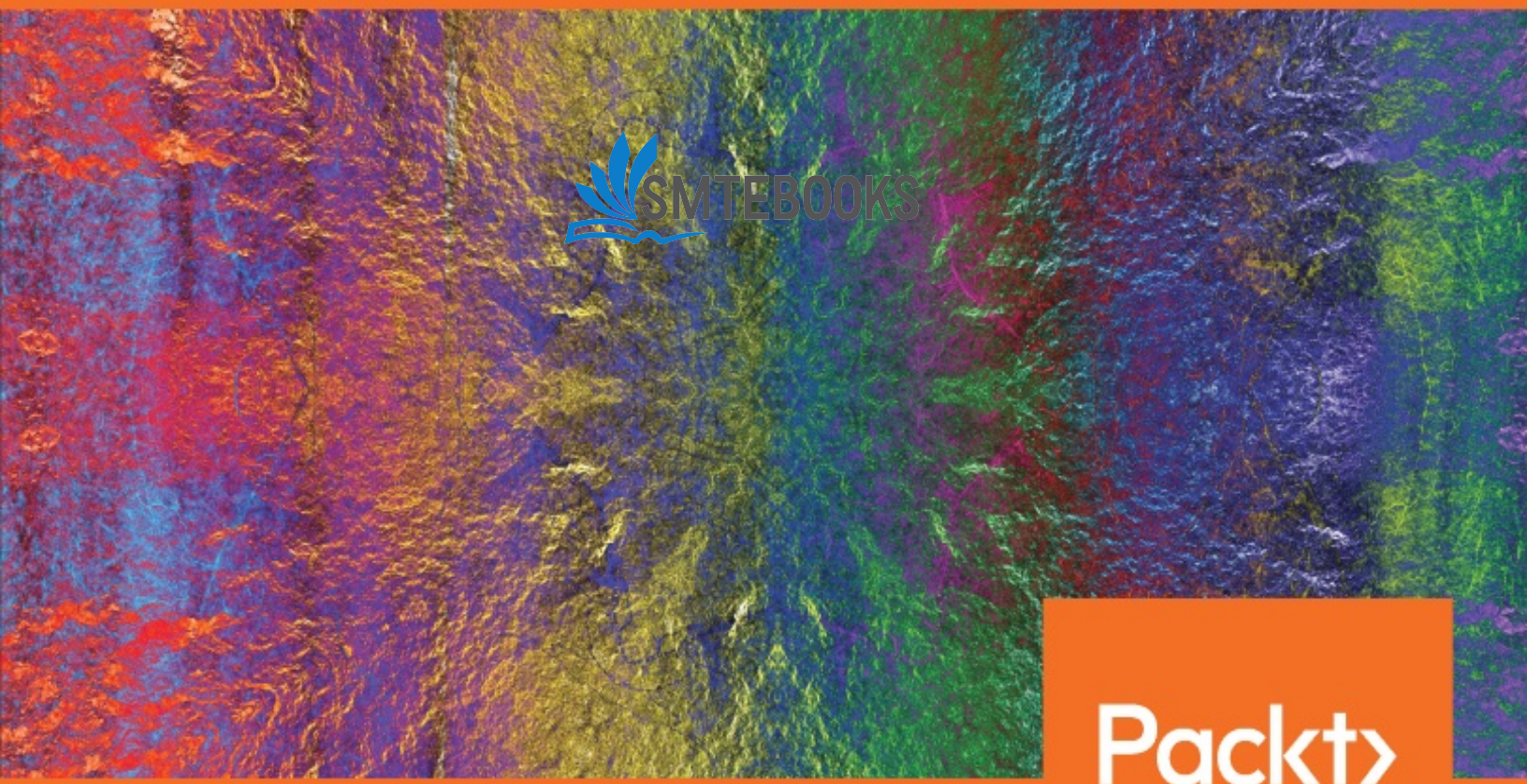
DevOps with Kubernetes

Second Edition

Accelerating software delivery with container orchestrators



Hideto Saito, Hui-Chuan Chloe Lee
and Cheng-Yang Wu

Packt
www.packt.com

DevOps with Kubernetes
Second Edition

Accelerating software delivery with container orchestrators

Hideto Saito
Hui-Chuan Chloe Lee
Cheng-Yang Wu

BIRMINGHAM - MUMBAI

DevOps with Kubernetes Second Edition

Commissioning Editor: Gebin George
Acquisition Editor: Shrilekha Inani
Content Development Editor: Deepti Thore
Technical Editor: Varsha Shivhare
Copy Editor: Safis Editing
Language Support Editor: Storm Mann, Mary McGowan
Project Coordinator: Jagdish Prabhu
Proofreader: Safis Editing
Indexer: Rekha Nair
Graphics: Jisha Chirayil
Production Coordinator: Aparna Bhagat First published: October 2017
Second edition: January 2019

Production reference: 1280119

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

ISBN 978-1-78953-399-6

www.packtpub.com

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

Hideto Saito has around 20 years of experience in the computer industry. In 1998, while working for Sun Microsystems Japan, he was impressed with Solaris OS, OPENSTEP, and Sun Ultra Enterprise 10000 (AKA StarFire). Then, he decided to pursue the UNIX and macOS X operating systems.

In 2006, he relocated to Southern California as a software engineer to develop products and services running on Linux and macOS X. He was especially renowned for his quick Objective-C code when he was drunk. He is also an enthusiast of Japanese anime, drama, and motorsports, and loves Japanese Otaku culture.

Hui-Chuan Chloe Lee is a DevOps and software developer. She has worked in the software industry on a wide range of projects for over 5 years. As a technology enthusiast, Chloe loves trying and learning about new technologies, which makes her life happier and more fulfilled. In her free time, she enjoys reading, traveling, and spending time with the people she loves.

Cheng-Yang Wu has been tackling infrastructure and system reliability since he received his master's degree in computer science from National Taiwan University. His laziness prompted him to master DevOps skills to maximize his efficiency at work so as to squeeze in writing code for fun. He enjoys cooking as it's just like working with software – a perfect dish always comes from balanced flavors and fine-tuned tastes.

About the reviewer

Guang Ya Liu is a **Senior Technical Staff Member** (**STSM**) for IBM Cloud Private and is now focusing on cloud computing, container technology, and distributed computing. He is also a member of the IBM Academy of Technology. He used to be an OpenStack Magnum Core member from 2015 to 2017, and now serves as an Istio maintainer, Kubernetes member, Kubernetes Federation V2 maintainer, and Apache Mesos committer and PMC member.

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 explains fundamental concepts and useful skills for implementing DevOps principles with containers and Kubernetes. Our journey starts by introducing the core concepts of containers and Kubernetes, and we explore the various features provided by Kubernetes, such as persisting states and data for containers, different types of workloads, cluster network, and cluster management and extension. In order to supervise the cluster activity, we implement a monitoring and logging infrastructure in Kubernetes. For better availability and efficiency, we also learn how to autoscale containers and build a continuous delivery pipeline. Lastly, we learn how to operate the hosted Kubernetes platforms from the top three major public cloud providers.

Who this book is for

This book is intended for DevOps professionals with some software development experience who want to scale, automate, and shorten software delivery time to market.

What this book covers

`Chapter 1`, *Introduction to DevOps*, walks you through the evolution from the past to what we call DevOps today, and the tools that you should know in this field. Demand for people with DevOps skills has been growing rapidly over the last few years. DevOps practices have accelerated software development and delivery speed, as well as helping business agility.

`Chapter 2`, *DevOps with Containers*, helps you learn the fundamentals of working with containers. With the increasing trend toward microservices, containers are a handy and essential tool for every DevOps practitioner because of the agility they bring to managing heterogeneous services in an uniform way.

`Chapter 3`, *Getting Started with Kubernetes*, explores the key components and API objects in Kubernetes, and how to deploy and manage containers in a Kubernetes cluster.

`Chapter 4`, *Managing Stateful Workloads*, describes pod controllers for different workloads, along with the volume management feature for maintaining the state of an application.

Chapter 5, *Cluster Administration and Extension*, navigates you through the access control features of Kubernetes, and looks at the built-in admission controllers that provide finer granularity of control over your cluster. Furthermore, we'll also learn how to build our own custom resource to extend the cluster with customized features.

`Chapter 6`, *Kubernetes Network*, explains how default networking and routing rules work in Kubernetes. We'll also learn how to expose HTTP and HTTPS routes for external access. At the end of this chapter, the network policy and service mesh features are also introduced for better resiliency.

`Chapter 7`, *Monitoring and Logging*, shows you how to monitor a resource's usage at the application, container, and node levels using Prometheus. This chapter also shows how to collect logs from your applications, the service mesh, and Kubernetes with Elasticsearch, Fluent-bit/Fluentd, and the Kibana stack.

Ensuring the service is up and healthy is one of the major responsibilities of DevOps.

Chapter 8, *Resource Management and Scaling*, describes how to leverage the core of Kubernetes, the scheduler, to scale the application dynamically, thereby efficiently utilizing the resources of our cluster.

Chapter 9, *Continuous Delivery*, explains how to build a continuous delivery pipeline with GitHub/DockerHub/TravisCI. It also explains how to manage updates, eliminate the potential impact when doing rolling updates, and prevent possible failure. Continuous delivery is an approach to speed up your time-to-market.

Chapter 10, *Kubernetes on AWS*, walks you through AWS components and explains how to provision a cluster with the AWS-hosted Kubernetes service—EKS. EKS provides lots of integration with existing AWS services. We'll learn how to utilize those features in this chapter.

Chapter 11, *Kubernetes on GCP*, helps you learn the concept of GCP and how to run your applications in GCP's Kubernetes service offering—**Google Kubernetes Engine** (**GKE**). GKE has the most native support for Kubernetes. We'll learn how to administer GKE in this chapter.

Chapter 12, *Kubernetes on Azure*, describes basic Azure components, such as Azure virtual network, Azure virtual machines, disk storage options, and much more. We'll also learn how to provision and run a Kubernetes cluster with Azure Kubernetes Service.

To get the most out of this book

This book will guide you through the methodology of software development and delivery with Docker containers and Kubernetes using macOS and public cloud services (AWS, GCP, and Azure). You will need to install minikube, AWS CLI, Cloud SDK, and the Azure CLI to run the code samples in this book.

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/DevOps-with-Kubernetes-Second-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: http://www.packtpub.com/sites/default/files/downloads/9781789533996_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: "In light of this, `cgroups` is utilized here to limit resource usage."

A block of code is set as follows: ENV key value
ENV key1=value1 key2=value2 ...

