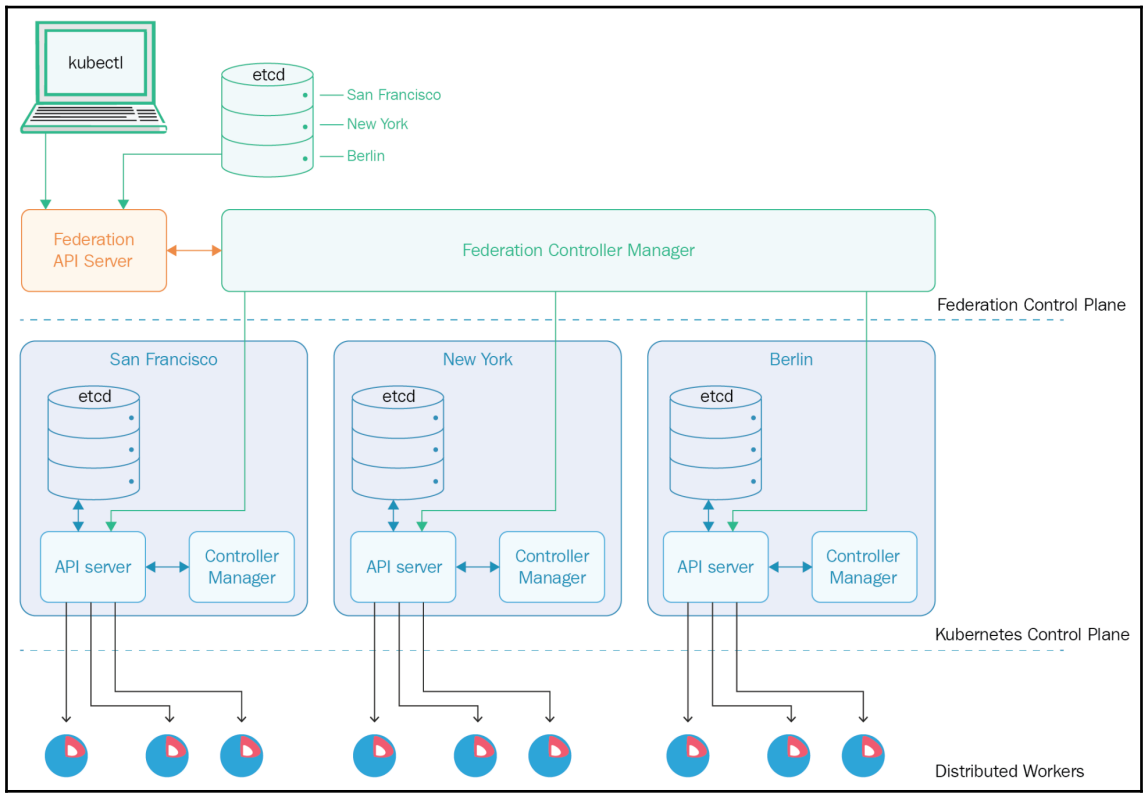
- `kubefed init`: Initialize a federation control plane
- `kubefed join`: Join a cluster to a federation
- `kubefed options`: Print the list of flags inherited by all commands
- `kubefed unjoin`: Unjoin a cluster from a federation
- `kubefed version`: Print the client and server version information

Here's a handy list of the options that can be used:

```
      --alsologtostderr                                log to standard error
as well as files
      --as string                                      Username to
impersonate for the operation
      --as-group stringArray                           Group to impersonate
for the operation, this flag can be repeated to specify multiple groups.
      --cache-dir string                               Default HTTP cache
directory (default "/Users/jrondeau/.kube/http-cache")
      --certificate-authority string                   Path to a cert file
for the certificate authority
      --client-certificate string                      Path to a client
certificate file for TLS
      --client-key string                              Path to a client key
file for TLS
      --cloud-provider-gce-lb-src-cidrs cidrs          CIDRs opened in GCE
firewall for LB traffic proxy & health checks (default
130.211.0.0/22,209.85.152.0/22,209.85.204.0/22,35.191.0.0/16)
      --cluster string                                 The name of the
kubeconfig cluster to use
      --context string                                 The name of the
kubeconfig context to use
      --default-not-ready-toleration-seconds int       Indicates the
tolerationSeconds of the toleration for notReady:NoExecute that is added by
default to every pod that does not already have such a toleration. (default
300)
      --default-unreachable-toleration-seconds int     Indicates the
tolerationSeconds of the toleration for unreachable:NoExecute that is added
by default to every pod that does not already have such a toleration.
(default 300)
  -h, --help                                           help for kubefed
      --insecure-skip-tls-verify                       If true, the server's
certificate will not be checked for validity. This will make your HTTPS
connections insecure
      --ir-data-source string                          Data source used by
InitialResources. Supported options: influxdb, gcm. (default "influxdb")
      --ir-dbname string                               InfluxDB database name
which contains metrics required by InitialResources (default "k8s")
      --ir-hawkular string                             Hawkular configuration
URL
      --ir-influxdb-host string                        Address of InfluxDB
which contains metrics required by InitialResources (default
"localhost:8080/api/v1/namespaces/kube-system/services/monitoring-
influxdb:api/proxy")
      --ir-namespace-only                              Whether the estimation
should be made only based on data from the same namespace.
      --ir-password string                             Password used for
```

```
connecting to InfluxDB (default "root")
      --ir-percentile int                          Which percentile of
samples should InitialResources use when estimating resources. For
experiment purposes. (default 90)
      --ir-user string                             User used for
connecting to InfluxDB (default "root")
      --kubeconfig string                          Path to the kubeconfig
file to use for CLI requests.
      --log-backtrace-at traceLocation             when logging hits line
file:N, emit a stack trace (default :0)
      --log-dir string                             If non-empty, write
log files in this directory
      --log-flush-frequency duration               Maximum number of
seconds between log flushes (default 5s)
      --logtostderr                                log to standard error
instead of files (default true)
      --match-server-version                       Require server version
to match client version
  -n, --namespace string                           If present, the
namespace scope for this CLI request
      --password string                            Password for basic
authentication to the API server
      --request-timeout string                     The length of time to
wait before giving up on a single server request. Non-zero values should
contain a corresponding time unit (e.g. 1s, 2m, 3h). A value of zero means
don't timeout requests. (default "0")
  -s, --server string                              The address and port
of the Kubernetes API server
      --stderrthreshold severity                   logs at or above this
threshold go to stderr (default 2)
      --token string                               Bearer token for
authentication to the API server
      --user string                                The name of the
kubeconfig user to use
      --username string                            Username for basic
authentication to the API server
  -v, --v Level                                    log level for V logs
      --vmodule moduleSpec                         comma-separated list
of pattern=N settings for file-filtered logging
```

Here's a high-level diagram that shows what all of these pieces look like when strung together:



Key components

There are two key components to the federation capability within Kubernetes. These components make up the federation control plane.

The first is `federation-controller-manager`, which embeds the core control loops required to operate federation. `federation-controller-manager` watches the state of your clusters via `apiserver` and makes changes in order to reach a desired state.

The second is `federation-apiserver`, which validates and configures Kubernetes objects such as pods, services, and controllers. `federation-apiserver` is the frontend for the cluster through which all other components interact.

Federated services

Now that we have the building blocks of federation conceptualized in our mind, let's review one more facet of this before setting up federation. How exactly does a common service, deployed across multiple clusters, work?

Federated services are created in a very similar fashion to regular services: first, by sending the desired state and properties of the service to an API endpoint, which is then brought to bear by the Kubernetes architecture. There are two main differences:

- A non-federated service will make an API call directly to a cluster API endpoint
- A federated service will make the call to the Federated API endpoint at `federation/v1beta1`, which will then redirect the API call to all of the individual clusters within the federation control plane

This second type of service allows us to extend such things as DNS service discovery across cluster boundaries. The DNS `resolv` chain is able to leverage service federation and public DNS records to resolve names across multiple clusters.

The API for a federated service is 100% compatible with regular services.

When a service is created, federation takes care of several things. First, it creates matching services in all clusters where `kubefed` specifies they reside. The health of those services is monitored so that traffic can be routed or re-routed to them. Lastly, federation ensure that there's a definitive set of public DNS records available through providers such as Route 53 or Google Cloud DNS.

Microservices residing on different pods within your Kubernetes clusters will use all of this machinery in order to locate the federated service either within their own cluster or navigate to the nearest healthy example within your federation map.

Setting up federation

While we can use the cluster we had running for the rest of the examples, I would highly recommend that you start fresh. The default naming of the clusters and contexts can be problematic for the federation system. Note that the `--cluster-context` and `--secret-name` flags are there to help you work around the default naming, but for first-time federation, it can still be confusing and less than straightforward.

Hence, starting fresh is how we will walk through the examples in this chapter. Either use new and separate cloud provider (AWS and/or GCE) accounts or tear down the current cluster and reset your Kubernetes control environment by running the following commands:

```
$ kubectl config unset contexts
$ kubectl config unset clusters
```

Double-check that nothing is listed using the following commands:

```
$ kubectl config get-contexts
$ kubectl config get-clusters
```

Next, we will want to get the `kubefed` command on our path and make it executable. Navigate back to the folder where you have the Kubernetes download extracted. The `kubefed` command is located in the `/kubernetes/client/bin` folder. Run the following commands to get in the `bin` folder and change the execution permissions:

```
$ sudo cp kubernetes/client/bin/kubefed /usr/local/bin
$ sudo chmod +x /usr/local/bin/kubefed
```

Contexts

Contexts are used by the Kubernetes control plane to keep authentication and cluster configuration stored for multiple clusters. This allows us to access and manage multiple clusters accessible from the same `kubectl`. You can always see the contexts available with the `get-contexts` command that we used earlier.

New clusters for federation

Again, make sure you navigate to wherever Kubernetes was downloaded and move into the `cluster` sub-folder:

```
$ cd kubernetes/cluster/
```

Before we proceed, make sure you have the GCE command line and the AWS command line installed, authenticated, and configured. Refer to `Chapter 1`, *Introduction to Kubernetes*, if you need assistance doing so on a new box.

First, we will create the AWS cluster. Note that we are adding an environment variable named `OVERRIDE_CONTEXT`, which will allow us to set the context name to something that complies with the DNS naming standards. DNS is a critical component for federation as it allows us to do cross-cluster discovery and service communication. This is important in a federated world where clusters may be in different data centers and even providers.

Run these commands to create your AWS cluster:

```
$ export KUBERNETES_PROVIDER=aws
$ export OVERRIDE_CONTEXT=awsk8s
$ ./kube-up.sh
```

Next, we will create a GCE cluster, once again using the `OVERRIDE_CONTEXT` environment variable:

```
$ export KUBERNETES_PROVIDER=gce
$ export OVERRIDE_CONTEXT=gcek8s
$ ./kube-up.sh
```

If we take a look at our contexts now, we will notice both `awsk8s` and `gcek8s`, which we just created. The star in front of `gcek8s` denotes that it's where `kubectl` is currently pointing and executing against:

```
$ kubectl config get-contexts
```

The preceding command should produce something like the following:

```
CURRENT   NAME          CLUSTER       AUTHINFO      NAMESPACE
          awsk8s        awsk8s        awsk8s
*         gcek8s        gcek8s        gcek8s
```

Initializing the federation control plane

Now that we have two clusters, let's set up the federation control plane in the GCE cluster. First, we'll need to make sure that we are in the GCE context, and then we will initialize the federation control plane:

```
$ kubectl config use-context gcek8s
$ kubefed init master-control --host-cluster-context=gcek8s --dns-zone-
name="mydomain.com"
```

The preceding command creates a new context just for federation called `master-control`. It uses the `gcek8s` cluster/context to host the federation components (such as API server and controller). It assumes GCE DNS as the federation's DNS service. You'll need to update `dns-zone-name` with a domain suffix you manage.

By default, the DNS provider is GCE. You can use `--dns-provider="aws-route53"` to set it to AWS `route53`; however, out of the box implementation still has issues for many users.

If we check our contexts once again, we will now see three contexts:

```
$ kubectl config get-contexts
```

The preceding command should produce something like the following:

```
CURRENT   NAME             CLUSTER          AUTHINFO         NAMESPACE
          awsk8s           awsk8s           awsk8s
*         gcek8s           gcek8s           gcek8s
          master-control   master-control   master-control
```

Let's make sure we have all of the federation components running before we proceed. The federation control plane uses the `federation-system` namespace. Use the `kubectl get pods` command with the namespace specified to monitor the progress. Once you see two API server pods and one controller pod, you should be set:

```
$ kubectl get pods --namespace=federation-system
```

```
NAME                                              READY     STATUS
RESTARTS   AGE
master-control-apiserver-3595964982-s6lx9         2/2       Running
0          8m
master-control-controller-manager-516854663-r8m37   1/1       Running
0          8m
```

Now that we have the federation components set up and running, let's switch to that context for the next steps:

```
$ kubectl config use-context master-control
```

Adding clusters to the federation system

Now that we have our federation control plane, we can add the clusters to the federation system. First, we will join the GCE cluster and then the AWS cluster:

```
$ kubefed join gcek8s --host-cluster-context=gcek8s --secret-name=fed-
secret-gce
$ kubefed join awsk8s --host-cluster-context=gcek8s --secret-name=fed-
secret-aws
```

Federated resources

Federated resources allow us to deploy across multiple clusters and/or regions. Currently, version 1.5 of Kubernetes support a number of core resource types in the federation API, including ConfigMap, DaemonSets, Deployment, Events, Ingress, Namespaces, ReplicaSets, Secrets, and Services.

Let's take a look at a federated deployment that will allow us to schedule pods across both AWS and GCE. Save the following file as `node-js-deploy-fed.yaml`:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: node-js-deploy
  labels:
    name: node-js-deploy
spec:
  replicas: 3
  template:
    metadata:
      labels:
        name: node-js-deploy
    spec:
      containers:
      - name: node-js-deploy
        image: jonbaier/pod-scaling:latest
        ports:
        - containerPort: 80
```

Create this deployment with the following command:

```
$ kubectl create -f node-js-deploy-fed.yaml
```

Now, let's try listing the pods from this deployment:

```
$ kubectl get pods
```

```
the server doesn't have a resource type "pods"
```

We should see a message like the preceding one depicted. This is because we are still using `master-control` or federation context, which does not itself run pods. We will, however, see the deployment in the federation plane and, if we inspect the events, we will see that the deployment was in fact created on both of our federated clusters:

```
$ kubectl get deployments
$ kubectl describe deployments node-js-deploy
```

We should see something like the following. Notice that the `Events:` section shows deployments in both our GCE and AWS contexts:

```
Name:                   node-js-deploy
Namespace:              default
CreationTimestamp:      Fri, 10 Mar 2017 22:15:11 +0000
Labels:                 name=node-js-deploy
Selector:               name=node-js-deploy
Replicas:               0 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  1 max unavailable, 1 max surge
Events:
   FirstSeen     LastSeen        Count   From
SubObjectPath    Type            Reason          Message
   ---------     --------        -----   ----
-------------    --------        ------          -------
   4m            4m              1       {federated-deployment-controller }
Normal           CreateInCluster Creating deployment in cluster gcek8s
   4m            4m              1       {federated-deployment-controller }
Normal           CreateInCluster Creating deployment in cluster awsk8s
```

We can also see the federated events using the following command:

```
$ kubectl get events
```

```
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND         SUBOBJECT   TYPE
    REASON             SOURCE                                  MESSAGE
10m        10m         1         node-js-deploy  Deployment               Normal
    CreateInCluster    {federated-deployment-controller }   Creating deployment in
cluster gcek8s
10m        10m         1         node-js-deploy  Deployment               Normal
    CreateInCluster    {federated-deployment-controller }   Creating deployment in
cluster awsk8s
```

It may take a moment for all three pods to run. Once that happens, we can switch to each cluster context and see some of the pods on each. Note that we can now use `get pods` since we are on the individual clusters and not on the control plane:

```
$ kubectl config use-context awsk8s
$ kubectl get pods
```

```
NAME                              READY     STATUS    RESTARTS   AGE
node-js-deploy-1713031517-1661z   1/1       Running   0          7m
```

```
$ kubectl config use-context gcek8s
$ kubectl get pods
```

```
NAME                              READY     STATUS    RESTARTS   AGE
node-js-deploy-1713031517-bvdmf   1/1       Running   0          7m
node-js-deploy-1713031517-jnfnr   1/1       Running   0          7m
```

We should see the three pods spread across the clusters with two on one and a third on the other. Kubernetes has spread them across the cluster without any manual intervention. Any pods that fail will be restarted, but now we have the added redundancy of two cloud providers.

Federated configurations

In modern software development, it is common to separate configuration variables from the application code itself. In this way, it is easier to make updates to service URLs, credentials, common paths, and so on. Having these values in external configuration files means we can easily update configuration without rebuilding the entire application.

This separation solves the initial problem, but true portability comes when you can remove the dependency from the application completely. Kubernetes offers a configuration store for exactly this purpose. ConfigMaps are simple constructs that store key-value pairs.

Kubernetes also supports Secrets for more sensitive configuration data. This will be covered in more detail in `Chapter 10`, *Cluster Authentication, Authorization, and Container Security*. You can use the example there in both single clusters or on the federation control plane as we are demonstrating with ConfigMaps here.

Let's take a look at an example that will allow us to store some configuration and then consume it in various pods. The following listings will work for both federated and single clusters, but we will continue using a federated setup for this example.

The `ConfigMap` kind can be created using literal values, flat files and directories, and finally YAML definition files. The following listing is a YAML definition of the `configmap-fed.yaml` file:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-application-config
  namespace: default
data:
  backend-service.url: my-backend-service
```