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold: metadata:
name: nginx-external
annotations:
service.beta.kubernetes.io/aws-load-balancer-type: "nlb"

Any command-line input or output is written as follows: **$ docker run -p 80:5000 busybox /bin/sh -c **
"while :; do echo -e 'HTTP/1.1 200 OK\n\ngood day'|nc -lp 5000; done"
$ curl localhost
good day

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: "After clicking Create, the console will bring us to the following view for us to explore."

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.

Introduction to DevOps

Over the past few years, the software delivery cycle has been moving increasingly fast, while at the same time application deployment has become more and more complicated. This increases the workload of all roles involved in the release cycle, including **software developers**, **Quality Assurance** (**QA**) teams, and **IT operators**. In order to deal with rapidly-changing software systems, a new concept called **DevOps** was introduced in 2009, which is dedicated to helping the whole software delivery pipeline evolve in order to make it faster and more robust.

This chapter covers the following topics:

- How has the software delivery methodology changed?
- What is a microservices architecture? Why do people choose to adopt this architecture?
- What is DevOps? How can it make software systems more resilient?

Software delivery challenges

The **Software Development Life Cycle** (**SDLC**), or the way in which we build applications and deliver them to the market, has evolved significantly over time. In this section, we'll focus on the changes made and why.

Waterfall and static delivery

Back in the 1990s, software was delivered in a static way—using a **physical** floppy disk or CD-ROM. The SDLC always took years per cycle, because it wasn't easy to (re)deliver applications to the market.

At that time, one of the major software development methodologies was the **waterfall model**. This is made up of various phases, as shown in the following

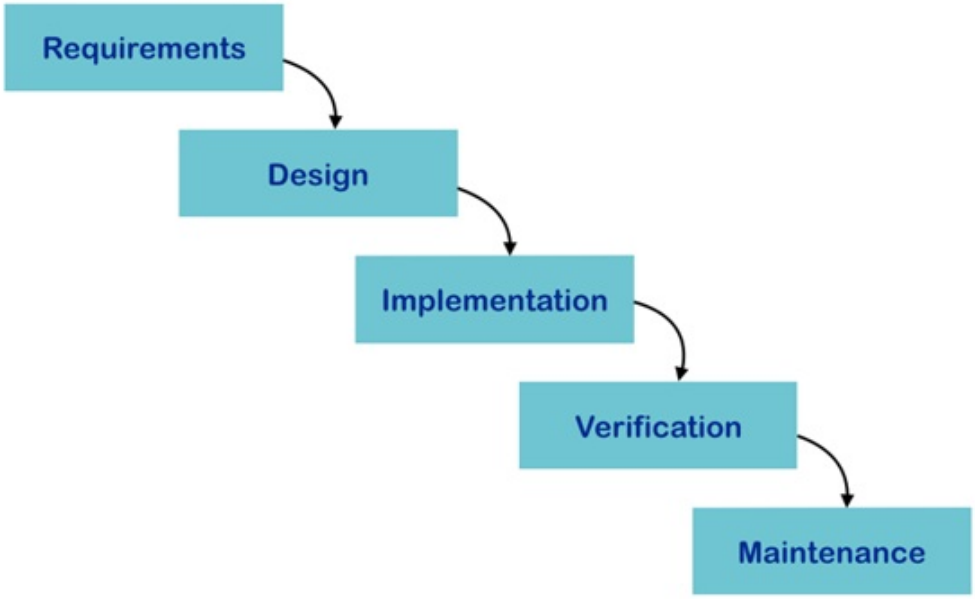


diagram:

Once one phase was started, it was hard go back to the previous phase. For example, after starting the **Implementation** phase, we wouldn't be able to go back to the **Design** phase to fix a technical expandability issue, for example, because any changes would impact the overall schedule and cost. Everything was hard to change, so new designs would be relegated to the next release cycle.

The waterfall method had to coordinate precisely with every department, including development, logistics, marketing, and distributors. The waterfall model and static delivery sometimes took several years and required tremendous effort.

Agile and digital delivery

A few years later, when the internet became more widely used, the software delivery method changed from physical to **digital**, using methods such as online downloads. For this reason, many software companies (also known as dot-com companies) tried to figure out how to shorten the SDLC process in order to deliver software that was capable of beating their competitors.

Many developers started to adopt new methodologies, such as incremental, iterative, or **agile** models, in the hope that these could help shorten the time to market. This meant that if new bugs were found, these new methods could deliver patches to customers via electronic delivery. From Windows 98, Microsoft Windows updates were also introduced in this manner.

In agile or digital models, software developers write relatively small modules, instead of the entire application. Each module is delivered to a QA team, while the developers continue to work on new modules. When the desired modules or functions are ready, they will be released as shown in the following diagram:

This model makes the SDLC cycle and software delivery faster and easily adjustable. The cycle ranges from a few weeks to a few months, which is short enough to make quick changes if necessary.

Although this model was favored by the majority at the time, application software delivery meant software binaries, often in the form of an EXE program, had to be installed and run on the customer's PC. However, the infrastructure (such as the server or the network) is very static and has to set up beforehand. Therefore, this model doesn't tend to include the infrastructure in the SDLC.

Software delivery on the cloud

A few years later, smartphones (such as the iPhone) and wireless technology (such as Wi-Fi and 4G networks) became popular and widely used. Application software was transformed from binaries to online services. The web browser became the interface of application software, which meant that it no longer requires installation. The infrastructure became very dynamic—in order to accommodate rapidly-changing application requirements, it now had to be able to grow in both capacity and performance.

This is made possible through virtualization technology and a **Software Defined Network** (**SDN**). Now, cloud services, such as **Amazon Web Services** (**AWS**), **Google Cloud Platform** (**GCP**), and **Microsoft Azure**, are often used. These can create and manage on-demand infrastructures easily.

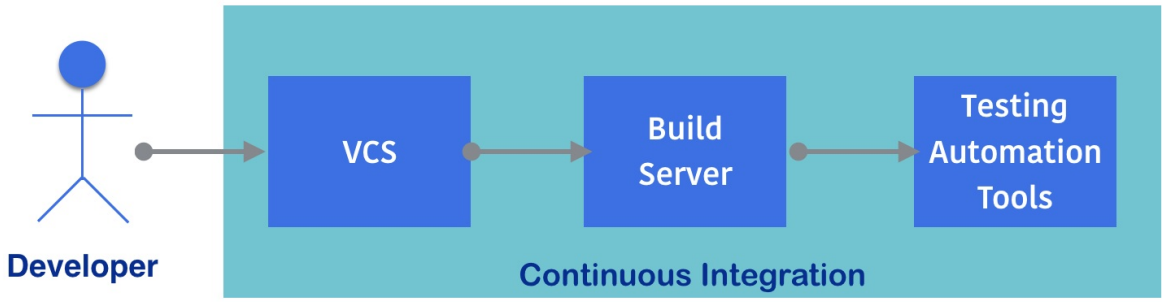
The infrastructure is one of the most important components within the scope of the **Software Development Delivery Cycle**. Because applications are installed and operated on the server side, rather than on a client-side PC, the software and service delivery cycle takes between just a few days and a few weeks.

Continuous integration

As mentioned previously, the software delivery environment is constantly changing, while the delivery cycle is getting increasingly shorter. In order to achieve this rapid delivery with a higher quality, developers and QA teams have recently started to adopt automation technologies. One of these is **Continuous Integration** (**CI**). This includes various tools, such as **Version Control Systems** (**VCSs**), **build servers**, and **testing automation tools**.

VCSs help developers keep track of the software source code changes in central servers. They preserve code revisions and prevent the source code from being overwritten by different developers. This makes it easier to keep the source code consistent and manageable for every release. Centralized build servers connect to VCSs to retrieve the source code periodically or automatically whenever the developer updates the code to VCS. They then trigger a new build. If the build fails, the build server notifies the developer rapidly. This helps the developer when someone adds broken code into the VCS. Testing automation tools are also integrated with the build server. These invoke the unit test program after the build succeeds, then notify the developer and QA team of the result. This helps to identify if somebody writes buggy code and stores it in the VCS.

The entire CI flow is shown in the following diagram:



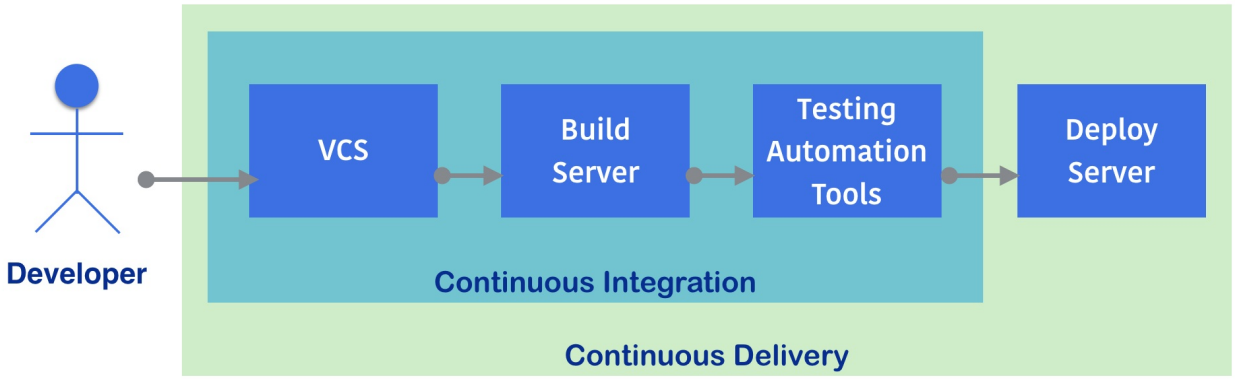
CI helps both developers and QA teams to not only increase the quality, but also shorten the process of archiving an application or a module package cycle. In the age of electronic delivery to the customer, CI is more than enough. Delivery to the customer means deploying the application to the server.

Continuous delivery

CI plus deployment automation is an ideal process for server-side applications to provide a service to customers. However, there are some technical challenges that need to be resolved, such as how to deploy the software to the server; how to gracefully shut down the existing application; how to replace and roll back the application; how to upgrade or replace system libraries that also need to be updated; and how to modify the user and group settings in the OS if necessary.

An infrastructure includes servers and networks. We normally have different environments for different software release stages, such as development, QA, staging, and production. Each environment has its own server configuration and IP ranges.

Continuous Delivery (**CD**) is a common way of resolving the previously mentioned challenges. This is a combination of CI, configuration management, and orchestration tools:



Configuration management

Configuration management tools help to configure OS settings, such as creating a user or group, or installing system libraries. It also acts as an orchestrator, which keeps multiple managed servers consistent with our desired state.

It's not a programming script, because a script is not necessarily idempotent. This means that if we execute a script twice, we might get an error, such as if we are trying to create the same user twice. Configuration management tools, however, watch the **state**, so if a user is created already, a configuration management tool wouldn't do anything. If we delete a user accidentally or even intentionally, the configuration management tool would create the user again.

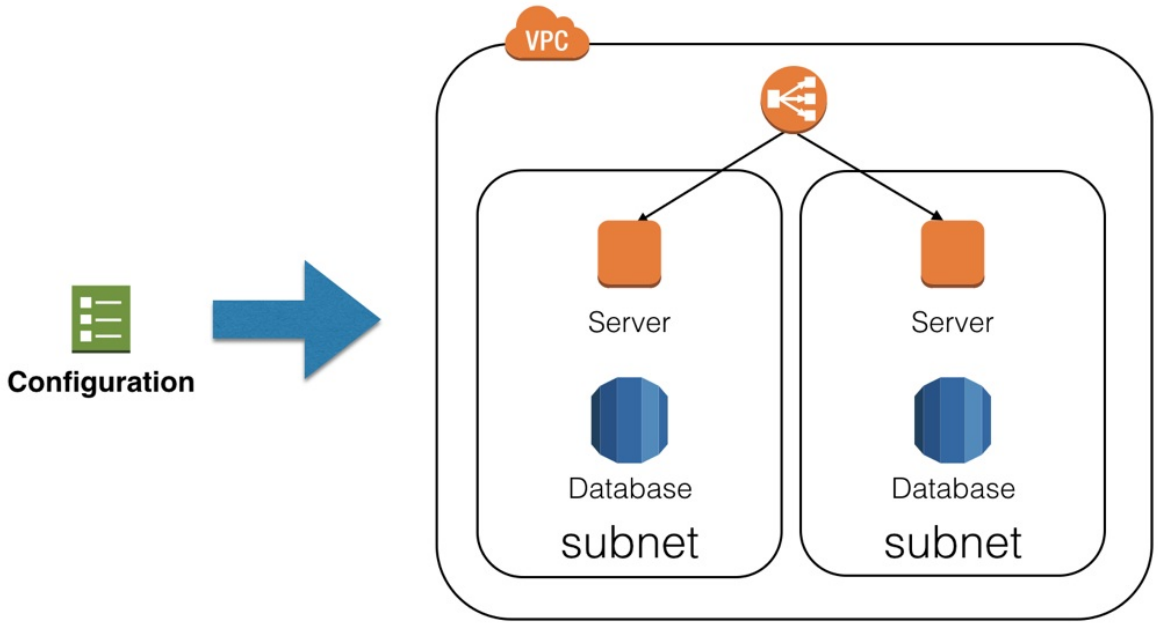
Configuration management tools also support the deployment or installation of software to the server. We simply describe what kind of software package we need to install, then the configuration management tool will trigger the appropriate command to install the software package accordingly.

As well as this, if you tell a configuration management tool to stop your application, to download and replace it with a new package (if applicable), and restart the application, it'll always be up-to-date with the latest software version. Via the configuration management tool, you can also perform blue-green deployments easily.

Blue-green deployment is a technique that prepares two sets of an application stack. Only one environment (for example, the blue one) services the production. Then, when you need to deploy a new version of the application, you can deploy it to the other side (for example, the green one), then perform the final test. If it works fine, you can change the load balancer or router setting to switch the network flow from blue to green. Then, the green side becomes the production environment, while the blue side becomes dormant and waits for the next version to be deployed.

Infrastructure as code

The configuration management tool supports not only a bare metal environment or a VM, but also cloud infrastructure. If you need to create and configure the network, storage, and VM on the cloud, the configuration management tool helps to set up the cloud infrastructure on the configuration file, as shown in the following diagram:



Configuration management has some advantages compared to a **Standard Operation Procedure** (**SOP**). It helps to maintain a configuration file via **VCS**, which can trace the history of all of the revisions.

It also helps to replicate the environment. For example, let's say we want to create a disaster recovery site in the cloud. If you follow the traditional approach, which involves using the SOP to build the environment manually, it's hard to predict and detect human or operational errors. On the other hand, if we use the configuration management tool, we can build an environment in the cloud quickly and automatically.

Infrastructure as code may or may not be included in the CD process, because the cost of replacing or updating the infrastructure is higher than simply replacing an application binary

on the server.

Orchestration

The orchestration tool is part of the configuration management tool set. However, this tool is more intelligent and dynamic with regard to configuring and allocating cloud resources. The orchestration tool manages several server resources and networks. Whenever the administrator wants to increase the application and network capacity, the orchestration tool can determine whether a server is available and can then deploy and configure the application and the network automatically. Although the orchestration tool is not included in SDLC, it helps the capacity management in the CD pipeline.

To conclude, the SDLC has evolved significantly such that we can now achieve rapid delivery using various processes, tools, and methodologies. Now, software delivery takes place anywhere and anytime, and software architecture and design is capable of producing large and rich applications.

The microservices trend

As mentioned previously, software architecture and design has continued to evolve based on the target environment and the volume of the application. This section will discuss the history and evolution of software design.

$ /usr/bin/ldd /usr/sbin/nginx

 linux-vdso.so.1 => (0x00007ffd96d79000) libdl.so.2 => /lib64/libdl.so.2 (0x00007fd96d61c000)   libpthread.so.0 => /lib64/libpthread.so.0
 (0x00007fd96d400000) libcrypt.so.1 => /lib64/libcrypt.so.1
 (0x00007fd96d1c8000) libpcre.so.1 => /lib64/libpcre.so.1 (0x00007fd96cf67000) libssl.so.10 => /lib64/libssl.so.10 (0x00007fd96ccf9000)   libcrypto.so.10 => /lib64/libcrypto.so.10
 (0x00007fd96c90e000) libz.so.1 => /lib64/libz.so.1 (0x00007fd96c6f8000)   libprofiler.so.0 => /lib64/libprofiler.so.0
 (0x00007fd96c4e4000) libc.so.6 => /lib64/libc.so.6 (0x00007fd96c122000)   ...

The `ldd(list dynamic dependencies)` command is included in the `glibc-common` package on CentOS.

Package management

The Java programming language, and several other scripting programming languages such as Python, Ruby, and JavaScript, have their own module or package management tool. Java, for example, has Maven (`http://maven.apache.org`), Python uses `pip` (`https://pip.pypa.io`), RubyGems (`https://rubygems.org`) is used for for Ruby, and `npm` is used (`https://www.npmjs.com`) for JavaScript.

Package management tools not only allow you to download the necessary packages, but can also register the module or package that you implement. The following screenshot shows the Maven repository for the AWS SDK:

mvnrepository.com/artifact/com.amazonaws/aws-java-sdk/1.11.125

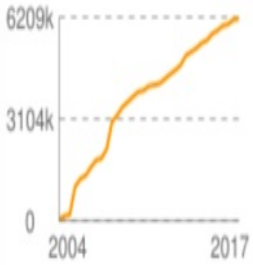
Maven Repository: com.amazonaws » aws-java-sdk » 1.11.125

MVNREPOSITORY

Search for groups, artifacts, categories

Search

Indexed Artifacts (6.22M)



Popular Categories

- Aspect Oriented
- Actor Frameworks
- Application Metrics
- Build Tools
- Bytecode Libraries
- Command Line Parsers
- Cache Implementations
- Cloud Computing
- Code Analyzers
- Collections
- Configuration Libraries
- Core Utilities
- Date and Time Utilities
- Dependency Injection

Home » com.amazonaws » aws-java-sdk » 1.11.125

AWS SDK For Java » 1.11.125

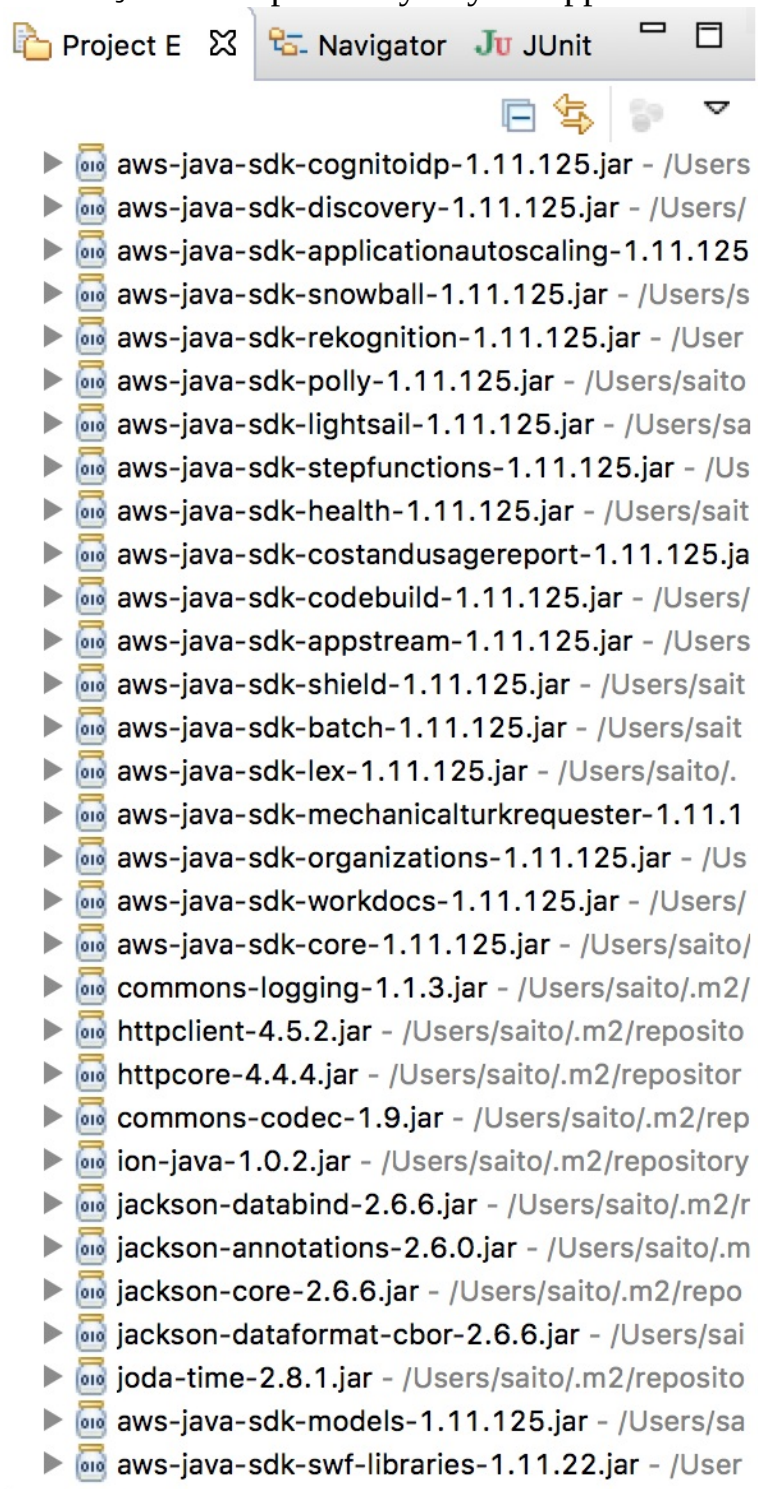
The Amazon Web Services SDK for Java provides Java APIs for building software on AWS' cost-effective, scalable, and reliable infrastructure products. The AWS Java SDK allows developers to code against APIs for all of Amazon's infrastructure web services (Amazon S3, Amazon EC2, Amazon SQS, Amazon Relational Database Service, Amazon AutoScaling, etc).