Let's first switch back to our federation plane:

```
$ kubectl config use-context master-control
```

Now, create this listing with the following command:

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

Let's display the `configmap` object that we just created. The `-o yaml` flag helps us to display the full information:

```
$ kubectl get configmap my-application-config -o yaml
```

```
apiVersion: v1
data:
  backend-service.url: my-backend-service
kind: ConfigMap
metadata:
  creationTimestamp: 2017-03-10T22:28:38Z
  name: my-application-config
  namespace: default
  resourceVersion: "1959"
  selfLink: /api/v1/namespaces/default/configmaps/my-application-config
  uid: e85a0028-05e0-11e7-bdf8-42010a800002
```

Now that we have a `ConfigMap` object, let's start up a federated `ReplicaSet` that can use the `ConfigMap` object. This will create replicas of pods across our cluster that can access the `ConfigMap` object. `ConfigMaps` can be accessed via environment variables or mount volumes. This example will use a mount volume that provides a folder hierarchy and the files for each key with the contents representing the values. Save the following file as `configmap-rs-fed.yaml`:

```
apiVersion: extensions/v1beta1
kind: ReplicaSet
metadata:
  name: node-js-rs
spec:
  replicas: 3
  selector:
    matchLabels:
      name: node-js-configmap-rs
  template:
    metadata:
      labels:
        name: node-js-configmap-rs
    spec:
      containers:
      - name: configmap-pod
        image: jonbaier/node-express-info:latest
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: configmap-volume
          mountPath: /etc/config
```

```
volumes:
      - name: configmap-volume
        configMap:
          name: my-application-config
```

Create this pod with `kubectl create -f configmap-rs-fed.yaml`. After creation, we will need to switch contexts to one of the clusters where the pods are running. You can choose either, but we will use the GCE context here:

```
$ kubectl config use-context gcek8s
```

Now that we are on the GCE cluster specifically, let's check `configmaps` here:

```
$ kubectl get configmaps
```

As you can see, the `ConfigMap` is propagated locally to each cluster. Next, let's find a pod from our federated `ReplicaSet`:

```
$ kubectl get pods
```

```
NAME                                READY     STATUS    RESTARTS   AGE
node-js-deploy-1713031517-cmd7q     1/1       Running   0          39m
node-js-deploy-1713031517-zncxr     1/1       Running   0          39m
node-js-rs-6g7nj                    1/1       Running   0          9m
node-js-rs-f4w7b                    1/1       Running   0          9m
```

Let's take one of the `node-js-rs` pod names from the listing and run a bash shell with `kubectl exec`:

```
$ kubectl exec -it node-js-rs-6g7nj bash
```

Then, let's change directories to the `/etc/config` folder that we set up in the pod definition. Listing this directory reveals a single file with the name of the `ConfigMap` we defined earlier:

```
$ cd /etc/config
$ ls
```

If we then display the contents of the files with the following command, we should see the value we entered earlier, `my-backend-service`:

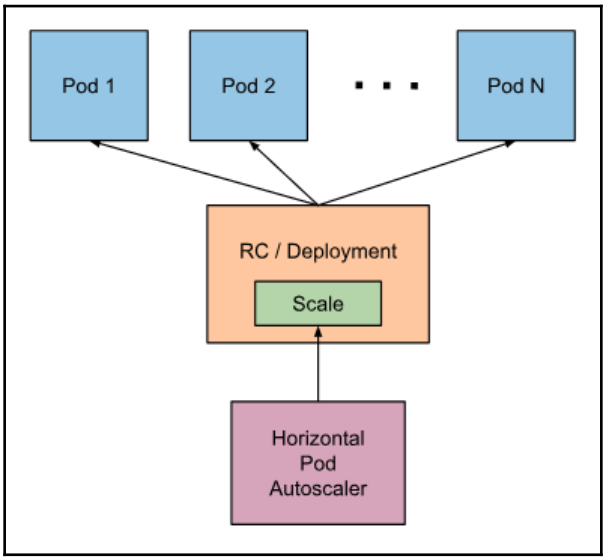
```
$ echo $(cat backend-service.url)
```

If we were to look in any of the pods across our federated cluster, we would see the same values. This is a great way to decouple configuration from an application and distribute it across our fleet of clusters.

Federated horizontal pod autoscalers

Let's look at another example of a newer resource that you can use with the federated model: **horizontal pod autoscalers** (**HPAs**).

Here's what the architecture of these looks like in a single cluster:



Credit: `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#how-does-the-horizontal-pod-autoscaler-work`.

These HPAs will act in a similar fashion to normal HPAs, with the same functionality and same API-based compatibility—only, with federation, the management will traverse your clusters. This is an alpha feature, so it is not enabled by default on your cluster. In order to enable it, you'll need to run `federation-apiserver` with the `--runtime-config=api/all=true` option. Currently, the only metrics that work to manage HPAs are CPU utilization metrics.

First, let's create a file that contains the HPA configuration, called `node-hpa-fed.yaml`:

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
 name: nodejs
 namespace: default
spec:
 scaleTargetRef:
   apiVersion: apps/v1beta1   kind: Deployment
name: nodejs
 minReplicas: 5
 maxReplicas: 20
 targetCPUUtilizationPercentage: 70
```

We can add this to our cluster with the following command:

```
kubectl --context=federation-cluster create -f node-hpa-fed.yaml
```

In this case, `--context=federation-cluster` is telling `kubectl` to send the request to `federation-apiserver` instead of `kube-apiserver`.

If, for example, you wanted to restrict this HPA to a subset of your Kubernetes clusters, you can use cluster selectors to restrict the federated object by using the `federation.alpha.kubernetes.io/cluster-selector` annotation. It's similar in function to nodeSelector, but acts upon full Kubernetes clusters. Cool! You'll need to create an annotation in JSON format. Here's a specific example of a ClusterSelector annotation:

```
metadata:
  annotations:
     federation.alpha.kubernetes.io/cluster-selector: '[{"key": "hipaa",
"operator":
       "In", "values": ["true"]}, {"key": "environment", "operator":
"NotIn", "values": ["nonprod"]}]'
```

This example is going to keep workloads with the `hipaa` label out of environments with the `nonprod` label.

For a full list of Top Level Federation API objects, see the following: `https://kubernetes.io/docs/reference/federation/`

You can check your clusters to see whether the HPA was created in an individual location by specifying the context:

```
kubectl --context=gce-cluster-01 get HPA nodejs
```

Once you're finished with the HPA, it can be deleted with the following `kubectl` command:

```
kubectl --context=federation-cluster delete HPA nodejs
```

How to use federated HPAs

HPAs used in the previous manner are an essential tool for ensuring that your clusters scale up as their workloads increase. The default behavior for HPA spreading in clusters ensure that maximum replicas are spread evenly first in all clusters. Let's say that you have 10 registered Kubernetes clusters in your federation control plane. If you have `spec.maxReplicas = 30`, each of the clusters will receive the following HPA `spec`:

```
spec.maxReplicas = 10
```

If you were to then set `spec.minReplicas = 5`, then some of the clusters will receive the following:

```
spec.minReplicas = 1
```

This is due to being unable to have a replica sum of 0. It's important to note that federation manipulates the minx/mix replicas it creates on the federated clusters, not by directly monitoring the target object metrics (in our case, CPU). The federated HPA controller is relying on HPAs within the federated cluster to monitor CPU utilization, which then makes changes to specs such as current and desired replicas.

Other federated resources

So far, we have seen federated Deployments, ReplicaSets, Events, and ConfigMaps in action. DaemonSets, Ingress, Namespaces, Secrets, and Services are also supported. Your specific setup will vary and you may have a set of clusters that differ from our example here. As mentioned earlier, these resources are still in beta, so it's worth spending some time to experiment with the various resource types and understand how well the federation constructs are supported for your particular mix of infrastructure.

Let's look at some examples that we can use to leverage other common Kubernetes API objects from a federated perspective.

Events

If you want to see what events are only stored in the federation control plane, you can use the following command:

```
kubectl --context=federation-cluster get events
```

Jobs

When you go to create a job, you'll use similar concepts as before. Here's what that looks like when you create a job within the federation context:

```
kubectl --context=federation-cluster create -f fedjob.yaml
```

You can get the list of these jobs within the federated context with the following:

```
kubectl --context=gce-cluster-01 get job fedjob
```

As with HPAs, you can spread your jobs across multiple underlying clusters with the appropriate specs. The relevant definitions are `spec.parallelism` and `spec.completions`, and they can be modified by specifying the correct `ReplicaAllocationPreferences` with the `federation.kubernetes.io/job-preferences` key.

True multi-cloud

This is an exciting space to watch. As it grows, it gives us a really good start to doing multi-cloud implementations and providing redundancy across regions, data centers, and even cloud providers.

While Kubernetes does provide an easy and exciting path to multi-cloud infrastructure, it's important to note that production multi-cloud requires much more than distributed deployments. A full set of capabilities from logging and monitoring to compliance and host-hardening, there is much to manage in a multi-provider setup.

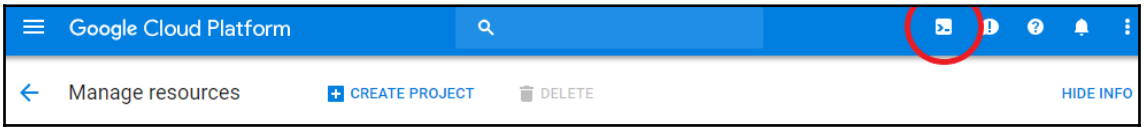
True multi-cloud adoption will require a well-planned architecture, and Kubernetes takes a big step forward in pursuing this goal.

Getting to multi-cloud

In this exercise, we're going to unite two clusters using Istio's multi-cloud feature. Normally, we'd create two clusters from scratch, across two CSPs, but for the purposes of exploring one single isolated concept at a time, we're going to use the GKE to spin up our clusters, so we can focus on the inner workings of Istio's multi-cloud functionality.

Let's get started by logging in to your Google Cloud Project! First, you'll want to create a project in the GUI called `gsw-k8s-3`, if you haven't already, and get your Google Cloud Shell to point to it. If you're already pointed at your GCP account, you can disregard that.

Click this button for an easy way to get access to the CLI tools:



Once you've launched the shell, you can point it to your project:

```
anonymuse@cloudshell:~$ gcloud config set project gsw-k8s-3
Updated property [core/project].
anonymuse@cloudshell:~ (gsw-k8s-3)$
```

Next, we'll set up an environment variable for the project ID, which can echo back to see:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ proj=$(gcloud config list --
format='value(core.project)')
anonymuse@cloudshell:~ (gsw-k8s-3)$ echo $proj
Gsw-k8s-3
```

Now, let's create some clusters. Set some variables for the zone and cluster name:

```
zone="us-east1-b"
cluster="cluster-1"
```

First, create cluster one:

```
gcloud container clusters create $cluster --zone $zone --username "
 --cluster-version "1.10.6-gke.2" --machine-type "n1-standard-2" --image-
type "COS" --disk-size "100" \
 --scopes gke-default \
 --num-nodes "4" --network "default" --enable-cloud-logging --enable-cloud-
monitoring --enable-ip-alias --async

WARNING: Starting in 1.12, new clusters will not have a client certificate
issued. You can manually enable (or disable) the issuance of the client
```

```
certificate using the `--[no-]issue-client-certificate` flag. This will
enable the autorepair feature for nodes. Please see
https://cloud.google.com/kubernetes-engine/docs/node-auto-repair for more
information on node autorepairs.

WARNING: Starting in Kubernetes v1.10, new clusters will no longer get
compute-rw and storage-ro scopes added to what is specified in --scopes
(though the latter will remain included in the default --scopes). To use
these scopes, add them explicitly to --scopes. To use the new behavior, set
container/new_scopes_behavior property (gcloud config set
container/new_scopes_behavior true).

NAME       TYPE LOCATION    TARGET STATUS_MESSAGE  STATUS START_TIME
END_TIME
cluster-1        us-east1-b                       PROVISIONING
```

You may need to change the cluster version to a newer GKE version as updates are made. Older versions become unsupported over time. For example, you might see a message such as this:

```
ERROR: (gcloud.container.clusters.create) ResponseError:
code=400, message=EXTERNAL: Master version "1.9.6-gke.1"
is unsupported.
```

You can check this web page to find out the currently supported version of GKE: `https://cloud.google.com/kubernetes-engine/release-notes`.

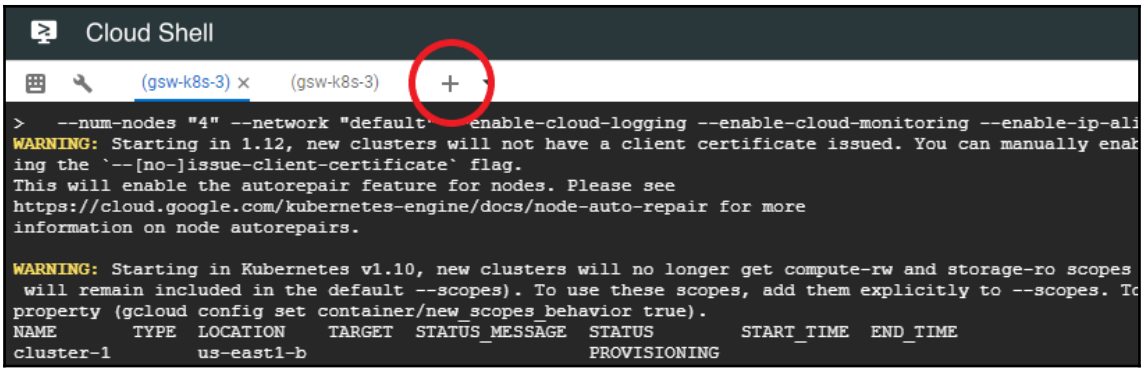
Next, specify `cluster-2`:

```
cluster="cluster-2"
```

Now, create it, where you'll see messages above. We'll omit them this time around:

```
gcloud container clusters create $cluster --zone $zone --username "admin" \
--cluster-version "1.10.6-gke.2" --machine-type "n1-standard-2" --image-
type "COS" --disk-size "100" \
 --scopes gke-default \
 --num-nodes "4" --network "default" --enable-cloud-logging --enable-cloud-
monitoring --enable-ip-alias --async
```

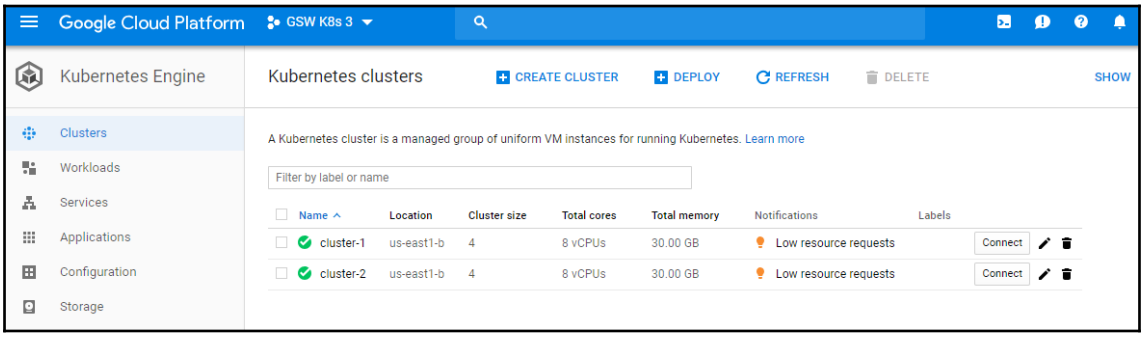
You'll see the same messaging above. You can create another Google Cloud Shell window by clicking on the **+** icon in order to create some `watch` commands to see the clusters created. Take a minute to do this while the instances are created:



In that window, launch this command: `gcloud container clusters list`. You should see the following:

```
gcloud container clusters list
<snip>
Every 1.0s: gcloud container clusters list
cs-6000-devshell-vm-375db789-dcd6-42c6-b1a6-041afea68875: Mon Sep 3
12:26:41 2018

NAME       LOCATION MASTER_VERSION  MASTER_IP MACHINE_TYPE  NODE_VERSION
NUM_NODES STATUS
cluster-1  us-east1-b 1.10.6-gke.2    35.237.54.93 n1-standard-2  1.10.6-
gke.2 4 RUNNING
cluster-2  us-east1-b 1.10.6-gke.2    35.237.47.212 n1-standard-2  1.10.6-
gke.2 4 RUNNING
```

On the dashboard, it'll look like so:



Next up, we'll grab the cluster credentials. This command will allow us to set a `kubeconfig` context for each specific cluster:

```
for clusterid in cluster-1 cluster-2; do gcloud container clusters get-
credentials $clusterid --zone $zone; done
Fetching cluster endpoint and auth data.
kubeconfig entry generated for cluster-1.
Fetching cluster endpoint and auth data.
kubeconfig entry generated for cluster-2.
```

Let's ensure that we can use `kubectl` to get the context for each cluster:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl config use-context
"gke_${proj}_${zone}_cluster-1"
Switched to context "gke_gsw-k8s-3_us-east1-b_cluster-1".
```

If you then run `kubectl get pods --all-namespaces` after executing each of the cluster context switches, you should see something similar to this for each cluster:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl get pods --all-namespaces
NAMESPACE NAME READY STATUS RESTARTS AGE
kube-system event-exporter-v0.2.1-5f5b89fcc8-2qj5c 2/2 Running 0 14m
kube-system fluentd-gcp-scaler-7c5db745fc-qxqd4 1/1 Running 0 13m
kube-system fluentd-gcp-v3.1.0-g5v24 2/2 Running 0 13m
kube-system fluentd-gcp-v3.1.0-qft92 2/2 Running 0 13m
kube-system fluentd-gcp-v3.1.0-v572p 2/2 Running 0 13m
kube-system fluentd-gcp-v3.1.0-z5wjs 2/2 Running 0 13m
kube-system heapster-v1.5.3-5c47587d4-4fsg6 3/3 Running 0 12m
kube-system kube-dns-788979dc8f-k5n8c 4/4 Running 0 13m
kube-system kube-dns-788979dc8f-ldxsw 4/4 Running 0 14m
kube-system kube-dns-autoscaler-79b4b844b9-rhxdt 1/1 Running 0 13m
kube-system kube-proxy-gke-cluster-1-default-pool-e320df41-4mnm 1/1 Running
0 13m
kube-system kube-proxy-gke-cluster-1-default-pool-e320df41-536s 1/1 Running
0 13m
kube-system kube-proxy-gke-cluster-1-default-pool-e320df41-9gqj 1/1 Running
0 13m
kube-system kube-proxy-gke-cluster-1-default-pool-e320df41-t4pg 1/1 Running
0 13m
kube-system l7-default-backend-5d5b9874d5-n44q7 1/1 Running 0 14m
kube-system metrics-server-v0.2.1-7486f5bd67-h9fq6 2/2 Running 0 13m
```

Next up, we're going to need to create a Google Cloud firewall rule so each cluster can talk to the other. We're going to need to gather all cluster networking data (tags and CIDR), and then create firewall rules with `gcloud`. The CIDR ranges will look something like this:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ gcloud container clusters list --
format='value(clusterIpv4Cidr)'
10.8.0.0/14
10.40.0.0/14
```

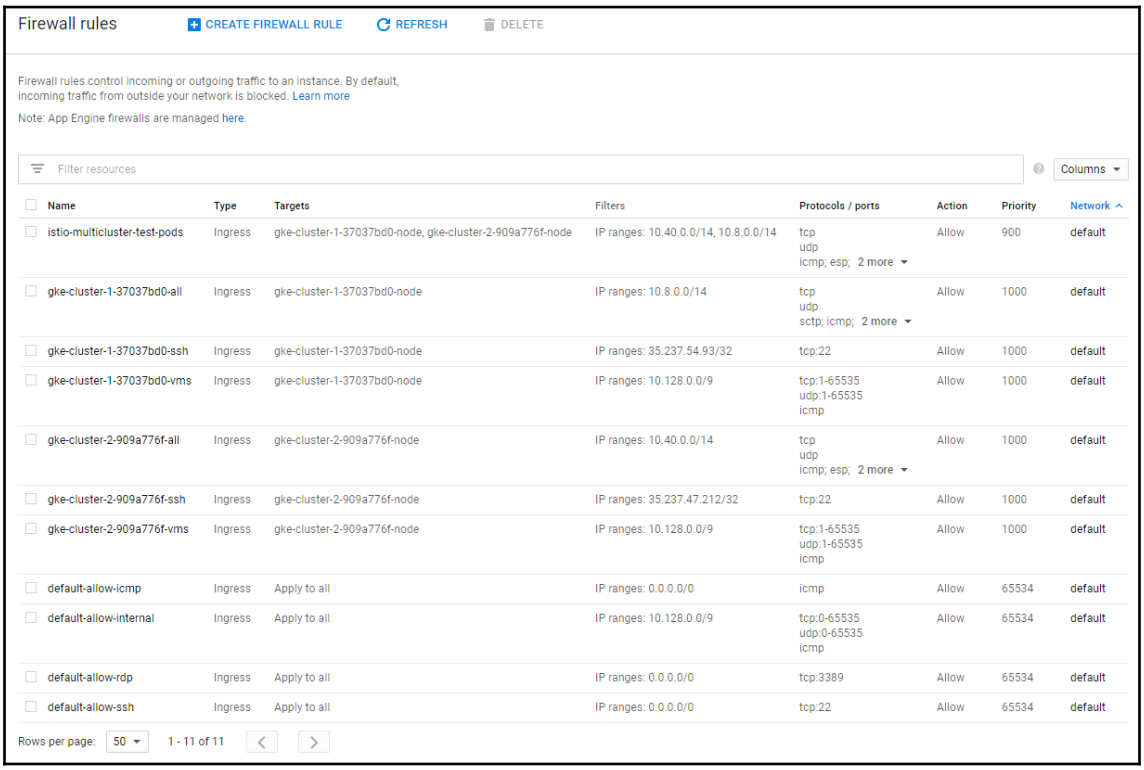
The tags will be per-node, resulting in eight total tags:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ gcloud compute instances list --
format='value(tags.items.[0])'
gke-cluster-1-37037bd0-node
gke-cluster-1-37037bd0-node
gke-cluster-1-37037bd0-node
gke-cluster-1-37037bd0-node
gke-cluster-2-909a776f-node
gke-cluster-2-909a776f-node
gke-cluster-2-909a776f-node
gke-cluster-2-909a776f-node
```

Let's run the full command now to create the firewall rules. Note the `join_by` function is a neat hack that allows us to join multiple elements of an array in Bash:

```
function join_by { local IFS="$1"; shift; echo "$*"; }
ALL_CLUSTER_CIDRS=$(gcloud container clusters list --
format='value(clusterIpv4Cidr)' | sort | uniq)
echo $ALL_CLUSTER_CDIRS
ALL_CLUSTER_CIDRS=$(join_by , $(echo "${ALL_CLUSTER_CIDRS}"))
echo $ALL_CLUSTER_CDIRS
ALL_CLUSTER_NETTAGS=$(gcloud compute instances list --
format='value(tags.items.[0])' | sort | uniq)
echo $ALL_CLUSTER_NETTAGS
ALL_CLUSTER_NETTAGS=$(join_by , $(echo "${ALL_CLUSTER_NETTAGS}"))
echo $ALL_CLUSTER_NETTAGS
gcloud compute firewall-rules create istio-multicluster-test-pods \
 --allow=tcp,udp,icmp,esp,ah,sctp \
 --direction=INGRESS \
 --priority=900 \
 --source-ranges="${ALL_CLUSTER_CIDRS}" \
 --target-tags="${ALL_CLUSTER_NETTAGS}"
```

That will set up our security firewall rules, which should look similar to this in the GUI when complete:



Let's create an admin role that we can use in future steps. First, set `KUBE_USER` to the email address associated with your GCP account with `KUBE_USER="<YOUR_EMAIL>"`. Next, we'll create a `clusterrolebinding`:

```
kubectl create clusterrolebinding gke-cluster-admin-binding \
 --clusterrole=cluster-admin \
 --user="${KUBE_USER}"
clusterrolebinding "gke-cluster-admin-binding" created
```

Next up, we'll install the Istio control plane with Helm, create a namespace, and deploy Istio using a chart.

Check to make sure you're using `cluster-1` as your context with `kubectl config current-context`. Next, we'll install Helm with these commands:

```
curl https://raw.githubusercontent.com/kubernetes/helm/master/scripts/get > 
get_helm.sh
 chmod 700 get_helm.sh
./get_helm.sh
Create a role for tiller to use. Youll need to clone the Istio repo first:
git clone https://github.com/istio/istio.git && cd istio
Now, create a service account for tiller.
kubectl apply -f install/kubernetes/helm/helm-service-account.yaml
And then we can intialize Tiller on the cluster.
/home/anonymuse/.helm
Creating /home/anonymuse/.helm/repository
...
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see:
https://docs.helm.sh/using_helm/#securing-your-helm-installation
Happy Helming!
anonymuse@cloudshell:~/istio (gsw-k8s-3)$
```

Now, switch to another, Istio-specific context where we'll install Istio in its own namespace:

```
kubectl config use-context "gke_${proj}_${zone}_cluster-1"
```

Copy over the installation chart for Istio into our home directory:

```
helm template install/kubernetes/helm/istio --name istio --namespace istio-
system > $HOME/istio_master.yaml
```

Create a namespace for it to be used in, install it, and enable injection:

```
kubectl create ns istio-system \
 && kubectl apply -f $HOME/istio_master.yaml \
 && kubectl label namespace default istio-injection=enabled
```

We'll now set some more environment variables to collect the IPs of our pilot, statsD, policy, and telemetry pods:

```
export PILOT_POD_IP=$(kubectl -n istio-system get pod -l istio=pilot -o 
jsonpath='{.items[0].status.podIP}')
export POLICY_POD_IP=$(kubectl -n istio-system get pod -l istio=mixer -o 
jsonpath='{.items[0].status.podIP}')
export STATSD_POD_IP=$(kubectl -n istio-system get pod -l istio=statsd-
prom-bridge -o jsonpath='{.items[0].status.podIP}')
export TELEMETRY_POD_IP=$(kubectl -n istio-system get pod -l istio-mixer-
type=telemetry -o jsonpath='{.items[0].status.podIP}')
```

We can now generate a manifest for our remote cluster, `cluster-2`:

```
helm template install/kubernetes/helm/istio-remote --namespace istio-system
\
 --name istio-remote \
 --set global.remotePilotAddress=${PILOT_POD_IP} \
 --set global.remotePolicyAddress=${POLICY_POD_IP} \
 --set global.remoteTelemetryAddress=${TELEMETRY_POD_IP} \
 --set global.proxy.envoyStatsd.enabled=true \
 --set global.proxy.envoyStatsd.host=${STATSD_POD_IP} > $HOME/istio-
remote.yaml
```

Now, we'll instill the minimal Istio components and sidecar inject in our target, `cluster-2`. Run the following commands in order:

```
kubectl config use-context "gke_${proj}_${zone}_cluster-2"
kubectl create ns istio-system
kubectl apply -f $HOME/istio-remote.yaml
kubectl label namespace default istio-injection=enabled
```

Now, we'll create more scaffolding to take advantage of the features of Istio. We'll need to create a file in which we can configure `kubeconfig` to work with Istio. First, change back into your home directory with `cd`. The `--minify` flag will ensure that you only see output associated with your current context. Now, enter the following groups of commands:

```
export WORK_DIR=$(pwd)
CLUSTER_NAME=$(kubectl config view --minify=true -o
"jsonpath={.clusters[].name}")
CLUSTER_NAME="${CLUSTER_NAME##*_}"
export KUBECFG_FILE=${WORK_DIR}/${CLUSTER_NAME}
SERVER=$(kubectl config view --minify=true -o
"jsonpath={.clusters[].cluster.server}")
NAMESPACE=istio-system
SERVICE_ACCOUNT=istio-multi
SECRET_NAME=$(kubectl get sa ${SERVICE_ACCOUNT} -n ${NAMESPACE} -o
jsonpath='{.secrets[].name}')
CA_DATA=$(kubectl get secret ${SECRET_NAME} -n ${NAMESPACE} -o
"jsonpath={.data['ca\.crt']}")
TOKEN=$(kubectl get secret ${SECRET_NAME} -n ${NAMESPACE} -o
"jsonpath={.data['token']}" | base64 --decode)
```

Create a file with the following `cat` command. This will inject the contents here into a file that's going to be located in `~/${WORK_DIR}/{CLUSTER_NAME}`:

```
cat <<EOF > ${KUBECFG_FILE}
apiVersion: v1
clusters:
```

```
 - cluster:
     certificate-authority-data: ${CA_DATA}
     server: ${SERVER}
   name: ${CLUSTER_NAME}
contexts:
   - context:
       cluster: ${CLUSTER_NAME}
       user: ${CLUSTER_NAME}
   name: ${CLUSTER_NAME}
current-context: ${CLUSTER_NAME}
kind: Config
preferences: {}
users:
   - name: ${CLUSTER_NAME}
     user:
       token: ${TOKEN}
EOF
```

Next up, we'll create a secret so that the control plane for Istio that exists on `cluster-1` can access `istio-pilot` on `cluster-2`. Switch back to the first cluster, create a Secret, and label it:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl config use-context gke_gsw-
k8s-3_us-east1-b_cluster-1
Switched to context "gke_gsw-k8s-3_us-east1-b_cluster-1".
kubectl create secret generic ${CLUSTER_NAME} --from-file ${KUBECFG_FILE} -
n ${NAMESPACE}
kubectl label secret ${CLUSTER_NAME} istio/multiCluster=true -n
${NAMESPACE}
```

Once we've completed these tasks, let's use all of this machinery to deploy one of Google's code examples, `bookinfo`, across both clusters. Run this on the first:

```
kubectl config use-context "gke_${proj}_${zone}_cluster-1"
kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml
kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml
kubectl delete deployment reviews-v3
```

Now, create a file called `reviews-v3.yaml` for deploying `bookinfo` to the remote cluster. The file contents can be found in the repository directory of this chapter:

```
##################################################################################
######################
# Ratings service
##################################################################################
######################
apiVersion: v1
kind: Service
```

```
metadata:
 name: ratings
 labels:
 app: ratings
spec:
 ports:
 - port: 9080
 name: http
---
##################################################################################
########################
# Reviews service
##################################################################################
########################
apiVersion: v1
kind: Service
metadata:
 name: reviews
 labels:
 app: reviews
spec:
 ports:
 - port: 9080
 name: http
 selector:
 app: reviews
---
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: reviews-v3
spec:
 replicas: 1
 template:
 metadata:
 labels:
 app: reviews
 version: v3
 spec:
 containers:
 - name: reviews
 image: istio/examples-bookinfo-reviews-v3:1.5.0
 imagePullPolicy: IfNotPresent
 ports:
 - containerPort: 9080
```

Let's install this deployment on the remote cluster, `cluster-2`:

```
kubectl config use-context "gke_${proj}_${zone}_cluster-2"
kubectl apply -f $HOME/reviews-v3.yaml
```

Once this is complete, you'll need to get access to the external IP of Istio's `isto-ingressgateway` service, in order to view the data in the `bookinfo` homepage. You can run this command to open that up. You'll need to reload that page dozens of times in order to see Istio's load balancing take place. You can hold down *F5* in order to reload the page many times.

You can access `http://<GATEWAY_IP>/productpage` in order to see the reviews.

Deleting the cluster

In order to clean up the control panel once you're finished, you can run the following commands.

First, delete the firewall rules:

```
gcloud compute firewall-rules delete istio-multicluster-test-pods
The following firewalls will be deleted:
 - [istio-multicluster-test-pods]
Do you want to continue (Y/n)? y
Deleted
[https://www.googleapis.com/compute/v1/projects/gsw-k8s-3/global/firewalls/
istio-multicluster-test-pods].
anonymuse@cloudshell:~ (gsw-k8s-3)$
```

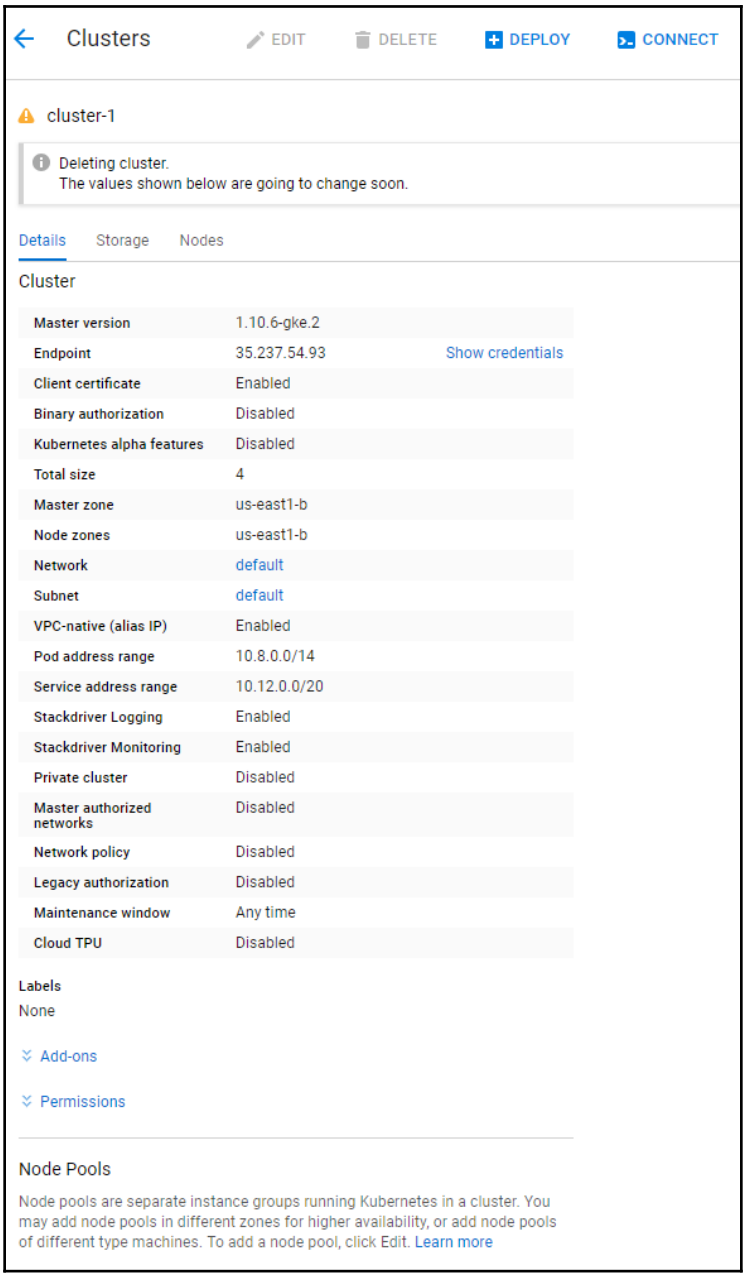
Next up, we'll delete our cluster-admin-role binding:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ kubectl delete clusterrolebinding gke-
cluster-admin-bindingclusterrolebinding "gke-cluster-admin-binding" deleted
anonymuse@cloudshell:~ (gsw-k8s-3)$
```