License	Apache 2.0
Categories	Cloud Computing
HomePage	https://aws.amazon.com/sdkforjava
Date	(Apr 29, 2017)
Files	Download (JAR) **(2 KB)**
Repositories	Central Sonatype Releases
Used By	**578 artifacts**

Maven | Gradle | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk -->
<dependency>
  <groupId>com.amazonaws</groupId>
  <artifactId>aws-java-sdk</artifactId>
  <version>1.11.125</version>
</dependency>
```

When you add dependencies to your application, Maven downloads the necessary packages. The following screenshot is the result you get when you add the `aws-java-sdk` dependency to your application:

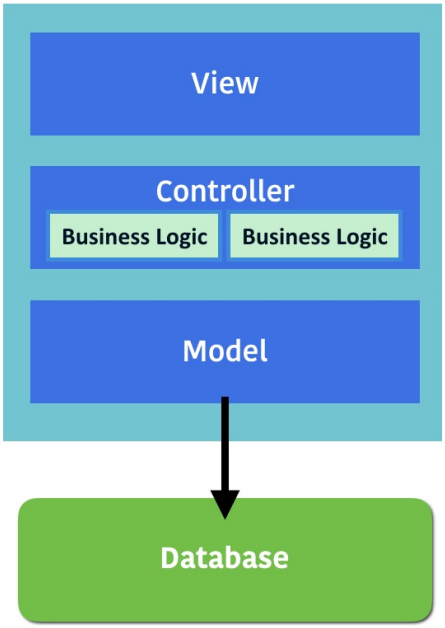


Modular programming helps you to accelerate software development speed.

However, applications nowadays have become more sophisticated. They require an ever-increasing number of modules, packages, and frameworks, and new features and logic are continuously added. Typical server-side applications usually use authentication methods such as LDAP, connect to a centralized database such as RDBMS, and then return the result to the user. Developers have recently found themselves required to utilize software design patterns in order to accommodate a bunch of modules in an application.

The MVC design pattern

One of the most popular application design patterns is **Model-View-Controller** (**MVC**). This defines three layers: the Model layer is in charge of data queries and persistence, such as loading and storing data to a database; the **View** layer is in charge of the **User Interface** (**UI**) and the **Input/Output** (**I/O**); and the **Controller** layer is in charge of business logic, which lies in between the **View** and the **Model**:



There are some frameworks that help developers to make MVC easier, such as Struts (`https://struts.apache.org/`), SpringMVC (`https://projects.spring.io/spring-framework/`), Ruby on Rails (`http://rubyonrails.org/`), and Django (`https://www.djangoproject.com/`). MVC is one of the most successful software design pattern, and is used for the foundation of modern web applications and services.

MVC defines a borderline between every layer, which allows several developers to jointly develop the same application. However, it also causes some negative side effects. The size of the source code within the application keeps getting bigger. This is because the database code (the **Model**), the presentation code (the **View**), and the business logic (the **Controller**) are all within the same VCS

repository. This eventually has an impact on the software development cycle. This type of application is called a **monolithic** application. It contains a lot of code that builds a giant EXE or war program.

Monolithic applications

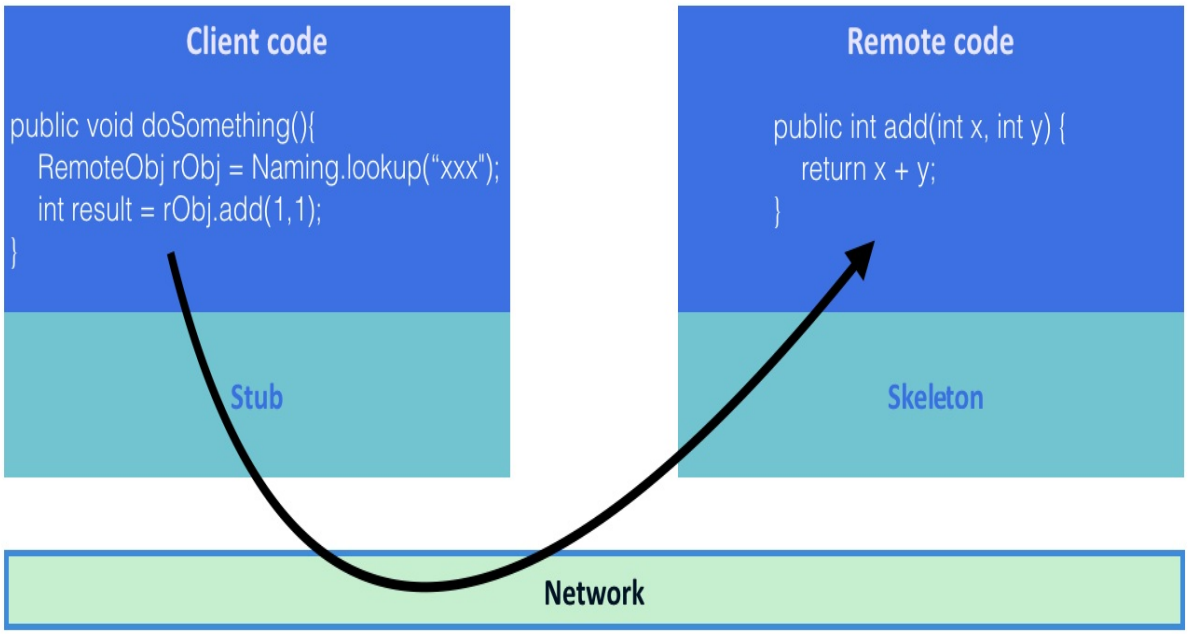
There's no concrete measurement that we can use to define an application as monolithic, but a typical monolithic app tends to have more than 50 modules or packages, more than 50 database tables, and requires more than 30 minutes to build. If we need to add or modify one of those modules, the changes made might affect a lot of code. Therefore, developers try to minimize code changes within the application. This reluctance can lead to the developer hesitation to maintain the application code, however, if problems aren't dealt with in a timely manner. For this reason, developers now tend to divide monolithic applications into smaller pieces and connect them over the network.

Remote procedure call

In fact, dividing an application into small pieces and connecting them via a network was first attempted back in the 1990s, when Sun Microsystems introduced the **Sun Remote Procedure Call** (**SunRPC**). This allows you to use a module remotely. One of most popular implementation is **Network File System** (**NFS**). The NFS client and the NFS server can communicate over the network, even if the server and the client use different CPUs and OSes.

Some programming languages also support RPC-style functionality. UNIX and the C language have the `rpcgen` tool, which generates a stub code that contains some complicated network communication code. The developer can use this over the network to avoid difficult network-layer programming.

Java has the **Java Remote Method Invocation** (**RMI**), which is similar to the Sun RPC, but specific to the Java language. The **RMI Compiler** (**RMIC**) generates the stub code that connects remote Java processes to invoke the method and return a result. The following diagram shows the procedure flow of the Java RMI:



Objective C has a **distributed object** and .NET has **remoting**, both of which work in a similar fashion. Most modern programming languages have RPC capabilities out of the box. These RPC designs are capable of dividing a single application into multiple processes (programs). Individual programs can have separate source code repositories. While the RPC designs worked well, machine resources (CPU and memory) were limited during the 1990s and early 2000s. Another disadvantage was that the same programming language was intended to be used throughout and these designs were intended to be used for a client/server model architecture, rather than a distributed architecture. In addition, there was less security consideration when these designs were developed, so they are not recommended to be used over a public network.

In the early 2000s, initiative **web services** that used **SOAP** (HTTP/SSL) as data transport were developed. These used XML for data presentation and the **Web Services Description Language** (**WSDL**) to define services. Then, **Universal Description, Discovery, and Integration** (**UDDI**) was used as the service registry to look up a web services application. However, as machine resources were not plentiful at the time and due to the complexity of programming and maintaining web services, this was not widely accepted by developers.

Nowadays, gRPC (https://grpc.io/) has led to a complete reevaluation of programming

techniques because gRPC is a simple, secure, multi-language support.

RESTful design

In the 2010s, machines and even smartphones were able to access plenty of CPU resources, and network bandwidths of a few hundred Mbps were everywhere. Developers started to utilize these resources to make application code and system structures as easy as possible, making the software development cycle quicker.

Nowadays, there are sufficient hardware resources available, so it makes sense to use HTTP/SSL as the RPC transport. In addition, from experience, developers choose to make this process easier as follows:

- By making HTTP and SSL/TLS as standard transport
- By using HTTP method for **Create**/**Load**/**Upload**/**Delete** (**CLUD**) operation, such as `GET`, `POST`, `PUT`, or `DELETE`
- By using the URI as the resource identifier, the user with the ID `123`, for example, would have the URI of `/user/123/`
- By using JSON for standard data presentation

These concepts are known as **Representational State Transfer** (**RESTful**) design. They have been widely accepted by developers and have become the de facto standard of distributed applications. RESTful applications allow the use of any programming language, as they are HTTP-based. It is possible to have, for example, Java as the RESTful server and Python as the client.

RESTful design brings freedom and opportunities to the developer. It makes it easy to perform code refactoring, to upgrade a library, and even to switch to another programming language. It also encourages the developer to build a distributed modular design made up of multiple RESTful applications, which are called microservices.

If you have multiple RESTful applications, you might be wondering how to manage multiple source codes on VCS and how to deploy multiple RESTful servers. However, CI and CD automation makes it easier to build and deploy multiple RESTful server applications. For this reason, the microservices design is becoming increasingly popular for web application developers.

Microservices

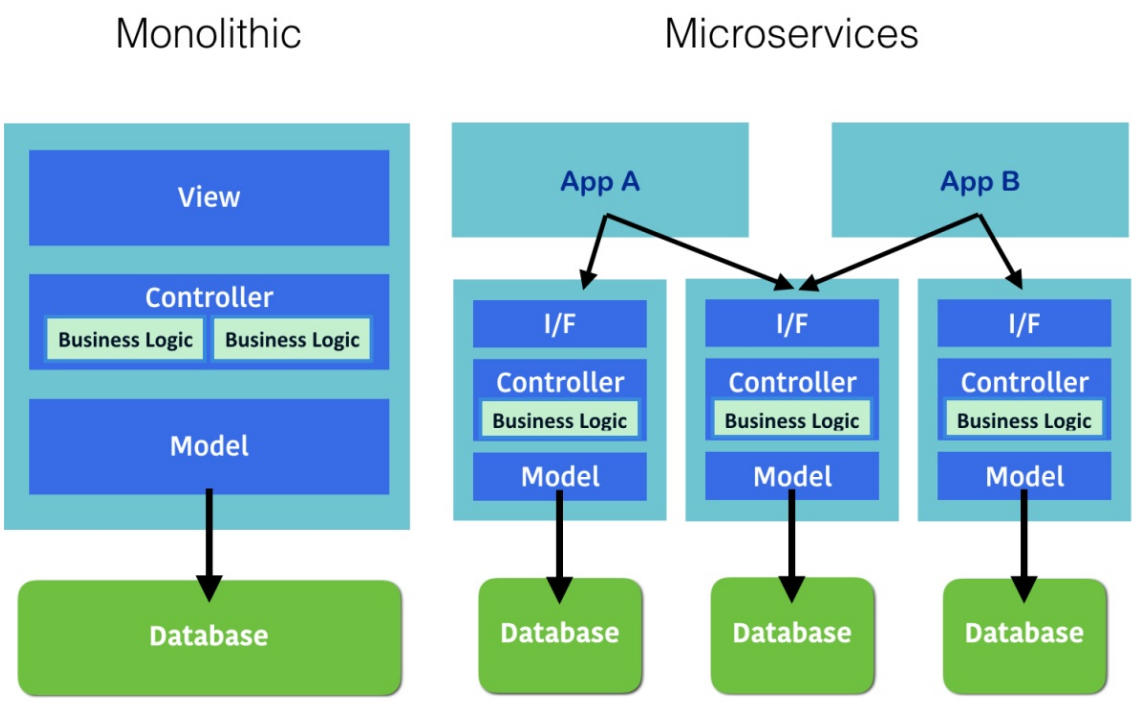
Although microservices have the word micro in their name, they are actually pretty heavy compared to applications from the 1990s or early 2000s. They use full stack HTTP/SSL servers and contain entire MVC layers.

The microservices design has the following advantages:

- **Stateless**: They don't store user sessions to the system, which helps to scale the application.
- **No shared data store**: Microservices should have their own data stores, such as databases. They shouldn't share these with other applications. They help to encapsulate the backend database so that it is easier to refactor and update the database scheme within a single microservice.
- **Versioning and compatibility**: Microservices may change and update the API, but they should define versions, such as `/api/v1` and `/api/v2`, that have backward compatibility. This helps to decouple other microservices and applications.
- **Integrate CI/CD**: The microservice should adopt the CI and CD process to eliminate management effort.

There are some frameworks that can help to build microservice-based applications, such as Spring Boot (`https://projects.spring.io/spring-boot/`) and Flask (`http://flask.pocoo.org`). However, there're a lot of HTTP-based frameworks, so developers can feel free to choose any preferred framework or programming language. This is the beauty of the microservice design.

The following diagram is a comparison between the monolithic application design and the microservices design. It indicates that a microservice design is the same as the monolithic design; they both contain an interface layer, a business logic layer, a model layer, and a data store. The difference is, however, that the application is constructed of multiple microservices. Different applications can share the same microservices:



The developer can add the necessary microservices and modify existing microservices with a rapid software delivery method that won't affect an existing application or service. This is an important breakthrough. It represents an entire software development environment and methodology that's widely accepted by developers.

Although CI and CD automation processes help to develop and deploy microservices, the number of resources, such as VMs, OS, libraries, disk volumes, and networks, can't compare with monolithic applications. There are some tools that can support these large automation environments on the cloud.

Automation and tools

As discussed previously, automation is the best way to achieve rapid software delivery. It solves the issue of managing microservices. However, automation tools aren't ordinary IT or infrastructure applications such as **Active Directory**, **BIND** (DNS), or **Sendmail** (MTA). In order to achieve automation, we need an engineer who should have both a developer skill set to write code, particularly in scripting languages, and an infrastructure operator skill set with knowledge related to VMs, networks, and storage operations.

DevOps is short for development and operations. It refers to the ability to make automation processes such as CI, infrastructure as code, and CD. It uses some DevOps tools for these automation processes.

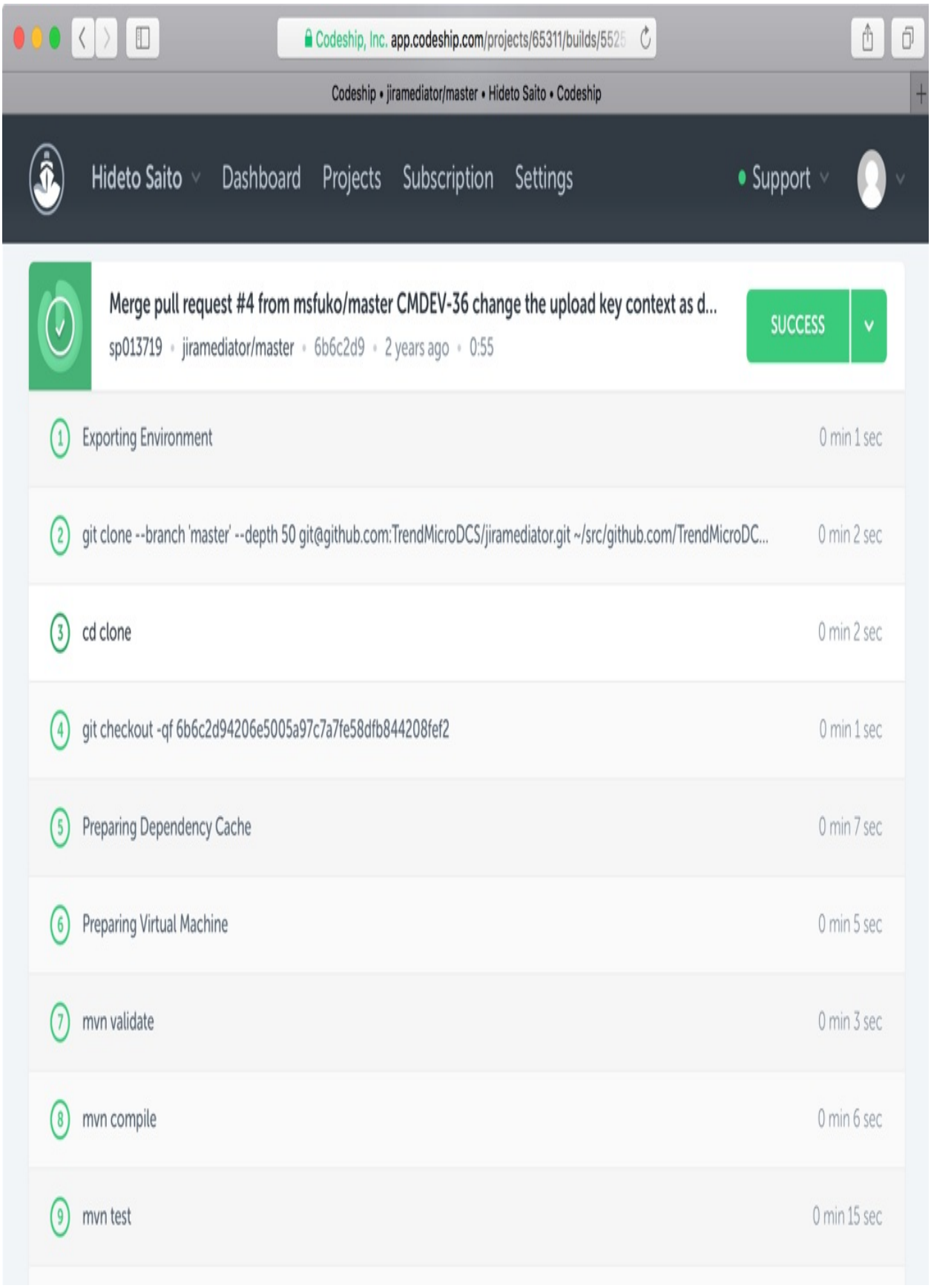
Continuous integration tools

One of the popular VCS tools is Git (https://git-scm.com). A developer uses Git to check-in and check-out code all the time. There are various hosting Git services, including GitHub (https://github.com) and Bitbucket (https://bitbucket.org). These allow you to create and save your Git repositories and collaborate with other users over the internet. The following screenshot shows a sample pull request on GitHub:


GitHub, Inc. github.com/TrendMicroDCS/opsworks-recipes/pu
Add recipes that send Omnibus logs to AWS Kinesis by hidetosaito · Pull Request #18 · TrendMicroDCS/opsworks-recipes
This repository
Search
Pull requests
Issues
Gist
TrendMicroDCS / opsworks-recipes
Unwatch 20
Star 2
Fork 6
Code
Issues 0
Pull requests 0
Projects 0
Wiki
Pulse
Graphs
Add recipes that send Omnibus logs to AWS Kinesis #18
Edit
Merged
hidetosaito merged 3 commits into TrendMicroDCS:master from hidetosaito:master on Nov 30, 2015
Conversation 0
Commits 3
Files changed 3
Changes from all commits
3 files
+61 -0
Unified
Split
Review changes
34 Omnibus/recipes/aws-kinesis-agent.rb
View
@@ -0,0 +1,34 @@
+remote_file "#{Chef::Config[:file_cache_path]}/aws-kinesis-agent-1.0-1.amzn1.noarch.rpm" do
+ source "https://s3.amazonaws.com/streaming-data-agent/aws-kinesis-agent-1.0-1.amzn1.noarch.rpm"
+ action :create
+ notifies :install, 'rpm_package[aws-kinesis-agent]'
+end
+
+
+rpm_package "aws-kinesis-agent" do
+ source "#{Chef::Config[:file_cache_path]}/aws-kinesis-agent-1.0-1.amzn1.noarch.rpm"

The build server has a lot of variation. Jenkins (`https://jenkins.io`) is one of the most well established applications, along with TeamCity (`https://www.jetbrains.com/teamcity/`). As well as build servers, you also have hosted services, otherwise known as **Software as a Service** (**SaaS**), such as Codeship (`https://codeship.com`) and Travis CI (`https://travis-ci.org`). SaaS can integrate with other SaaS tools. The build server is capable of invoking external commands, such as unit test programs. This makes the build server a key tool within the CI pipeline.

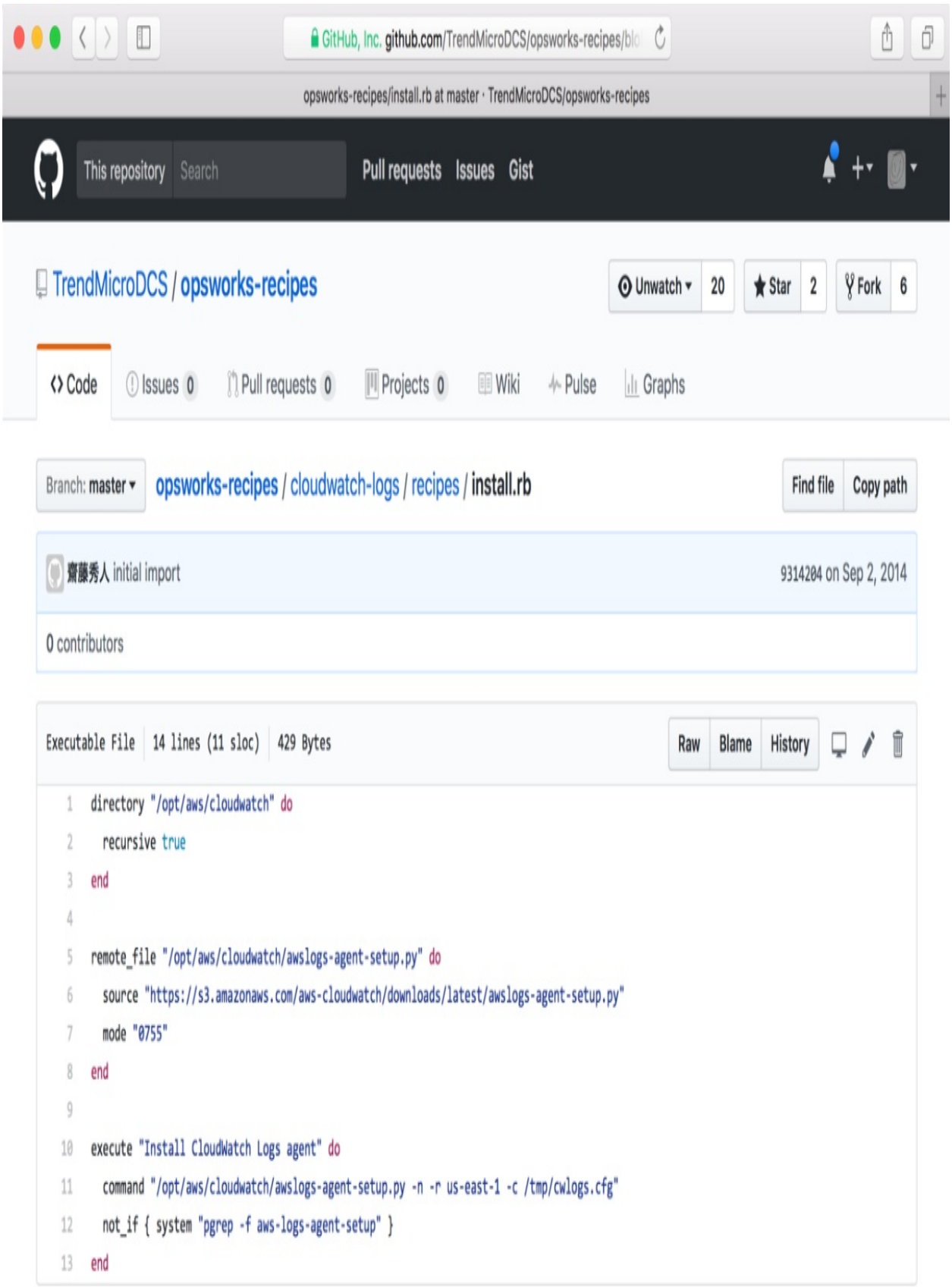
The following screenshot shows a sample build using Codeship. We check out the code from GitHub and invoke Maven for building (`mvn compile`) and unit testing (`mvn test`) our sample application:


Codeship, Inc. app.codeship.com/projects/65311/builds/5525
Codeship • jiramediator/master • Hideto Saito • Codeship
Hideto Saito
Dashboard
Projects
Subscription
Settings
Support
Merge pull request #4 from msfuko/master CMDEV-36 change the upload key context as d...
sp013719 • jiramediator/master • 6b6c2d9 • 2 years ago • 0:55
SUCCESS
1 Exporting Environment 0 min 1 sec
2 git clone --branch 'master' --depth 50 git@github.com:TrendMicroDCS/jiramediator.git ~/src/github.com/TrendMicroDC... 0 min 2 sec
3 cd clone 0 min 2 sec
4 git checkout -qf 6b6c2d94206e5005a97c7a7fe58dfb844208fef2 0 min 1 sec
5 Preparing Dependency Cache 0 min 7 sec
6 Preparing Virtual Machine 0 min 5 sec
7 mvn validate 0 min 3 sec
8 mvn compile 0 min 6 sec
9 mvn test 0 min 15 sec

Configuration management tools

There are a variety of configuration management tools available. The most popular ones include Puppet (https://puppet.com), Chef (https://www.chef.io), and Ansible (https://www.ansible.com).

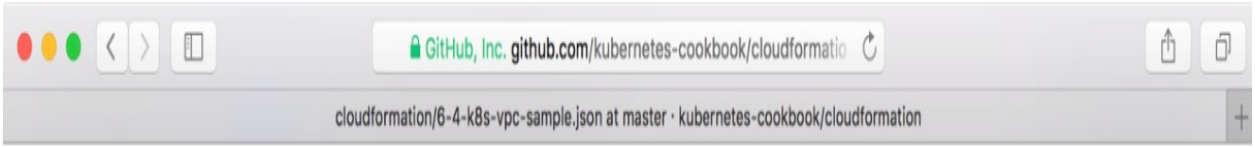
AWS OpsWorks (https://aws.amazon.com/opsworks/) provides a managed Chef platform on AWS Cloud. The following screenshot shows a Chef recipe (configuration) of an installation of the Amazon CloudWatch Log agent using AWS OpsWorks. AWS OpsWorks automates the installation of the CloudWatch Log agent when launching an EC2 instance:


GitHub, Inc. github.com/TrendMicroDCS/opsworks-recipes/blo
opsworks-recipes/install.rb at master · TrendMicroDCS/opsworks-recipes
This repository
Search
Pull requests
Issues
Gist
TrendMicroDCS / opsworks-recipes
Unwatch 20
Star 2
Fork 6
Code
Issues 0
Pull requests 0
Projects 0
Wiki
Pulse
Graphs
Branch: master
opsworks-recipes / cloudwatch-logs / recipes / install.rb
Find file
Copy path
齋藤秀人 initial import
9314204 on Sep 2, 2014
0 contributors
Executable File
14 lines (11 sloc)
429 Bytes
Raw
Blame
History
directory "/opt/aws/cloudwatch" do
recursive true
end
remote_file "/opt/aws/cloudwatch/awslogs-agent-setup.py" do
source "https://s3.amazonaws.com/aws-cloudwatch/downloads/latest/awslogs-agent-setup.py"
mode "0755"
end
execute "Install CloudWatch Logs agent" do
command "/opt/aws/cloudwatch/awslogs-agent-setup.py -n -r us-east-1 -c /tmp/cwlogs.cfg"
not_if { system "pgrep -f aws-logs-agent-setup" }
end

AWS CloudFormation (https://aws.amazon.com/cloudformation/) helps to achieve infrastructure as code. It supports the automation of AWS operations, so that we can perform the following functions:

- Creating a VPC
- Creating a subnet on VPC
- Creating an internet gateway on VPC
- Creating a routing table to associate a subnet to the internet gateway
- Creating a security group
- Creating a VM instance
- Associating a security group to a VM instance

The configuration of CloudFormation is written by JSON, as shown in the following screenshot:



msfuko Add 6-4 cloudformation template — e0562c4 on May 7, 2016

1 contributor

526 lines (526 sloc) | 12.9 KB — Raw | Blame | History

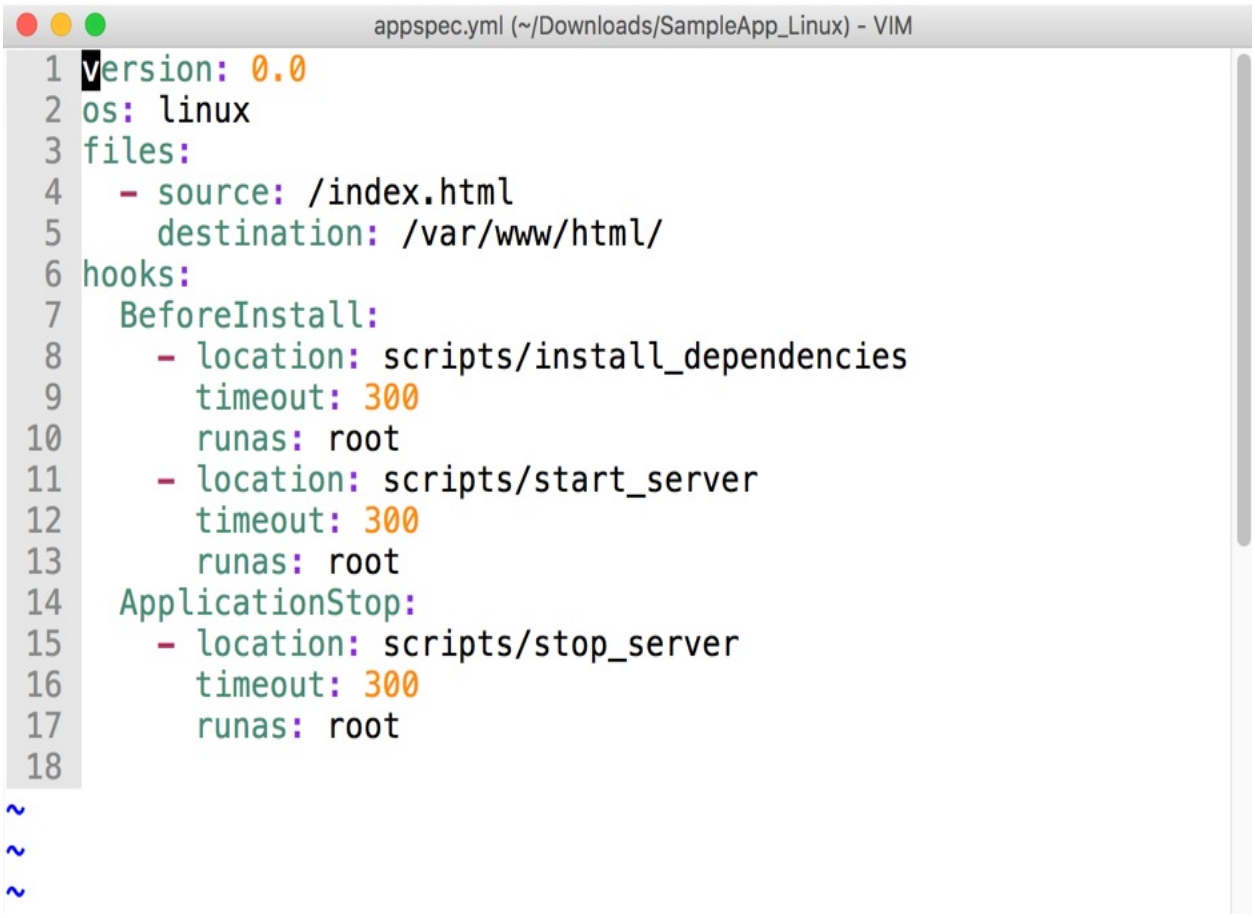
```
{
  "AWSTemplateFormatVersion":"2010-09-09",
  "Description":"Kubernetes Cookbook CloudFormation Sample - VPC",
  "Parameters":{
    "Prefix":{
      "Description":"Prefix of resources",
      "Type":"String",
      "Default":"KubernetesSample",
      "MinLength":"1",
      "MaxLength":"24",
      "ConstraintDescription":"Length is too long"
    },
    "CIDRPrefix":{
      "Description":"Network cidr prefix",
      "Type":"String",
      "Default":"10.0",
      "MinLength":"1",
      "MaxLength":"8",
      "ConstraintDescription":"Length is too long"
    }
  },
  "Resources":{
    "VPC":{
      "Type":"AWS::EC2::VPC",
      "Properties":{
        "CidrBlock":{
          "Fn::Join":[
```

CloudFormation supports parameterizing, so it's easy to create an additional environment with different parameters (such as VPC and CIDR) using a JSON file with the same configuration. It also supports the update operation. If we need to change a part of the infrastructure, there's no need to recreate the whole thing. CloudFormation can identify a delta of configuration and perform only the necessary infrastructure operations on your behalf.

AWS CodeDeploy (`https://aws.amazon.com/codedeploy/`) is another useful automation tool that focuses on software deployment. It allows the user to define the deployment steps. You can carry out the following actions on the YAML file:

- Specify where to download and install the application
- Specify how to stop the application
- Specify how to install the application
- Specify how to start and configure an application

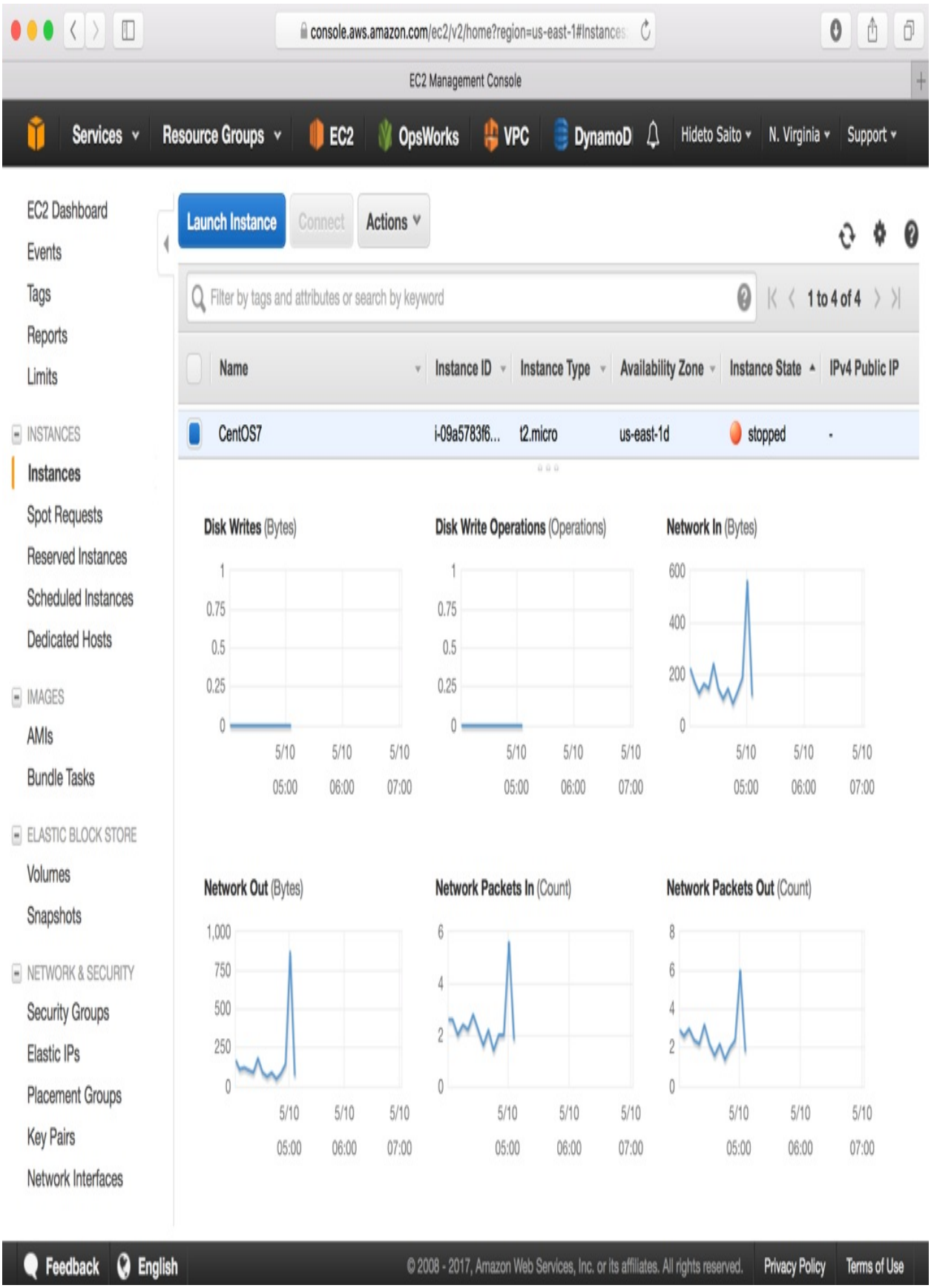
The following screenshot is an example of the AWS CodeDeploy configuration file, `appspec.yml`:


appspec.yml (~/Downloads/SampleApp_Linux) - VIM
version: 0.0
os: linux
files:
- source: /index.html
destination: /var/www/html/
hooks:
BeforeInstall:
- location: scripts/install_dependencies
timeout: 300
runas: root
- location: scripts/start_server
timeout: 300
runas: root
ApplicationStop:
- location: scripts/stop_server
timeout: 300
runas: root

Monitoring and logging tools

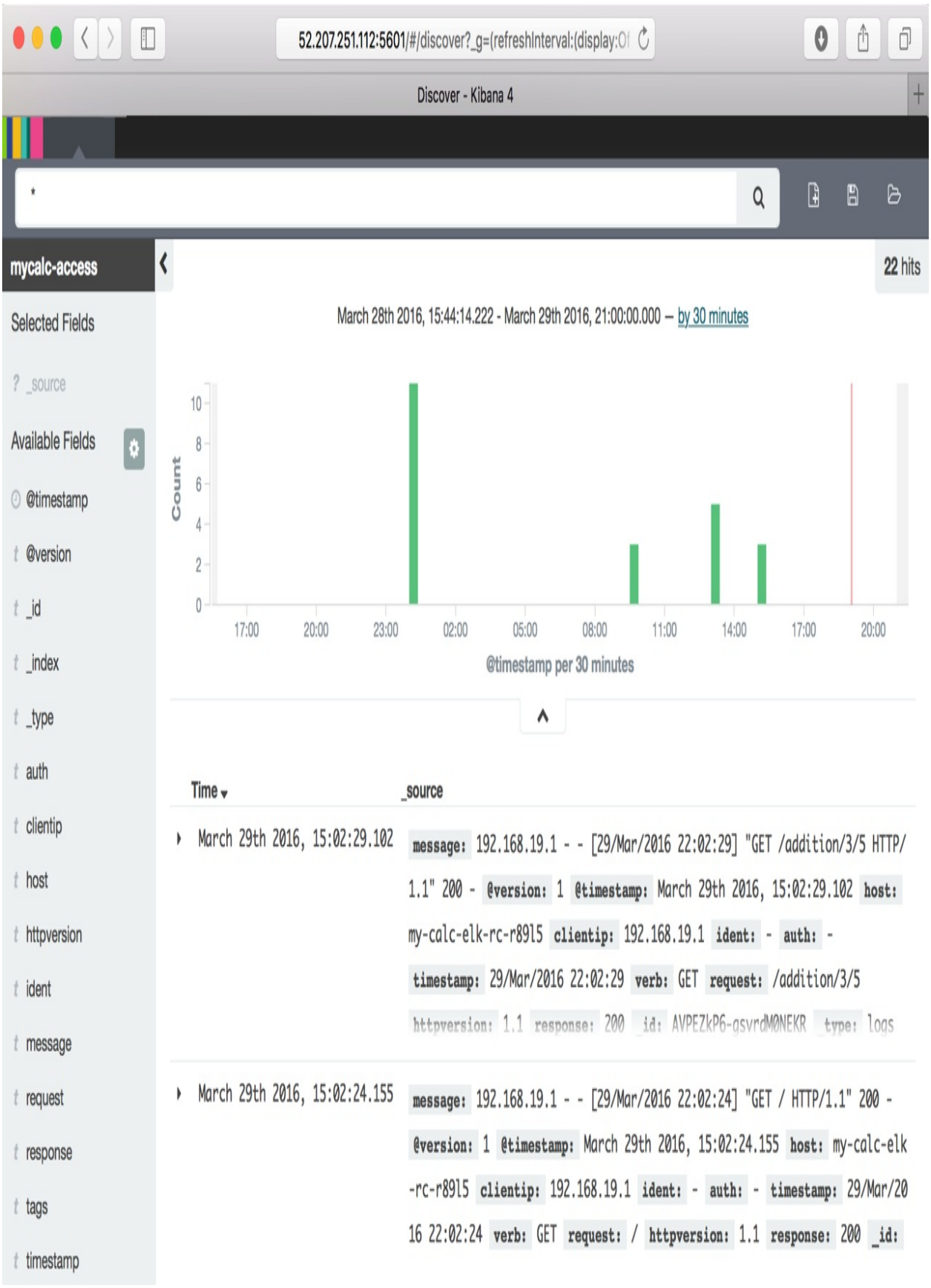
Once you start to manage microservices using a cloud infrastructure, there are various monitoring tools that can help you to manage your servers.

Amazon CloudWatch is the built-in monitoring tool for AWS. No agent installation is needed; it automatically gathers metrics from AWS instances and allows the user to visualize these in order to carry out DevOps tasks. It also supports the ability to set an alert based on the criteria that you set. The following screenshot shows the Amazon CloudWatch metrics for an EC2 instance:


console.aws.amazon.com/ec2/v2/home?region=us-east-1#Instances
EC2 Management Console
Services
Resource Groups
EC2
OpsWorks
VPC
DynamoD
Hideto Saito
N. Virginia
Support
EC2 Dashboard
Events
Tags
Reports
Limits
INSTANCES
Instances
Spot Requests
Reserved Instances
Scheduled Instances
Dedicated Hosts
IMAGES
AMIs
Bundle Tasks
ELASTIC BLOCK STORE
Volumes
Snapshots
NETWORK & SECURITY
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Launch Instance
Connect
Actions
Filter by tags and attributes or search by keyword
1 to 4 of 4
Name
Instance ID