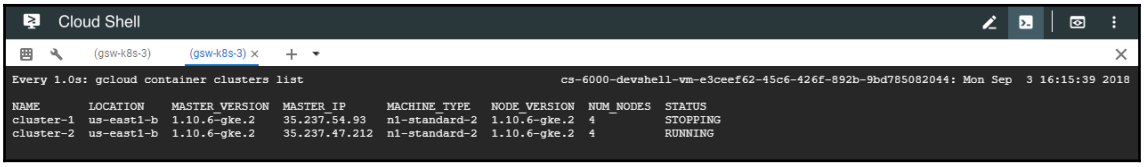
Lastly, let's delete our GKE clusters:

```
anonymuse@cloudshell:~ (gsw-k8s-3)$ gcloud container clusters delete
cluster-1 --zone $zone
The following clusters will be deleted. - [cluster-1] in [us-east1-b]
Do you want to continue (Y/n)? y
Deleting cluster cluster-1...done.
Deleted
[https://container.googleapis.com/v1/projects/gsw-k8s-3/zones/us-east1-b/cl
usters/cluster-1].
anonymuse@cloudshell:~ (gsw-k8s-3)
```

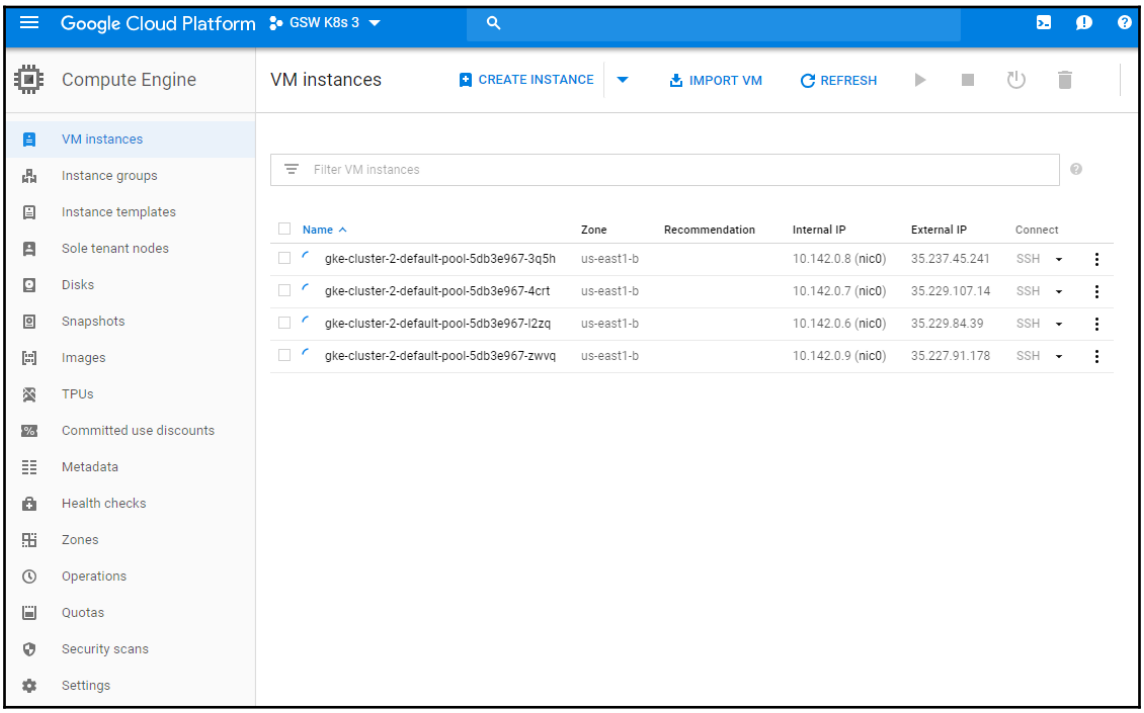
In the GUI, you can see the cluster being deleted:



You can also see it on the command line from your `watch` command:



Run the same command with your other cluster. You can double-check the **Compute Engine** dashboard to ensure that your instances are being deleted:



Summary

In this chapter, we looked at the new federation capabilities in Kubernetes. We saw how we can deploy clusters to multiple cloud providers and manage them from a single control plane. We also deployed an application across clusters in both AWS and GCE. While these features are new and still mainly in alpha and beta, we should now have the skills to utilize them as they evolve and become part of the standard Kubernetes operating model.

In the next chapter, we will take a look at another advanced topic: security. We will cover the basics for secure containers and also how to secure your Kubernetes cluster. We will also look at the Secrets construct, which gives us the capability to store sensitive configuration data similar to our preceding `ConfigMap` example.

Questions

1. What is the main goal of federation?
2. What is the main advantage of using federation?
3. What are the building blocks of federation?
4. What is the Kubernetes CLI command that controls federation?
5. What are the two software components of Kubernetes federation?
6. What is the main difference between HPAs and federated HPAs?
7. What types of federated resources are available?

Further reading

If you'd like more information on mastering Kubernetes, check out another excellent Packt resource called *Mastering Kubernetes* (`https://www.packtpub.com/application-development/mastering-kubernetes-second-edition`).

13
Cluster Authentication, Authorization, and Container Security

This chapter will discuss the basics of container security from the container runtime level to the host itself. We will discuss how to apply these concepts to workloads running in a Kubernetes cluster and some of the security concerns and practices that relate specifically to running your Kubernetes cluster.

This chapter will discuss the following topics:

- Basic container security
- Container image security and continuous vulnerability scanning
- Kubernetes cluster security
- Kubernetes secrets

Basics of container security

Container security is a deep subject area and in itself can fill its own book. Having said this, we will cover some of the high-level concerns and give you a starting point so that you can start thinking about this area.

In the *A brief overview of containers* section of `Chapter 1`, *Introduction to Kubernetes*, we looked at some of the core isolation features in the Linux kernel that enable container technology. Understanding the details of how containers work is the key to grasping the various security concerns in managing them.

A good paper to dive deeper is *NCC's Whitepaper*, *Understanding and Hardening Linux Containers*. In *section 7*, the paper explores the various attack vectors of concern for container deployments, which I will summarize.

Keeping containers contained

One of the most obvious features that is discussed in the paper we mentioned in the preceding section is that of escaping the isolation/virtualization of the container construct. Modern container implementations guard against using namespaces to isolate processes as well as allowing the control of Linux capabilities that are available to a container. Additionally, there is an increased move toward secure default configurations of the out-of-the-box container environment. For example, by default, Docker only enables a small set of capabilities. Networking is another avenue of escape and it can be challenging since there are a variety of network options that plug into most modern container setups.

The next area discussed in the paper is that of attacks between two containers. The *User* namespace model gives us added protection here by mapping the root user within the container to a lower-level user on the host machine. Networking is, of course, still an issue, and something that requires proper diligence and attention when selecting and implementing your container networking solution.

Attacks within the container itself are another vector and, as with previous concerns, namespaces and networking are key to protection here. Another aspect that is vital in this scenario is the application security itself. The code still needs to follow secure coding practices and the software should be kept up to date and patched regularly. Finally, the efficiency of container images has an added benefit of shrinking the attack surface. The images should be built with only the packages and software that's necessary.

Resource exhaustion and orchestration security

Similar to the **denial-of-service** (**DoS**) attacks, we've seen in various other areas of computing that resource exhaustion is very much a pertinent concern in the container world. While cgroups provide some limitations on resource usage for things such as CPU, memory, and disk usage, there are still valid attack avenues for resource exhaustion. Tools such as Docker offer some starting defaults to the cgroups limitations, and Kubernetes also offers additional limits that can be placed on groups of containers running in the cluster. It's important to understand these defaults and to adjust for your deployments.

While the Linux kernel and the features that enable containers give us some form of isolation, they are fairly new to the Linux operating system. As such, they still contain their own bugs and vulnerabilities. The built-in mechanisms for capabilities and namespaces can and do have issues, and it is important to track these as part of your secure container operations.

The final area covered in the NCC paper is the attack of the container management layer itself. The Docker engine, image repositories, and orchestration tools are all significant vectors of attack and should be considered when developing your strategy. We'll look in more depth at how we can address the repositories and Kubernetes as an orchestration layer in the following sections.

If you're interested in knowing more about the specific security features of Docker's implementation, take a look here: `https://docs.docker.com/engine/security/security/`.

Image repositories

Vulnerability management is a critical component of any modern day IT operation. Zero-day vulnerabilities are on the rise and even those vulnerabilities with patches can be cumbersome to remediate. First, application owners must be made aware of their vulnerabilities and potential patches. Then, these patches must be integrated into systems and code, and often this requires additional deployments or maintenance windows. Even when there is visibility to vulnerabilities, there is often a lag in remediation, often taking large organizations several months to patch.

While containers greatly improve the process of updating applications and minimizing downtime, there still remains a challenge that's inherent in vulnerability management. Especially since an attacker only needs to expose one such vulnerability, making anything less than 100% of the systems patched is a risk of compromise.

What's needed is a faster feedback loop in addressing vulnerabilities. Continuous scanning and tying into the software deployment life cycle is key to speeding up the information and remediation of vulnerabilities. Luckily, this is exactly the approach that's being built into the latest container management and security tooling.

Continuous vulnerability scanning

One such open source project that has emerged in this space is **clair**. clair is an open source project for the static analysis of vulnerabilities in appc (`https://github.com/appc/spec`) and Docker (`https://github.com/moby/moby/blob/master/image/spec/v1.md`) containers.

You can visit clair at the following link: `https://github.com/coreos/clair`.

clair scans your code against **Common Vulnerabilities and Exploits** (**CVEs**). It can be integrated into your CI/CD pipeline and run as a response to new builds. If vulnerabilities are found, they can be taken as feedback into the pipeline, even stop deployment, and fail the build. This forces developers to be aware of and remediate vulnerabilities during their normal release process.

clair can be integrated with a number of container image repositories and CI/CD pipelines.

clair can even be deployed on Kubernetes: `https://github.com/coreos/clair/blob/master/Documentation/running-clair.md#kubernetes-helm`.

clair is also used as the scanning mechanism in CoreOS's Quay image repository. Quay offers a number of enterprise features, including continuous vulnerability scanning (`https://quay.io/`).

Both Docker Hub and Docker Cloud support security scanning. Again, containers that are pushed to the repository are automatically scanned against CVEs, and notifications of vulnerabilities are sent as a result of any findings. Additionally, binary analysis of the code is performed to match the signature of the components with that of known versions.

There are a variety of other scanning tools that can be used as well for scanning your image repositories, including OpenSCAP, Twistlock, Aqua Sec, and many more.

Image signing and verification

Whether you are using a private image repository in-house or a public repository such as Docker Hub, it's important to know that you are only running the code that your developers have written. The potential for malicious code or man-in-the-middle attacks on downloads is an important factor in protecting your container images.

As such, both rkt and Docker support the ability to sign images and verify that the contents have not changed. Publishers can use keys to sign the images when they are pushed to the repositories, and users can verify the signature on the client side when downloading for use.

This is from the rkt documentation:

"Before executing a remotely fetched ACI, rkt will verify it based on attached signatures generated by the ACI creator."

For more information, visit the following links:

- `https://github.com/rkt/rkt/blob/master/Documentation/subcommands/trust.md`
- `https://github.com/rkt/rkt/blob/master/Documentation/signing-and-verification-guide.md`

This is from the Docker documentation:
"Content trust gives you the ability to verify both the integrity and the publisher of all the data received from a registry over any channel. "

For more information, visit `https://docs.docker.com/engine/security/trust/content_trust/`.
This is from the Docker Notary GitHub page:

"The Notary project comprises a server and a client for running and interacting with trusted collections."

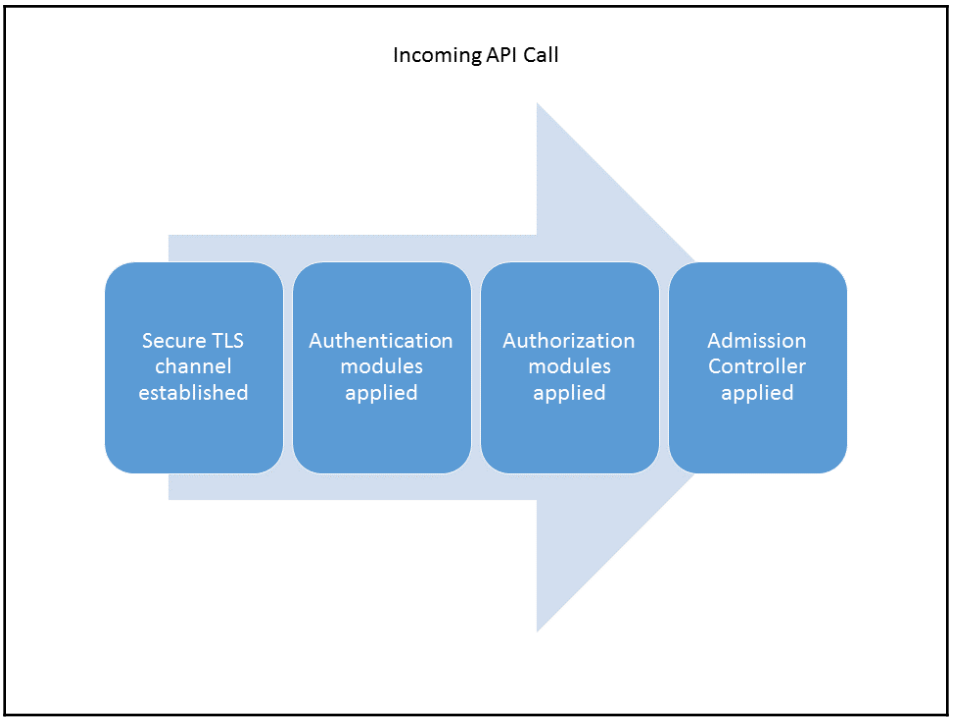
For more information, visit `https://github.com/docker/notary`.

Kubernetes cluster security

Kubernetes has continued to add a number of security features in their latest releases and has a well-rounded set of control points that can be used in your cluster – everything from secure node communication to pod security and even the storage of sensitive configuration data.

Secure API calls

During every API call, Kubernetes applies a number of security controls. This security life cycle is depicted here:



API call life cycle

After secure TLS communication is established, the API server runs through authorization and authentication. Finally, an admission controller loop is applied to the request before it reaches the API server.

Secure node communication

Kubernetes supports the use of secure communication channels between the API server and any client, including the nodes themselves. Whether it's a GUI or command-line utility such as `kubectl`, we can use certificates to communicate with the API server. Hence, the API server is the central interaction point for any changes to the cluster and is a critical component to secure.

In deployments such as GCE, the `kubelet` on each node is deployed for secure communication by default. This setup uses TLS bootstrapping and the new certificates' API to establish a secure connection with the API server using TLS client certificates and a **Certificate Authority** (**CA**) cluster.

Authorization and authentication plugins

The plugin mechanisms for authentication and authorization in Kubernetes are still being developed. They have come a long way, but still have plugins in beta stages and enhancements in the works. There are also third-party providers that integrate with the features here, so bear that in mind when building your hardening strategy.

Authentication is currently supported in the form of tokens, passwords, and certificates, with plans to add the plugin capability at a later stage. OpenID Connect tokens are supported and several third-party implementations, such as Dex from CoreOS and user account and authentication from Cloud Foundry, are available.

Authorization already supports three modes. The **role-based access control** (**RBAC**) mode recently went to general availability in the 1.8 release and brings the standard role-based authentication model to Kubernetes. **Attribute-based access control** (**ABAC**) has long been supported and lets a user define privileges via attributes in a file.

Additionally, a Webhook mechanism is supported, which allows for integration with third-party authorization via REST web service calls. Finally, we have the new node authorization method, which grants permissions to kubelets based on the pods they are scheduled to run.

You can learn more about each area at the following links:

- `http://kubernetes.io/docs/admin/authorization/`
- `http://kubernetes.io/docs/admin/authentication/`
- `https://kubernetes.io/docs/reference/access-authn-authz/node/`

Admission controllers

Kubernetes also provides a mechanism for integrating, with additional verification as a final step. This could be in the form of image scanning, signature checks, or anything that is able to respond in the specified fashion.

When an API call is made, the hook is called and that server can run its verification. Admission controllers can also be used to transform requests and add or alter the original request. Once the operations are run, a response is then sent back with a status that instructs Kubernetes to allow or deny the call.

This can be especially helpful for verifying or testing images, as we mentioned in the last section. The `ImagePolicyWebhook` plugin provides an admission controller that allows for integration with additional image inspection.

For more information, visit the **Using Admission Controller** page in the following documentation: `https://kubernetes.io/docs/admin/admission-controllers/`.

RBAC

As mentioned earlier in this chapter, Kubernetes has now made RBAC a central component to authorization within the cluster. Kubernetes offers two levels for this kind of control. First, there is a *ClusterRole*, which provides cluster-wide authorization to resources. This is handy for enforcing access control across multiple teams, products, or to cluster-wide resources such as the underlying cluster nodes. Second, we have a *Role*, which simply provides access to resources within a specific namespace.

Once you have a role, you need a way to provide users with membership to that role. These are referred to as *Bindings*, and again we have *ClusterRoleBinding* and *RoleBinding*. As with the roles themselves, the former is meant for cluster-wide access and the latter is meant to apply within a specific namespace.

We will not dive into the details of RBAC in this book, but it is something you'll want to explore as you get ready for production grade deployments. The *PodSecurityPolicy* discussed in the next section typically utilizes Roles and RoleBindings to control which policies each user has access to.

For more information, please refer to the documentation here: `https://kubernetes.io/docs/reference/access-authn-authz/rbac/`.

Pod security policies and context

One of the latest additions to the Kubernetes' security arsenal is that of pod security policies and contexts. These allow users to control users and groups for container processes and attached volumes, limit the use of host networks or namespaces, and even set the root filesystem to read-only. Additionally, we can limit the capabilities available and also set SELinux options for the labels that are applied to the containers in each pod.

In addition to SELinux, Kubernetes also added beta support for using AppArmor with your pods by using annotations. For more information, refer to the following documentation page: `https://kubernetes.io/docs/admin/apparmor/`.

PodSecurityPolicies are enforced using the admission controller we spoke of earlier in this book. By default, Kubernetes doesn't enable PodSecurityPolicy, so if you have a GKE cluster running, you can try the following:

```
$ kubectl get psp
```

You should see `'No resources found.'`, assuming you haven't enabled them.

Let's try an example by using the Docker image from our previous chapters. If we use the following `run` command on a cluster with no PodSecurityPolicy applied, it will happily run:

```
$ kubectl run myroottest --image=jonbaier/node-express-info:latest
```

Follow this with `kubectl get pods` and in a minute or so we should see a pod starting with `myroottest` in the listings.

Go ahead and clean this up with the following code before proceeding:

```
$ kubectl delete deployment myroottest
```

Enabling PodSecurityPolicies

Now, let's try this with a cluster that can utilize PodSecurityPolicies. If you are using GKE, it is quite easy to create a cluster with PodSecurityPolicy enabled. Note you will need the Beta APIs enabled for this:

```
$ gcloud beta container clusters create [Cluster Name] --enable-pod-
security-policy --zone=[Zone To Deply Cluster]
```

If you have an existing GKE cluster, you can enable it with a command similar to the preceding one. Simply replace the `create` keyword with `update`.

For clusters created with `kube-up`, like we saw in `Chapter 1`, *Introduction to Kubernetes*, you'll need to enable the admission controller on the API server. Take a look here for more information: `https://kubernetes.io/docs/concepts/policy/pod-security-policy/#enabling-pod-security-policies`.

Once you have PodSecurityPolicy enabled, you can see the applied policies by using the following code:

```
$ kubectl get psp
```

```
NAME                              DATA      CAPS      SELINUX    RUNASUSER   FSGROUP
gce.event-exporter                false               RunAsAny   RunAsAny    RunAsAny
gce.fluentd-gcp                   false               RunAsAny   RunAsAny    RunAsAny
gce.persistent-volume-binder      false               RunAsAny   RunAsAny    RunAsAny
gce.privileged                    true      *         RunAsAny   RunAsAny    RunAsAny
gce.unprivileged-addon            false               RunAsAny   RunAsAny    RunAsAny
```

GKE default pod security policies

You'll notice a few predefined policies that GKE has already defined. You can explore the details and the YAML used to create these policies with the following code:

```
$ kubectl get psp/[PSP Name] -o yaml
```

It's important to note that PodSecurityPolicies work with the RBAC features of Kubernetes. There are a few default roles, role bindings, and namespaces that are defined by GKE. As such, we will see different behaviors based on how we interact with Kubernetes. For example, by using `kubectl` in a GCloud Shell, you may be sending commands as a cluster admin and therefore have access to all policies, including `gce.privileged`. However, using the `kubectl run` command, as we did previously, will invoke the pods through the kube-controller-manager, which will be restricted to the policies bound to its role. Thus, if you simply create a pod with `kubectl`, it will create it without an issue, but by using the `run` command, we will be restricted.

Sticking to our previous method of using `kubectl run`, let's try the same deployment as the preceding one:

```
$ kubectl run myroottest --image=jonbaier/node-express-info:latest
```

Now, if we follow this with `kubectl get pods`, we won't see any pods prefaced with `myroottest`. We can dig a bit deeper by describing our deployment:

```
$ kubectl describe deployment myroottest
```

By using the name of the replica set listed in the output from the preceding command, we can then get the details on the failure. Run the following command:

```
$ kubectl describe rs [ReplicaSet name from deployment describe]
```

Under the events at the bottom, you will see the following pod security policy validation error:

```
Name:           myroottest-588dfdcf85
Namespace:      default
Selector:       pod-template-hash=1448987941,run=myroottest
Labels:         pod-template-hash=1448987941
                run=myroottest
Annotations:    deployment.kubernetes.io/desired-replicas=1
                deployment.kubernetes.io/max-replicas=2
                deployment.kubernetes.io/revision=1
Controlled By:  Deployment/myroottest
Replicas:       0 current / 1 desired
Pods Status:    0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:  pod-template-hash=1448987941
           run=myroottest
  Containers:
   myroottest:
    Image:        jonbaier/node-express-info:latest
    Port:         <none>
    Host Port:    <none>
    Environment:  <none>
    Mounts:       <none>
  Volumes:        <none>
Conditions:
  Type             Status  Reason
  ----             ------  ------
  ReplicaFailure   True    FailedCreate
Events:
  Type     Reason        Age                From                   Message
  ----     ------        ----               ----                   -------
  Warning  FailedCreate  10s (x14 over 51s)  replicaset-controller  Error creating: pods "myroottest-588dfdcf85-"
                                                                    is forbidden: unable to validate against any
                                                                    pod security policy: []
```

Replica set pod security policy validation error

Again, because the `run` command uses the controller manager and that role has no bindings that allow the use of the existing **PodSecurityPolicies**, we are unable to run any pods.

Understanding that running containers securely is not merely the task of administrators adding constraints is important. The work must be done in collaboration with developers, who will properly create the images.

You can find all of the possible parameters for PodSecurityPolicies in the source code, but I've created the following table for convenience. You can find more handy lookups like this on my new site, `http://www.kubesheets.com`:

Parameter	Type	Description	Required
`Privileged`	`bool`	Allows or disallows running a pod as privileged.	No
`DefaultAddCapabilities`	`[]v1.Capaility`	This defines a default set of capabilities that are added to the container. If the pod specifies a capability drop that will override, then add it here. Values are strings of POSIX capabilities minus the leading `CAP_`. For example, `CAP_SETUID` would be `SETUID` (`http://man7.org/linux/man-pages/man7/capabilities.7.html`).	No
`RequiredDropCapabilities`	`[]v1.Capaility`	This defines a set of capabilities that must be dropped from a container. The pod cannot specify any of these capabilities. Values are strings of POSIX capabilities minus the leading `CAP_`. For example, `CAP_SETUID` would be `SETUID` (`http://man7.org/linux/man-pages/man7/capabilities.7.html`).	No
`AllowedCapabilities`	`[]v1.Capaility`	This defines a set of capabilities that are allowed and can be added to a container. The pod can specify any of these capabilities. Values are strings of POSIX capabilities minus the leading `CAP_`. For example, `CAP_SETUID` would be `SETUID` (`http://man7.org/linux/man-pages/man7/capabilities.7.html`).	No
`Volumes`	`[]string`	This list defines which volumes can be used. Leave this empty for all types (`https://github.com/kubernetes/kubernetes/blob/release-1.5/pkg/apis/extensions/v1beta1/types.go#L1127`).	No
`HostNetwork`	`bool`	This allows or disallows the pod to use the host network.	No

`HostPorts`	`[]HostPortRange`	This lets us restrict allowable host ports that can be exposed.	No
`HostPID`	`bool`	This allows or disallows the pod to use the host PID.	No
`HostIPC`	`bool`	This allows or disallows the pod to use the host IPC.	No
`SELinux`	`SELinuxStrategyOptions`	Set it to one of the strategy options, as defined here: `https://kubernetes.io/docs/concepts/policy/pod-security-policy/#selinux`.	Yes
`RunAsUser`	`RunAsUserStrategyOptions`	Set it to one of the strategy options, as defined here: `https://kubernetes.io/docs/concepts/policy/pod-security-policy/#users-and-groups`.	Yes
`SupplementalGroups`	`SupplementalGroupsStrategyOptions`	Set it to one of the strategy options, as defined here: `https://kubernetes.io/docs/concepts/policy/pod-security-policy/#users-and-groups`	Yes
`FSGroup`	`FSGroupStrategyOptions`	Set it to one of the strategy options, as defined here: `https://kubernetes.io/docs/user-guide/pod-security-policy/#strategies`	Yes
`ReadOnlyRootFilesystem`	`bool`	Setting this to `true` will either deny the pod or force it to run with a read-only root filesystem.	No
`allowedHostPaths`	`[]AllowedHostPath`	This provides a whitelist of host paths that can be used at volumes.	No
`allowedFlexVolumes`	`[]AllowedFlexVolume`	This provides a whitelist of flex volumes that can be mounted.	No
`allowPrivilegeEscalation`	`bool`	This governs where `setuid` can be used to change the user a process is running under. Its default is `true`.	No
`defaultAllowPrivilegeEscalation`	`bool`	Sets the default for `allowPrivilegeEscalation`.	No

Additional considerations

In addition to the features we just reviewed, Kubernetes has a number of other constructs that should be considered in your overall cluster hardening process. Earlier in this book, we looked at namespaces that provide a logical separation for multi-tenancy. While the namespaces themselves do not isolate the actual network traffic, some of the network plugins, such as Calico and Canal, provide additional capability for network policies. We also looked at quotas and limits that can be set for each namespace, which should be used to prevent a single tenant or project from consuming too many resources within the cluster.

Securing sensitive application data (secrets)

Sometimes, our application needs to hold sensitive information. This can be credentials or tokens to log in to a database or service. Storing this sensitive information in the image itself is something to be avoided. Here, Kubernetes provides us with a solution in the construct of secrets.

Secrets give us a way to store sensitive information without including plaintext versions in our resource definition files. Secrets can be mounted to the pods that need them and then accessed within the pod as files with the secret values as content. Alternatively, you can also expose the secrets via environment variables.

Given that Kubernetes still relies on plaintext etcd storage, you may want to explore integration with more mature secrets vaults, such as Vault from Hashicorp. There is even a GitHub project for integration: `https://github.com/Boostport/kubernetes-vault`.

We can easily create a secret either with YAML or on the command line. Secrets do need to be base-64 encoded, but if we use the `kubectl` command line, this encoding is done for us.

Let's start with the following secret:

```
$ kubectl create secret generic secret-phrases --from-literal=quiet-
phrase="Shh! Dont' tell"
```

We can then check for the secret with this command:

```
$ kubectl get secrets
```

Now that we have successfully created the secret, let's make a pod that can use the secret. Secrets are consumed in pods by way of attached volumes. In the following `secret-pod.yaml` file, you'll notice that we use `volumeMount` to mount the secret to a folder in our container:

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  containers:
  - name: secret-pod
    image: jonbaier/node-express-info:latest
    ports:
    - containerPort: 80
      name: web
    volumeMounts:
      - name: secret-volume
        mountPath: /etc/secret-phrases
  volumes:
  - name: secret-volume
    secret:
      secretName: secret-phrases
```

Create this pod with `kubectl create -f secret-pod.yaml`. Once created, we can get a bash shell in the pod with `kubectl exec` and then change directories to the `/etc/secret-phrases` folder that we set up in the pod definition. Listing this directory reveals a single file with the name of the secret that we created earlier:

```
$ kubectl exec -it secret-pod bash
$ cd /etc/secret-phrases
$ ls
```

If we then display its contents, we should see the phrase we encoded previously, `Shh! Dont' tell`:

```
$ cat quiet-phrase
```

Typically, this would be used for a username and password to a database or service, or any sensitive credentials and configuration data.

Bear in mind that secrets are still in their early stages, but they are a vital component for production operations. There are several improvements being planned for future releases. At the moment, secrets are still stored in plaintext in the etcd server. However, the secrets construct does allow us to control which pods can access it, and it stores the information on the tmpfs, but does not store it at rest for each pod. You can limit users with access to etcd and perform additional wipe procedures when you decommission servers, but you'll likely want more protection in place for a production-ready system.

Summary

In this chapter, we took a look at basic container security and some essential areas of consideration. We also touched on basic image security and continuous vulnerability scanning. Later in this chapter, we looked at the overall security features of Kubernetes, including secrets for storing sensitive configuration data, secure API calls, and even setting up security policies and contexts for pods running on our cluster.

You should now have a solid starting point for securing your cluster and moving toward production. To that end, the next chapter will cover an overall strategy for moving toward production and will also look at some third-party vendors that offer tools to fill in the gaps and assist you on the way.

Questions

- Which component can be used as a central point for managing and prevent vulnerabilities from being released?
- What are three methods for authorization within a Kubernetes cluster?
- Which parameter of a PodSecurityPolicy disallows the running of privileged containers?
- How do you list all secrets that you have access to in a cluster?

Further reading

- https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-10pdf/
- https://github.com/moby/moby/blob/89dac8427e7366cbd6a47e713fe8f445198ca3d4/oci/defaults.go#L14
- https://github.com/kubernetes/kubernetes/blob/2d7b92ee743de20d17406003e463a829a0db5a51/pkg/apis/policy/types.go#L145

14
Hardening Kubernetes

In this chapter, we'll look at considerations for moving to production. We will also show you some helpful tools and third-party projects that are available in the Kubernetes community at large and where you can go to get more help.

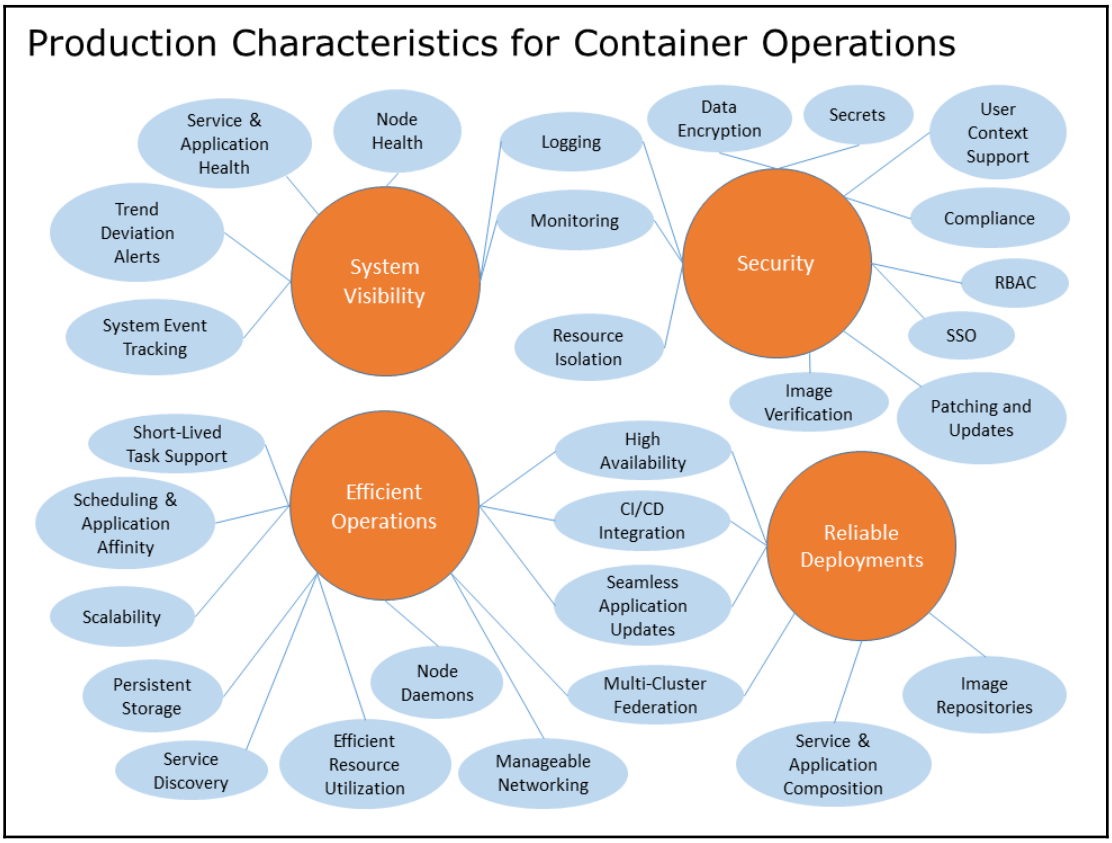
This chapter will discuss the following topics:

- Production characteristics
- Lessons learned from Kubernetes production
- Hardening the cluster
- The Kubernetes ecosystem
- Where can you get help?

Ready for production

So far in this book, we have walked through a number of typical operations using Kubernetes. As we have been, K8s offers a variety of features and abstractions that ease the burden of day-to-day management for container deployments.

There are many characteristics that define a production-ready system for containers. The following diagram provides a high-level view of the major concerns for production-ready clusters. This is by no means an exhaustive list, but it's meant to provide some solid ground for heading into production operations:



Production characteristics for container operations

We saw how the core concepts and abstractions of Kubernetes address a few of these concerns. The service abstraction has built-in service discovery and health checking at both the service and application level. We also get seamless application updates and scalability from the replication controller and deployment constructs. All of the core abstractions of services, replication controllers, replica sets, and pods work with a core scheduling and affinity rulesets and give us easy service and application composition.

There is built-in support for a variety of persistent storage options, and the networking model provides manageable network operations with options to work with other third-party providers. We also took a brief look at CI/CD integration with some of the popular tools in the marketplace.

Furthermore, we have built-in system events tracking, and with the major cloud providers, an out-of-the-box setup for monitoring and logging. We also saw how this can be extended to third-party providers such as Stackdriver and Sysdig. These services also address overall node health and proactive trend deviation alerts.

The core constructs also help us address high availability in our application and service layers. The scheduler can be used with autoscaling mechanisms to provide this at a node level. Then, there is support for making the Kubernetes master itself highly available. In `Chapter 12`, *Cluster Federation and Multi-Tenancy*, we took a brief look at the new federation capabilities that promise a multi-cloud and multi-data center model for the future.

Finally, we explored a new breed of operating systems that give us a slim base to build on and secure update mechanisms for patching and updates. The slim base, together with scheduling, can help us with efficient resource utilization. In addition, we looked at some hardened concerns and explored the image trust and verification tools available. Security is a wide topic and capability matrices exist for this topic alone.

Ready, set, go

While there are still some gaps, a variety of the remaining security and operation concerns are actively being addressed by third-party companies, as we will see in the following section. Going forward, the Kubernetes project will continue to evolve, and the community of projects and partners around K8s and Docker will also grow. The community is closing the remaining gaps at a phenomenal pace.

Lessons learned from production

Kubernetes has been around long enough now that there are a number of companies running Kubernetes. In our day jobs, we've seen Kubernetes run in production across a number of different industry verticals and in numerous configurations. Let's explore what folks across the industry are doing when providing customer-facing workloads. At a high level, there are several key areas:

- Make sure to set limits in your cluster.
- Use the appropriate workload types for your application.

- Label everything! Labels are very flexible and can contain a lot of information that can help identify an object, route traffic, or determine placement.
- Don't use default values.
- Tweak the default values for the core Kubernetes components.
- Use load balancers as opposed to exposing services directly on a node's port.
- Build your Infrastructure as Code and use provisioning tools such as CloudFormation or Terraform, and configuration tools such as Chef, Ansible, or Puppet.
- Consider not running stateful workloads in production clusters until you build up expertise in Kubernetes.
- Investigate higher-function templating languages to maintain the state of your cluster. We'll explore a few options for an immutable infrastructure in the following chapter.
- Use RBAC, the principle of least privilege, and separation of concerns wherever possible.
- Use TLS-enabled communications for all inter-cluster chatter. You can set up TLS and certificate rotation for the `kubelet` communication in your cluster.
- Until you're comfortable with managing Kubernetes, build lots of small clusters. It's more operational overhead, but it will get you into the deep end of experience faster so that you see more failure and experience the operator burden more heavily.
- As you get better at Kubernetes, build bigger clusters that use namespaces, network segmentation, and the authorization features to break up your cluster into pieces.
- Once you're running a few large clusters, manage them with `kubefed`.
- If you can, use the features of your given cloud service provider's built-in high availability on a Kubernetes platform. For example, run Regional Clusters on GCP, with GKE. This feature spreads your nodes across several availability zones in a region. This allows for resilience against a single zone failure, and provides the conceptual building blocks for the zero downtime upgrades of your master nodes.

In the next section, we'll explore one of these concepts – limits – in more detail.

Setting limits

If you've done work with containers before, you will know that one of the first and easiest things to set up for your containers is resource limits in the form of the following metrics:

- CPU
- Memory
- Requests

You may be familiar with setting runtime limits on resources with Docker's CLI, which specify flags to limit these items and more:

```
docker run --it --cpu-rt-runtime=950000 \
 --ulimit rtprio=99 \
 --memory=1024m \
 --cpus=".5"
 alpine /bin/sh
```

Here, we're setting a runtime parameter, creating a `ulimit`, and setting memory and CPU quotas. The story evolves a bit in Kubernetes, as you can create these limits to a specific namespace, which allows you to characterize your limits by the domains of your cluster. You have four overarching parameters so that you can work resource limits in Kubernetes:

```
spec.containers[].resources.limits.cpu
spec.containers[].resources.requests.cpu
spec.containers[].resources.limits.memory
spec.containers[].resources.requests.memory
```

Scheduling limits

When you create a pod with a memory limit, Kubernetes looks for a node with the right labels and selectors that has enough of the resource types, CPU, and memory, that the pod requires. Kubernetes is in charge of ensuring that the total memory request of the pods on a node is not less than the pod's total resources. This can sometimes result in unexpected outcomes, as you can have node limitations reached in terms of capacity, even if the net utilization of a pod is low. This is a design of the system in order to accommodate varying load levels.

You can look through pod logs to find out when this has occurred:

```
$ kubectl describe pod web| grep -C 3 Events
Events:
FirstSeen LastSeen Count From Subobject PathReason Message
74s 15s 2 {scheduler } FailedScheduling Failed for reason PodExceedsFreeCPU
and possibly others
```

You can address these issues by removing unneeded pods, ensuring that your pod isn't larger as a whole than any one available node, or simply add more resources to the cluster.

Memory limit example

Let's walk through an example. First, we'll create a namespace to house our memory limit:

```
master $ kubectl create namespace low-memory-area
namespace "low-memory-area" created
```

Once we've created the namespace, we can create a file that sets a `LimitRange` object, which will allow us to enforce a default for memory limits and requests. Create a file called `memory-default.yaml` with the following contents:

```
apiVersion: v1
kind: LimitRange
metadata:
 name: mem-limit-range
spec:
 limits:
 - default:
     memory: 512Mi
   defaultRequest:
     memory: 256Mi
   type: Container
```

And now, we can create it in the namespace:

```
master $ kubectl create -f test.ym --namespace=low-memory-area
limitrange "mem-limit-range" created
```

Let's create a pod without a memory limit, in the low-memory-area namespace, and see what happens.

Create the following `low-memory-pod.yaml` file:

```
apiVersion: v1
kind: Pod
metadata:
 name: low-mem-demo
spec:
 containers:
 - name: low-mem-demo
   image: redis
```

Then, we can create the pod with this command:

```
kubectl create -f low-memory-pod.yaml --namespace=low-memory-area
pod "low-mem-demo" created
```

Let's see if our resource constraints were added to the pod's configuration for containers, without having to explicitly specify it in the pod configuration. Notice the memory limits in place! We've removed some of the informational output for readability:

```
kubectl get pod low-mem-demo --output=yaml --namespace=low-memory-area
```

Here's the output of the preceding code:

```
apiVersion: v1
kind: Pod
metadata:
 annotations:
   kubernetes.io/limit-ranger: 'LimitRanger plugin set: memory request for
container
     low-mem-demo; memory limit for container low-mem-demo'
 creationTimestamp: 2018-09-20T01:41:40Z
 name: low-mem-demo
 namespace: low-memory-area
 resourceVersion: "1132"
 selfLink: /api/v1/namespaces/low-memory-area/pods/low-mem-demo
 uid: 52610141-bc76-11e8-a910-0242ac11006a
spec:
 containers:
 - image: redis
   imagePullPolicy: Always
   name: low-mem-demo
   resources:
     limits:
       memory: 512Mi
     requests:
       memory: 256Mi
   terminationMessagePath: /dev/termination-log
```

```
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
      name: default-token-t6xqm
      readOnly: true
  dnsPolicy: ClusterFirst
  nodeName: node01
  restartPolicy: Always
  schedulerName: default-scheduler
  securityContext: {}
  serviceAccount: default
  serviceAccountName: default
  terminationGracePeriodSeconds: 30
  tolerations:
  - effect: NoExecute
    key: node.kubernetes.io/not-ready
    operator: Exists
    tolerationSeconds: 300
  - effect: NoExecute
    key: node.kubernetes.io/unreachable
    operator: Exists
    tolerationSeconds: 300
  volumes:
  - name: default-token-t6xqm
    secret:
      defaultMode: 420
      secretName: default-token-t6xqm
  hostIP: 172.17.1.21
  phase: Running
  podIP: 10.32.0.3
  qosClass: Burstable
  startTime: 2018-09-20T01:41:40Z
```

You can delete the pod with the following command:

```
Kubectl delete pod low-mem-demo --namespace=low-memory-area
pod "low-mem-demo" delete
```

There are a lot of options for configuring resource limits. If you create a memory limit, but don't specify the default request, the request will be set to the maximum available memory, which will correspond to the memory limit. That will look like the following:

```
resources:
 limits:
   memory: 4096m
 requests:
   memory: 4096m
```

In a cluster with diverse workloads and API-driven relationships, it's incredibly important to set memory limits with your containers and their corresponding applications in order to prevent misbehaving applications from disrupting your cluster. Services don't implicitly know about each other, so they're very susceptible to resource exhaustion if you don't configure limits correctly.

Scheduling CPU constraints

Let's look at another type of resource management, the constraint. We'll use the CPU dimension here, and we'll explore how to set the maximum and minimum values for available resources for a given container and pod in a namespace. There are a number of reasons you might want to limit CPU on a Kubernetes cluster:

- If you have a namespaced cluster that has different levels of production and non-production workloads, you may want to specify higher limits for your production workloads. You can allow quad-core CPU consumption for production; put pin development, staging, or UAT-type workloads to a single CPU; or stagger them according to environment needs.
- You can also ban requests from pods that require more CPU resources than your nodes have available. If you're running a certain type of machine on a cloud service provider, you can ensure that workloads that require X cores aren't scheduled on machines with <X cores.

CPU constraints example

Let's go ahead and create another namespace in which to hold our example:

```
kubectl create namespace cpu-low-area
```

Now, let's set up a `LimitRange` for CPU constraints, which uses the measurement of millicpus. If you're requesting 500 m, it means that you're asking for 500 millicpus or millicores, which is equivalent to 0.5 in notational form. When you request 0.5 or 500 m, you're asking for half of a CPU in whatever form your platform provides (vCPU, Core, Hyper Thread, vCore, or vCPU).

As we did previously, let's create a `LimitRange` for our CPU constraints:

```
apiVersion: v1
kind: LimitRange
metadata:
 name: cpu-demo-range
spec:
 limits:
 - max:
     cpu: "500m"
   min:
     cpu: "300m"
   type: Container
```

Now, we can create the `LimitRange`:

```
kubectl create -f cpu-constraint.yaml --namespace=cpu-low-area
```

Once we create the `LimitRange`, we can inspect it. What you'll notice is that the `defaultRequest` is specified as the same as the maximum, because we didn't specify it. Kubernetes sets the `defaultRequest` to max:

```
kubectl get limitrange cpu-demo-range --output=yaml --namespace=cpu-low-
area

limits:
- default:
    cpu: 500m
 defaultRequest:
    cpu: 500m
 max:
    cpu: 500m
 min:
    cpu: 300m
 type: Container
```

This is the intended behavior. When further containers are scheduled in this namespace, Kubernetes first checks to see whether the pod specifies a request and limit. If it doesn't, the defaults are applied. Next, the controller confirms that the CPU request is more than the lower bound in the `LimitRange`, 300 m. Additionally, it checks for the upper bound to make sure that the object is not asking for more than 500 m.

You can check the container constraints again by looking at the YAML output of the pod:

```
kubectl get pod cpu-demo-range --output=yaml --namespace=cpu-low-area

resources:
 limits:
   cpu: 500m
 requests:
   cpu: 300m
```

Now, don't forget to delete the pod:

```
kubectl delete pod cpu-demo-range --namespace=cpu-low-area
```

Securing a cluster

Let's look at some other common recommendations for hardening your cluster in production. These use cases cover both intentional, malicious actions against your cluster, as well as accidental misuse. Let's take a look at what we can do to secure things.

First off, you want to ensure that access to the Kubernetes API is controlled. Given that all actions in Kubernetes are API-driven, we should secure this interface first. We can control access to this API with several settings:

- **Encode all traffic**: In order to keep communication secure, you should make sure that **Transport Level Security** (**TLS**) is set up for API communication in the cluster. Most of the installation methods we've reviewed in this book create the necessary component certificates, but it's always on the cluster operators to identify all in-use local ports that may not use the more secure settings.
- **Authenticate your access**: Just as with any large scale computer system, you want to ensure that the identity of a user is established. For small clusters, you can use certs or tokens, while larger production clusters should use OpenID or LDAP.
- **Control your access**: After you've established the identity of the role accessing your API, you always want to ensure that you pass your authenticated access request through an authorization filter with Kubernetes' built-in **role-based-access-control** (**RBAC**), which helps operators limit control and access by roles and users. There are two authorizer plugins, node and RBAC, that can be used, along with the `NodeRestriction` admission plugin. A key point to keep in mind is that role granularity should increase as cluster size increases, and even more so from non-production environments toward production environments.

By default, authentication to use the `kubelet` is turned off. You can enable authorization/authentication on the `kubelet` by turning on certificate rotation.

You can read more about certificate rotation here: `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet-authentication-authorization/`.

We can also modify the usage of Kubernetes at runtime:

- Long-time operators of Kubernetes in production will recognize this as a reference point to our previous discussions on limits and policies. As discussed previously, resource quotas limit the number of resources provided within a namespace, while limits ranges between restrict minimum and maximum sizes.
- You can determine the privileges of your pods by defining a security context. Here, you can specify things like a particular Linux user, group, volume mount access, allowing privilege escalation, and more.
- You can also restrict access to logical partitions of your cluster by using network policies. You can ensure that certain namespaces are off limits to users, or determine whether or not they're able to set up services with specific load balancer configuration or open ports on host nodes.

While the preceding patterns are useful for operations inside of Kubernetes, there are also several actions that you should take when securing your cluster from an external perspective:

- First off, enable and monitor your logs! While this seems like a no-brainer, we see a lot of problems cropping up from people that aren't watching their logs, or who haven't created alerts based off these logs. Another hint: don't store logs inside of your cluster! If your cluster is breached, then those logs will be an invaluable source of information for malicious agents.
- Make double sure that you restrict access to your etcd cluster. This can come in the form of setting up security groups or firewalls and ensuring that your etcd cluster nodes have the appropriate identity and access management from an infrastructure perspective. From a cluster perspective, make sure that you're always using TLS certificates for authentication and strong credentials. In no case should any components inside your cluster have read/write access to the full etcd key/value space.

- Make sure to vet alpha/beta components and review third-party integrations. Make sure that you know what you're using when you enable it, and what it does when you turn it on! Emerging features or third-party tools may create attack surfaces or threat models where you're not aware of what dependencies they have. Beware of any tools that need to do work inside the kube-system, as it's a particularly powerful portion of the cluster.
- Encrypt your secrets at rest in etcd. This is good advice for any computerized system, and Kubernetes is no different here. The same goes for your backups to ensure that an attacker can't gain access to your cluster via inspection of those resources.

For your production cluster, you should also be doing things such as scanning your container images, running static analysis of your YAML files, running containers as non-root users where possible, and running an **intrusion detection system** (**IDS**). Once you have all of this in place, you can begin to explore the functional capabilities of the service meshes out in the wild.

Third-party companies

Since the Kubernetes project's initial release, there has been a growing ecosystem of partners. We looked at CoreOS, Sysdig, and many others in the previous chapters, but there are a variety of projects and companies in this space. We will highlight a few that may be useful as you move toward production. This is by no means an exhaustive list and it is merely meant to provide some interesting starting points.

Private registries

In many situations, organizations will not want to place their applications and/or intellectual property in public repositories. For those cases, a private registry solution is helpful in securely integrating deployments end to end.

Google Cloud offers the Google Container Registry at `https://cloud.google.com/container-registry/`.

Docker has its own trusted registry offering at `https://www.docker.com/docker-trusted-registry`.

Quay also provides secure private registries, vulnerability scanning, and comes from the CoreOS team, and can be found at `https://quay.io/`.

Google Kubernetes Engine

Google was the main author of the original Kubernetes project and is still a major contributor. Although this book has mostly focused on running Kubernetes on our own, Google also offers a fully managed container service through the Google Cloud Platform.

Find more information on the **Google Kubernetes Engine** (**GKE**) website at `https://cloud.google.com/container-engine/`.

Kubernetes will be installed on GKE and will be managed by Google engineers. They also provide private registries and integration with your existing private networks.

You create your first GKE cluster by using the following steps:

1. From the GCP console, in **Compute**, click on **Container Engine**, and then on **Container Clusters**.
2. If this is your first time creating a cluster, you'll have an information box in the middle of the page. Click on the **Create a container cluster** button.
3. Choose a name for your cluster and the zone. You'll also be able to choose the machine type (instance size) for your nodes and how many nodes (cluster size) you want in your cluster. You'll also see a choice for node image, which lets you choose the base OS and machine image for the nodes themselves. The master is managed and updated by the Google team themselves.
4. Leave Stackdriver logging and Stackdriver monitoring checked. Click on **Create**, and in a few minutes, you'll have a new cluster ready for use.
5. You'll need `kubectl`, which is included with the Google SDK, to begin using your GKE cluster. Refer to `Chapter 1`, *Introduction to Kubernetes*, for details on installing the SDK. Once we have the SDK, we can configure `kubectl` and the SDK for our cluster using the steps outlined at `https://cloud.google.com/container-engine/docs/before-you-begin#install_kubectl`.

Azure Kubernetes Service

Another cloud-managed offering is Microsoft's **Azure Kubernetes Service** (**AKS**). AKS is really nice because it allows you to choose from industry standard tools such as Docker Swarm, Kubernetes, and Mesos. It then creates a managed cluster for you, but uses one of these toolsets as the foundation. The advantage is that you can still use the tool's native API and management tools, but leave the management of the cloud infrastructure to Azure.

You can find out more about ACS at `https://azure.microsoft.com/en-us/services/container-service/`.

ClusterHQ

ClusterHQ provides a solution for bringing stateful data into your containerized applications. They provide Flocker, a tool for managing persistent storage volumes with containers, and FlockerHub, which provides a storage repository for your data volumes.

Portworx

Portworx is another player in the storage space. It provides solutions for bringing persistence storage to your containers. Additionally, it has features for snapshotting, encryption, and even multi-cloud replication.

Please refer to the Portworx website for more information: `https://portworx.com/`.

Shippable

Shippable is a continuous integration, continuous deployment, and release automation platform that has built-in support for a variety of modern container environments. The product touts support for any language with a uniform support for packaging and test.

Please refer to the Shippable website for more information: `https://app.shippable.com/`.

Twistlock

Twistlock.io is a vulnerability and hardening tool that's tailor-made for containers. It provides the ability to enforce policies, hardens according to CIS standards, and scans images in any popular registry for vulnerabilities. It also provides scan integration with popular CI/CD tools and RBAC solutions for many orchestration tools such as Kubernetes.

Please refer to the Twistlock website for more information: `https://www.twistlock.com/`.

Aqua Sec

Aqua Sec is another security tool that provides a variety of features. Image scanning with popular registries, policy enforcement, user access control, and container hardening are all covered. Additionally, Aqua Sec has some interesting functionality in network segmentation.

Please refer to the Aqua's website for more information: `https://www.aquasec.com/`.

Mesosphere (Kubernetes on Mesos)

Mesosphere itself is building a commercially supported product around the open source Apache Mesos project. Apache Mesos is a cluster management system that offers scheduling and resource sharing, a bit like Kubernetes itself, but at a much higher level. The open source project is used by several well-known companies, such as Twitter and Airbnb.

You can find out more information about the Mesos OS project and the Mesosphere offerings at the following sites:

- `http://mesos.apache.org/`
- `https://mesosphere.com/`

Mesos, by its nature, is modular, and allows the use of different frameworks for a variety of platforms. A Kubernetes framework is now available, so we can take advantage of the cluster management in Mesos while still maintaining the useful application-level abstractions in K8s. Refer to the following link for more information: `https://github.com/kubernetes-incubator/kube-mesos-framework`.

Deis

The Deis project provides an open source **Platform as a Service** (**PaaS**) solution based on and around Kubernetes. This allows companies to deploy their own PaaS on-premise or on the public cloud. Deis provides tools for application composition and deployment, package management (at the pod level), and service brokering.

OpenShift

Another PaaS solution is OpenShift from Red Hat. The OpenShift platform uses the Red Hat Atomic platform as a secure and slim OS for running containers. In version 3, Kubernetes was added as the orchestration layer for all container operations on your PaaS. This is a great combination for managing PaaS installations at a large scale.

More information on OpenShift can be found at `https://enterprise.openshift.com/`.

Summary

In this chapter, we left a few breadcrumbs to guide you on your continuing journey with Kubernetes. You should have a solid set of production characteristics to get you started. There is a wide community in both the Docker and Kubernetes worlds. There are also a few additional resources that we provided if you need a friendly face along the way.

By now, you have seen the full spectrum of container operations with Kubernetes. You should be more confident in how Kubernetes can streamline the management of your container deployments and how you can plan to move containers off developer laptops and onto production servers. Now get out there and start shipping your containers!

Questions

- What are some of the key characteristics of production systems?
- What are two examples of third-party monitoring systems?
- Which tools can help you build Infrastructure as Code?
- What is RBAC?
- What limits can you set in a Kubernetes cluster?
- In which dimensions can constraints be set?
- Which technology should be used to secure communication within a cluster?

Further reading

The Kubernetes project is an open source effort, so there is a broad community of contributors and enthusiasts. One great resource in order to find more assistance is the Kubernetes Slack channel: `http://slack.kubernetes.io/`.

There is also a Kubernetes group on Google groups. You can join it at `https://groups.google.com/forum/#!forum/kubernetes-users`.

If you enjoyed this book, you can find more of my articles, how-tos, and various musings on my blogs and Twitter page:

- `https://medium.com/@grizzbaier`
- `https://twitter.com/grizzbaier`

15
Kubernetes Infrastructure Management

In this chapter, we'll discuss how to make changes to the infrastructure that powers your Kubernetes infrastructure, whether or not it is a purely public cloud platform or a hybrid installation. We'll discuss methods for handling underlying instance and resource instability, and strategies for running highly available workloads on partially available underlying hardware. We'll cover a few key topics in this chapter in order to build your understanding of how to manage infrastructure in this way:

- How do we plan to deploy Kubernetes components?
- How do we secure Kubernetes infrastructure?
- How do we upgrade the cluster and `kubeadm`?
- How do we scale up the cluster?
- What external resources are available to us?

In this chapter, you'll learn about the following:

- Cluster upgrades
- How to manage `kubeadm`
- Cluster scaling
- Cluster maintenance
- The SIG Cluster Lifecycle group

Technical requirements

You'll need to have your Google Cloud Platform account enabled and logged in, or you can use a local Minikube instance of Kubernetes. You can also use Play with Kubernetes over the web: `https://labs.play-with-k8s.com/`.

Here's the GitHub repository for this chapter: `https://github.com/PacktPublishing/Getting-Started-with-Kubernetes-third-edition/tree/master/Code-files/Chapter15`.

Planning a cluster

Looking back over the work we've done up till now in this book, there are a lot of options when it comes to building a cluster with Kubernetes. Let's briefly highlight the options you have available to you when you're planning on building your cluster. We have a few key areas to investigate when planning ahead.

Picking what's right

The first and arguably most important step when choosing a cluster is to pick the right hosted platform for your Kubernetes cluster. At a high level, here are the choices you have:

- Local solutions include the following:
    - **Minikube**: A single-node Kubernetes cluster
    - **Ubuntu on LXD**: This uses LXD to deploy a nine-instance cluster of Kubernetes
    - **IBM's Cloud Private-CE**: This uses VirtualBox to deploy Kubernetes on *n+1* instances
    - `kubeadm-dind` (Docker-in-Docker): This allows for multi-node Kubernetes clusters
- Hosted solutions include the following:
    - Google Kubernetes Engine
    - Amazon Elastic Container Services
    - Azure Kubernetes Service
    - Stackpoint
    - Openshift online
    - IBM Cloud Kubernetes Services
    - Giant Swarm
- On all of the aforementioned clouds and more, there are many turnkey solutions that allow you to spin up full clusters with community-maintained scripts

As of this book's publishing, here's a list of projects and solutions:

IaaS Provider	Config. Mgmt.	OS	Networking
any	any	multi-support	any CNI
Google Kubernetes Engine			GCE
Stackpoint.io		multi-support	multi-support
AppsCode.com	Saltstack	Debian	multi-support
Madcore.Ai	Jenkins DSL	Ubuntu	flannel
Platform9		multi-support	multi-support
Kublr	custom	multi-support	multi-support
Kubermatic		multi-support	multi-support
IBM Cloud Kubernetes Service		Ubuntu	IBM Cloud Networking + Calico
Giant Swarm		CoreOS	flannel and/or Calico
GCE	Saltstack	Debian	GCE
Azure Kubernetes Service		Ubuntu	Azure
Azure (IaaS)		Ubuntu	Azure
Bare-metal	custom	Fedora	*none*
Bare-metal	custom	Fedora	flannel

Check out this link for more turnkey solutions: `https://kubernetes.io/docs/setup/pick-right-solution/#turnkey-cloud-solutions`.

Securing the cluster

As we've discussed, there are several areas of focus when securing a cluster. Ensure that you have read through and made configuration changes (in code) to your cluster configuration in the following areas:

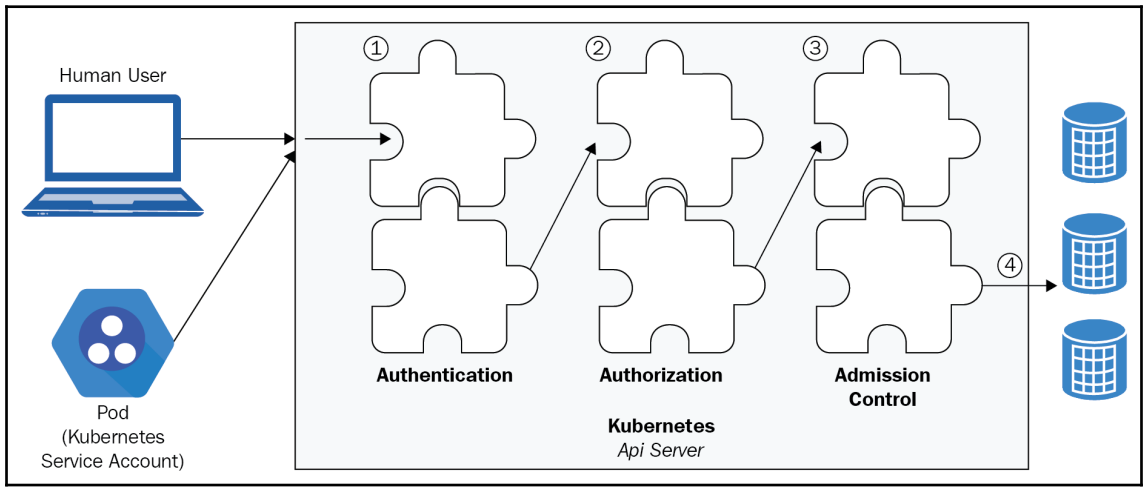
- **Logging**: Ensure that your Kubernetes logs are enabled. You can read more about audit logging here: `https://kubernetes.io/docs/tasks/debug-application-cluster/audit/`.
- Make sure you have authentication enabled so that your users, operators, and services identify themselves as unique identifiers. Read more about authentication here: `https://kubernetes.io/docs/reference/access-authn-authz/authentication/`.
- Ensure that you have proper separation of duties, role-based access control, and fine grained privileges using authorization. You can read more about HTTP-based controls here: `https://kubernetes.io/docs/reference/access-authn-authz/authorization/`.
- Make sure that you have locked down the API to specific permissions and groups. You can read more about access to the API here: `https://kubernetes.io/docs/reference/access-authn-authz/controlling-access/`.
- When appropriate, enable an admission controller to further re-validate requests after they pass through the authentication and authorization controls. These controllers can take additional, business-logic based validation steps in order to secure your cluster further. Read more about admission controllers here: `https://kubernetes.io/docs/reference/access-authn-authz/controlling-access`.
- Tune Linux system parameters via the `sysctl` interface. This allows you to modify kernel parameters for node-level and namespaced `sysctl` features. There are safe and unsafe system parameters. There are several subsystems that can be tweaked with `sysctls`. Possible parameters are as follows:
    - `abi`: Execution domains and personalities
    - `fs`: Specific filesystems, filehandle, inode, dentry, and quota tuning
    - `kernel`: Global kernel information/tuning
    - `net`: Networking
    - `sunrpc`: SUN **Remote Procedure Call** (**RPC**)
    - `vm`: Memory management tuning, buffer, and cache management
    - `user`: Per user per user namespace limits

You can read more about `sysctl` calls here: `https://kubernetes.io/docs/tasks/administer-cluster/sysctl-cluster/`.

You can enable unsafe `sysctl` values by running the following command:

```
kubelet --allowed-unsafe-sysctls 'net.ipv4.route.min_pmtu'
```

Here's a diagram of the authorization, authentication, and admission control working together:



Tuning examples

If you'd like to experiment with modifying `sysctls`, you can set a security context as follows, per pod:

```
apiVersion: v1
kind: Pod
metadata:
 name: sysctl-example
spec:
 securityContext:
   sysctls:
   - name: kernel.shm_rmid_forced
     value: "0"
   - name: net.core.somaxconn
     value: "10000"
```

```
  - name: kernel.msgmax
    value: "65536"
  - name: ipv4.ip_local_port_range
     value: '1024 65535'
```

You can also tune variables such as the ARP cache, as Kubernetes consumes a lot of IPs at scale, which can exhaust space in the ARP cache. Changing these settings is common in large scale HPC clusters and can help with address exhaustion with Kubernetes as well. You can set these values, as follows:

```
net.ipv4.neigh.default.gc_thresh1 = 90000
net.ipv4.neigh.default.gc_thresh2 = 100000
net.ipv4.neigh.default.gc_thresh3 = 120000
```

Upgrading the cluster

In order to run your cluster over long periods of time, you'll need to update your cluster as needed. There are several ways to manage cluster upgrades, and the difficulty level of the upgrades is determined by the platform you've chosen previously. As a general rule, hosted **Platform as a service** (**PaaS**) options are simpler, while roll your own options rely on you to manage your cluster upgrades.

Upgrading PaaS clusters

Upgrading PaaS clusters is a lot simpler than updating your hand-rolled clusters. Let's check out how the major cloud service providers update their hosted Kubernetes platforms.

With Azure, it's relatively straightforward to manage an upgrade of both the control plane and nodes of your cluster. You can check which upgrades are available for your cluster with the following command:

```
az aks get-upgrades --name "myAKSCluster" --resource-group myResourceGroup --
--output table
Name ResourceGroup MasterVersion NodePoolVersion Upgrades

------- --------------- --------------- ----------------- -----------------
--

default gsw-k8s-aks 1.8.10 1.8.10 1.9.1, 1.9.2, 1.9.6
```

When upgrading AKS clusters, you have to upgrade through minor versions. AKS handles adding a new node to your cluster and manages to cordon and drain process in order to prevent any disruption to your running applications. You can see how the drain process works in a following section.

You can run the `upgrade` command as follows. You should experiment with this feature before running on production workloads so you can understand the impact on running applications:

```
az aks upgrade --name myAKSCluster --resource-group myResourceGroup --
kubernetes-version 1.9.6
```

You should see a lot of output that identifies the update, which will look something like this:

```
{
  "id": "/subscriptions/<Subscription
ID>/resourcegroups/myResourceGroup/providers/Microsoft.ContainerService/man
agedClusters/myAKSCluster",
  "location": "eastus",
  "name": "myAKSCluster",
  "properties": {
    "accessProfiles": {
      "clusterAdmin": {
        "kubeConfig": "..."
      },
      "clusterUser": {
        "kubeConfig": "..."
      }
    },
    "agentPoolProfiles": [
      {
        "count": 1,
        "dnsPrefix": null,
        "fqdn": null,
        "name": "myAKSCluster",
        "osDiskSizeGb": null,
        "osType": "Linux",
        "ports": null,
        "storageProfile": "ManagedDisks",
        "vmSize": "Standard_D2_v2",
        "vnetSubnetId": null
      }
    ],
    "dnsPrefix": "myK8sClust-myResourceGroup-4f48ee",
    "fqdn": "myk8sclust-
myresourcegroup-4f48ee-406cc140.hcp.eastus.azmk8s.io",
```

```
      "kubernetesVersion": "1.9.6",
      "linuxProfile": {
        "adminUsername": "azureuser",
        "ssh": {
          "publicKeys": [
            {
              "keyData": "..."
            }
          ]
        }
      },
      "provisioningState": "Succeeded",
      "servicePrincipalProfile": {
        "clientId": "e70c1c1c-0ca4-4e0a-be5e-aea5225af017",
        "keyVaultSecretRef": null,
        "secret": null
      }
    },
    "resourceGroup": "myResourceGroup",
    "tags": null,
    "type": "Microsoft.ContainerService/ManagedClusters"
  }
```

You can additionally show the current version:

```
az aks show --name myAKSCluster --resource-group myResourceGroup --output
table
```

To upgrade a GCE cluster, you'll follow a similar procedure. In GCE's case, there are two mechanisms that allow you update your cluster:

- For manager node upgrades, GCP deletes and recreates the master nodes using the same **Persistent Disk** (**PD**) to preserve your state across upgrades
- With your worker nodes, you'll use GCP's manage instance groups and perform a rolling upgrade of your cluster, wherein each node is destroyed and replaced to avoid interruption to your workloads

You can upgrade your cluster master to a specific version:

```
cluster/gce/upgrade.sh -M v1.0.2
```

Or, you can update your full cluster with this command:

```
cluster/gce/upgrade.sh -M v1.0.2
```

To upgrade a Google Kubernetes Engine cluster, you have a simple, user-initiated option. You'll need to set your project ID:

```
gcloud config set project [PROJECT_ID]
```

And, make sure that you have the latest set of `gcloud` components:

```
gcloud components update
```

When updating Kubernetes clusters on GCP, you get the following benefits. You can downgrade your nodes, but you cannot downgrade your master:

- GKE will handle node and pod drainage without application interruption
- Replacement nodes will be recreated with the same node and configuration as their predecessors
- GKE will update software for the following pieces of the cluster:
    - `kubelet`
    - `kube-proxy`
    - Docker daemon
    - OS

You can see what options your server has for upgrades with this command:

```
gcloud container get-server-config
```

Keep in mind that data stored in the `hostPath` and `emptyDir` directories will be deleted during the upgrade, and only PDs will be preserved during it. You can turn on automatic node updates for your cluster with GKE, or you can perform them manually.

To turn on automatic node automatic upgrades read this: `https://cloud.google.com/kubernetes-engine/docs/concepts/node-auto-upgrades`.

You can also create clusters with this set to default with the `--enable-autoupgrade` command:

```
gcloud container clusters create [CLUSTER_NAME] --zone [COMPUTE_ZONE] \
 --enable-autoupgrade
```

If you'd like to update your clusters manually, you can issue specific commands. It is recommended for production systems to turn off automatic upgrades and to perform them during periods of low traffic or during maintenance windows to ensure minimal disruption for your applications. Once you build confidence in updates, you may be able to experiment with auto-upgrades.

To manually kick off a node upgrade, you can run the following command:

```
gcloud container clusters upgrade [CLUSTER_NAME]
```

If you'd like to upgrade to a specific version of Kubernetes, you can add the `--cluster-version` tag.

You can see a running list of operations on your cluster to keep track of the update operation:

```
gcloud beta container operations list
NAME TYPE ZONE TARGET STATUS_MESSAGE STATUS START_TIME END_TIME
operation-1505407677851-8039e369 CREATE_CLUSTER us-west1-a my-cluster DONE
20xx-xx-xxT16:47:57.851933021Z 20xx-xx-xxT16:50:52.898305883Z
operation-1505500805136-e7c64af4 UPGRADE_CLUSTER us-west1-a my-cluster DONE
20xx-xx-xxT18:40:05.136739989Z 20xx-xx-xxT18:41:09.321483832Z
operation-1505500913918-5802c989 DELETE_CLUSTER us-west1-a my-cluster DONE
20xx-xx-xxT18:41:53.918825764Z 20xx-xx-xxT18:43:48.639506814Z
```

You can then describe your particular upgrade operation with the following:

```
gcloud beta container operations describe [OPERATION_ID]
```

The previous command will tell you details about the cluster upgrade action:

```
gcloud beta container operations describe operation-1507325726639-981f0ed6
endTime: '20xx-xx-xxT21:40:05.324124385Z'
name: operation-1507325726639-981f0ed6
operationType: UPGRADE_CLUSTER
selfLink:
https://container.googleapis.com/v1/projects/.../kubernetes-engine/docs/zon
es/us-central1-a/operations/operation-1507325726639-981f0ed6
startTime: '20xx-xx-xxT21:35:26.639453776Z'
status: DONE
targetLink:
https://container.googleapis.com/v1/projects/.../kubernetes-engine/docs/zon
es/us-central1-a/clusters/...
zone: us-central1-a
```

Scaling the cluster

As with PaaS versus hosted clusters, you have several options for scaling up your production Kubernetes cluster.

On GKE and AKS

When upgrading a GKE cluster, all you need to do is issue a scaling command that modifies the number of instances in your minion group. You can resize the node pools that control your cluster with the following:

```
gcloud container clusters resize [CLUSTER_NAME] \
 --node-pool [POOL_NAME]
 --size [SIZE]
```

Keep in mind that new nodes are created with the same configuration as the current machines in your node pool. When additional pods are scheduled, they'll be scheduled on the new nodes. Existing pods will not be relocated or rebalanced to the new nodes.

Scaling up the AKS cluster engine is a similar exercise, where you'll need to specify the `--resource-group` node count to your required number of nodes:

```
az aks scale --name myAKSCluster --resource-group gsw-k8s-group --node-
count 1
```

DIY clusters

When you add resources to your hand-rolled Kubernetes cluster, you'll need to do more work. In order to have nodes join in as you add them automatically via a scaling group, or manually via Infrastructure as code, you'll need to ensure that automatic registration of nodes is enabled via the `--register-node` flag. If this flag is turned on, new nodes will attempt to auto-register themselves. This is the default behavior.

You can also join nodes manually, using a pre-vetted token, to your clusters. If you initialize `kubeadm` with the following token:

```
kubeadm init --token=101tester101 --kubernetes-version $(kubeadm version -o
short)
```

You can then add additional nodes to your clusters with this command:

```
kubeadm join --discovery-token-unsafe-skip-ca-verification --
token=101tester101:6443
```

Normally in a production install of `kubeadm`, you would not specify the token and need to extract it and store it from the `kubeadm init` command.

Node maintenance

If you're scaling your cluster up or down, it's essential to know how the manual process of node deregistration and draining works. We'll use the `kubectl` drain command here to remove all pods from your node before removing the node from your cluster. Removing all pods from your nodes ensures that there are not running workloads on your instance or VM when you remove it.

Let's get a list of available nodes using the following command:

```
kubectl get nodes
```

Once we have the node list, the command to drain nodes is fairly simple:

```
kubectl drain <node>
```

This command will take some time to execute, as it has to reschedule the workloads on the node onto other machines that have available resources. Once the draining is complete, you can remove the node via your preferred programmatic API. If you're merely removing the node for maintenance, you can add it back to the available nodes with the `uncordon` command:

```
kubectl uncordon <node>
```

Additional configuration options

Once you've built up an understanding of how Kubernetes cluster configuration is managed, it's a good idea to explore the additional tools that offer enhanced mechanisms or abstractions to configure the state of your clusters.

ksonnet is one such tool, which allows you to build a structure around your various configurations in order to keep many environments configured. ksonnet uses another powerful tool called Jsonnet in order to maintain the state of the cluster. ksonnet is a different approach to cluster management that's different from the Helm approach we discussed in earlier chapters, in that it doesn't define packages by dependency, but instead takes a composable prototype approach, where you build JSON templates that are rendered by the ksonnet CLI to apply state on the cluster. You start with parts that create prototypes, which becomes a component once it's configured, and those components can then get combined into applications. This helps avoid repeated code in your code base. Check it out here: `https://ksonnet.io/`.

Summary

In this chapter, we discussed how to make changes to the infrastructure that provides compute, storage, and networking capacity to your Kubernetes infrastructure, whether it be a purely public cloud platform or a hybrid installation. In observing the public cloud platforms, we discussed methods for handling underlying instance and resource instability, and strategies for running highly available workloads on partially available underlying hardware.

Additionally, we covered a key topic on how to build infrastructure using tools such as `kubeadm`, `kubectl`, and public cloud provider tools that can scale up and down your clusters.

Questions

1. Name two available local solutions for Kubernetes
2. Name three hosted solutions for Kubernetes
3. What are four of the key areas for securing your cluster?
4. What is the command to upgrade each of the major CSPs hosted Kubernetes clusters?
5. Which cloud provider has the most production ready PaaS for Kubernetes?
6. Which command is use to take a node out of rotation?
7. Which command is used to add it back in?

Further reading

If you'd like to learn more about Jsonnet, check out this link: `https://jsonnet.org/`.

Assessments

Chapter 1: Introduction to Kubernetes

1. Minikube, Google Cloud Platform, and Azure Kubernetes Service.
2. Virtual Machines, FreeBSD Jails, LXC (Linux Containers), Open VZ, and Solaris Containers.
3. Memory, filesystem CPU, threads, processes, namespaces, and memory interface files.
4. It allows companies to ship incremental updates to software. It also allows packaging of software from a developer laptop all of the way to production.
5. An account and billing set up. You'll also need to enable the API on GCE.
6. `kube-apiserver`, `etcd`, `kube-scheduler`, `kube-controller-manager`, and `cloud-controller-manager`.
7. `kops`, `kubespray`, `kubeadm`, and `bootkube`.
8. `kubeadm`.

Chapter 2: Building a Foundation with Core Kubernetes Constructs

1. HTTP, TCP, and application-level health checks
2. ReplicaSet
3. Ecosystem, interface, governance, application, and nucleus

4. Calico and Flannel
5. `rkt`, `kata`, `frakti`, `containerd`, and `runv`
6. Cluster control plane, cluster state, and cluster nodes
7. Equality-based selectors

Chapter 3: Working with Networking, Load Balancers, and Ingress

1. Communication is governed between pods, not containers. Pod communication to service is provided by the services object. K8s doesn't use NAT to communicate between containers.
2. Network address translation
3. CNI plugins that use the overlay network, or the `kubenet` plugin, which uses the bridge and host-local features.
4. Canal, Calico, Flannel, and Kube-router.
5. Pods.
6. Userspace, iptables, and ipvs.
7. Virtual IPs, service proxies, and multi-port.
8. The spec.
9. GCE, nginx, Kong, Traefik, and HAProxy.
10. Use namespaces, RBAC, container permissions, ingress rules, and clear network policing.

Chapter 4: Implementing Reliable, Container-Native Applications

1. The four use cases for Kubernetes deployments are as follows:
    - Roll out a ReplicaSet
    - Update the state of a set of pods
    - Roll back to an earlier version of a deployment
    - Scale up to accommodate cluster load
2. The selector.

3. The record flag, `--record`.
4. `ReplicationControllers`.
5. Horizontal pod autoscaling.
6. Scheduled jobs.
7. DaemonSet simply define a pod to run on every single node in the cluster or a defined subset of those nodes. This can be very useful for a number of production–related activities, such as monitoring and logging agents, security agents, and filesystem daemons.

Chapter 5: Exploring Kubernetes Storage Concepts

1. Persistent, temporary disks, cloud volumes, `emptyDir`, and `nfs`
2. `emptydir`
3. EBS volumes in AWS and disk storage on Azure
4. Different application performance or durability requirements.
5. Binding, using, reclaiming, and expanding.
6. Persistent volume claim.

Chapter 6: Application Updates, Gradual Rollouts, and Autoscaling

1. `kubectl scale --replicas`
2. `rolling-update`
3. ClientIP
4. Horizontal pod autoscaling
5. CPU and memory usage settings at minimum and maximum values
6. Helm
7. A chart

Chapter 7: Designing for Continuous Integration and Delivery

1. Node.js
2. Jenkins
3. Helm charts
4. Persistent volume
5. Installing the Jenkins plugin
6. A ReplicationController
7. npm

Chapter 8: Monitoring and Logging

1. cAdvisor and Heapster.
2. Kube-system.
3. Grafana.
4. A collector.
5. Stackdriver.
6. Good reasons to use Prometheus are as follows:
    - **Simple to operate**: It was built to run as individual servers using local storage for reliability.
    - **It's precise**: You can use a query language similar to JQL, DDL, DCL, or SQL queries to define alerts and provide a multi-dimensional view of status.
    - **Lots of libraries**: You can use more than ten languages and numerous client libraries in order to introspect your services and software.
    - **Efficient**: With data stored in an efficient, custom format both in memory and on disk, you can scale out easily with sharding and federation, creating a strong platform from which to issue powerful queries that can construct powerful data models and ad hoc tables, graphs, and alerts.

Chapter 10: Designing for High Availability and Scalability

1. Availability, responsivity, and durability.
2. Uptime is the measure of time a given system, application, network, or other logical and physical object has been *up* and available to be used by the appropriate end user.
3. The five 9s of availability.
4. It means that it fails *gracefully*.
5. **Google Kubernetes Engine** (**GKE**).
6. A set of master nodes that has the Kubernetes control plane and the `etcd` servers collocated.
7. The Workloads API.

Chapter 11: Kubernetes SIGs, Incubation Projects, and the CNCF

1. Kubernetes and Prometheus.
2. Linkerd, rkt, CNI, TUF, Vitess, CoreDNS, Jaeger, Envoy.
3. Spiffe, Spire, Open Policy Agent, Telepresence, Harbor, TiKV, Cortex, and Buildpacks. See more here: `https://www.cncf.io/sandbox-projects/`.
4. Committees are there to define meta-standards and address community-wide issues.
5. It's a great way to understand the core concepts and inner workings of Kubernetes. It's a fun way to meet other motivated, smart people. Lastly, Kubernetes, at its essence, is a community project, and relies on the contributions of its members and users.
6. SSH keys and SSL connectivity.

Chapter 12: Cluster Federation and Multi-Tenancy

1. Using federation, we can run multiple Kubernetes clusters on-premise and in one or more public cloud providers and manage applications utilizing the entire set of our organizational resources.
2. Federation allows you increase the availability and tenancy capabilities of your Kubernetes clusters.
3. Resource synchronization across clusters and multi-cluster service discovery.
4. Kubefed.
5. Federation-controller-manager and the federation-apiserver.
6. HPAs will act in a similar fashion to normal HPAs, with the same functionality and same API-based compatibility—only, with federation, the management of pods will traverse your clusters.
7. Deployments, ReplicaSets, Events, ConfigMaps, DaemonSets, Ingress, Namespaces, Secrets, and Services.

Chapter 13: Cluster Authentication, Authorization, and Container Security

1. Container repository or registry
2. Any three from the following: Node, ABAC, RBAC, Webhook
3. Privileged
4. `kubectl get secrets`

Chapter 14: Hardening Kubernetes

1. Data encryption, secrets, service discovery, compliance, RBAC, system event tracking, and trend deviation alerts
2. Stackdriver, Sysdig, Datadog, and Sensu
3. Terraform or CloudFormation and Ansible, Chef, or Puppet

4. The principle of least privilege
5. CPU, memory, and limits
6. Maximum and minimum
7. Transport Layer Security (TLS)

Chapter 15: Kubernetes Infrastructure Management

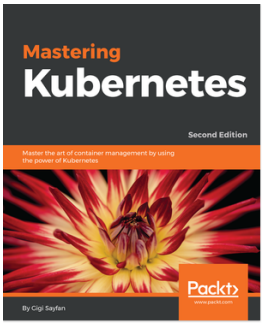
1. `kubeadm-dind`, Minikube, and Ubuntu on LXD.
2. Google Kubernetes Engine, Amazon Elastic Container Services, Azure Kubernetes Service, and Stackpoint.io.
3. Logging, Authentication, Authorization, and Linux System Parameters.
4. The commands to upgrade each of the major CSPs hosted Kubernetes clusters are as follows:

```
az aks upgrade --name <CLUSTER_NAME> --resource-group
<RESOURCE_GROUP> --kubernetes-version <VERSION>

gcloud container clusters upgrade <CLUSTER_NAME>
```

5. Google Compute Platform, with EKS
6. `kubectl drain <node>`
7. `kubectl uncordon <node>`

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Kubernetes - Second Edition
Gigi Sayfan

ISBN: 978-1-78899-978-6

- Architect a robust Kubernetes cluster for long-time operation
- Discover the advantages of running Kubernetes on GCE, AWS, Azure, and bare metal
- Understand the identity model of Kubernetes, along with the options for cluster federation
- Monitor and troubleshoot Kubernetes clusters and run a highly available Kubernetes
- Create and configure custom Kubernetes resources and use third-party resources in your automation workflows
- Enjoy the art of running complex stateful applications in your container environment
- Deliver applications as standard packages

Kubernetes Cookbook - Second Edition
Hideto Saito, Hui-Chuan Chloe Lee, Ke-Jou Carol Hsu

ISBN: 978-1-78883-760-6

- Build your own container cluster
- Deploy and manage highly scalable, containerized applications with Kubernetes
- Build high-availability Kubernetes clusters
- Build a continuous delivery pipeline for your application
- Track metrics and logs for every container running in your cluster
- Streamline the way you deploy and manage your applications with large-scale container orchestration

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W