Instance Type
Availability Zone
Instance State
IPv4 Public IP
CentOS7
i-09a5783f6...
t2.micro
us-east-1d
stopped
Disk Writes (Bytes)
Disk Write Operations (Operations)
Network In (Bytes)
Network Out (Bytes)
Network Packets In (Count)
Network Packets Out (Count)
5/10 05:00
5/10 06:00
5/10 07:00
Feedback
English
© 2008 - 2017, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

Amazon CloudWatch also supports the gathering of an application log. This requires us to install an agent on an EC2 instance. Centralized log management is useful when you need to start managing multiple microservice instances.

ELK is a popular combination of stacks that stands for Elasticsearch (`https://www.elastic.co/products/elasticsearch`), Logstash (`https://www.elastic.co/products/logstash`), and Kibana (`https://www.elastic.co/products/kibana`). Logstash aggregates the application log, transforms it to JSON format, and then sends it to Elasticsearch. Elasticsearch is a distributed JSON database. Kibana can visualize the data that's stored on Elasticsearch. The following Kibana example shows an `nginx` access log:


52.207.251.112:5601/#/discover?_g=(refreshInterval:(display:Of
Discover - Kibana 4
*
mycalc-access
22 hits
Selected Fields
? _source
Available Fields
@timestamp
@version
_id
_index
_type
auth
clientip
host
httpversion
ident
message
request
response
tags
timestamp
March 28th 2016, 15:44:14.222 - March 29th 2016, 21:00:00.000 — by 30 minutes
Count
10
8
6
4
2
0
17:00
20:00
23:00
02:00
05:00
08:00
11:00
14:00
17:00
20:00
@timestamp per 30 minutes
Time
_source
March 29th 2016, 15:02:29.102
message: 192.168.19.1 - - [29/Mar/2016 22:02:29] "GET /addition/3/5 HTTP/1.1" 200 - @version: 1 @timestamp: March 29th 2016, 15:02:29.102 host: my-calc-elk-rc-r89l5 clientip: 192.168.19.1 ident: - auth: - timestamp: 29/Mar/2016 22:02:29 verb: GET request: /addition/3/5 httpversion: 1.1 response: 200 _id: AVPEZkP6-gsvrdM0NEKR _type: logs
March 29th 2016, 15:02:24.155
message: 192.168.19.1 - - [29/Mar/2016 22:02:24] "GET / HTTP/1.1" 200 - @version: 1 @timestamp: March 29th 2016, 15:02:24.155 host: my-calc-elk-rc-r89l5 clientip: 192.168.19.1 ident: - auth: - timestamp: 29/Mar/2016 22:02:24 verb: GET request: / httpversion: 1.1 response: 200 _id:

Grafana (https://grafana.com) is another popular visualization tool. It used to be connected with time series databases such as Graphite (https://graphiteapp.org) or InfluxDB (https://www.influxdata.com). A time series database is designed to store data that's flat, de-normalized, and numeric, such as CPU usage or network traffic. Unlike RDBMS, a time series database has some optimization in order to save data space and can carry out faster queries on historical numeric data. Most DevOps monitoring tools use time series databases in the backend.

The following Grafana screenshot shows some **Message Queue Server** statistics:


Search or enter website name
Grafana - Rabbit MQ LAS
Rabbit MQ LAS
Zoom Out
Last 6 hours
Message Ack
30
25
20
15
10
5
0
20:00
22:00
00:00
lasrabbitmq01-prod Current: 4
lasrabbitmq02-prod Current: 1
lasrabbitmq03-prod Current: 1
Messages Delivered
30
25
20
15
10
5
0
20:00
22:00
2017-05-09 22:46:00
lasrabbitmq01-prod: 2
lasrabbitmq02-prod: 4
lasrabbitmq03-prod: 1
lasrabbitmq01-prod Current: 4
lasrabbitmq02-prod Current: 1
lasrabbitmq03-prod Current: 1
Messages Published
12.5
10.0
7.5
5.0
2.5
20:00
22:00
00:00
bitmq01-prod Current: 1.00
bitmq02-prod Current: 1.00
lasrabbitmq03-prod Current: 1.00
Total Messages Published
600
500
400
300
200
100
0
20:00
22:00
00:00
lasrabbitmq03-prod Current: 21 Total: 35.629 K
Total Messages Redliver
1.0
0.5
0
-0.5
-1.0
20:00
22:00
00:00
lasrabbitmq01-prod Current: 0 Total: 0
lasrabbitmq02-prod Current: 0 Total: 0
lasrabbitmq03-prod Current: 0 Total: 0
Total Messages Delivered
2.0 K
1.5 K
1.0 K
500
0
20:00
22:00
00:00
lasrabbitmq01-prod Current: 164 Total: 116.227 K
lasrabbitmq02-prod Current: 84 Total: 116.039 K
lasrabbitmq03-prod Current: 84 Total: 116.203 K

Communication tools

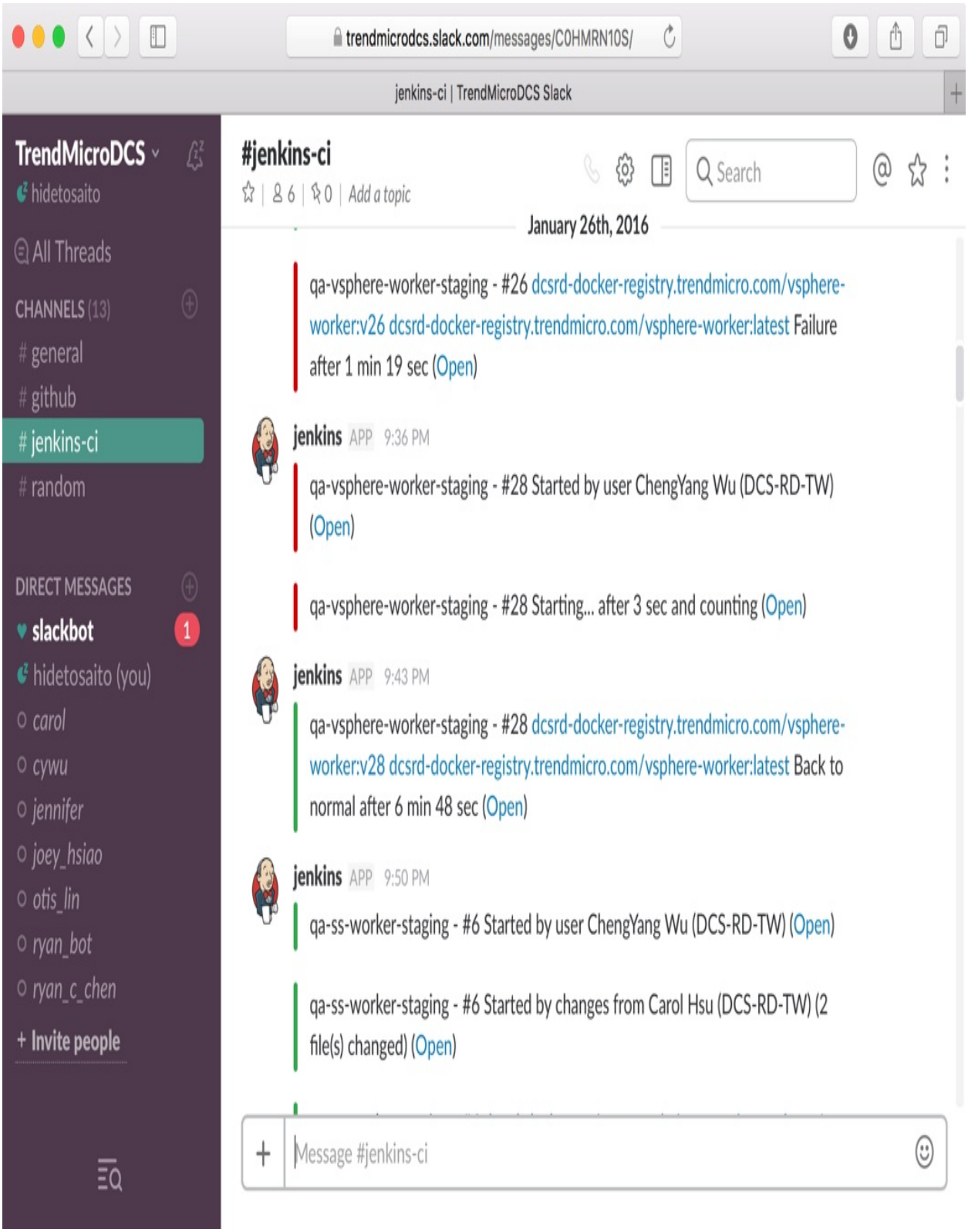
When you start to use several DevOps tools, you need to go back and forth to visit several consoles to check whether the CI and CD pipelines work properly or not. In particular, the following events need to be monitored:

- Merging the source code to GitHub
- Triggering the new build on Jenkins
- Triggering AWS CodeDeploy to deploy the new version of the application

These events need to be tracked. If there's any trouble, DevOps teams needs to discuss this with the developers and the QA team. However, communication can be a problem here, because DevOps teams are required to capture each event one by one and then pass it on as appropriate. This is inefficient.

There are some communication tools that help to integrate these different teams. They allow anyone to join to look at the events and communicate. Slack (`https://slack.com`) and HipChat (`https://www.hipchat.com`) are the most popular communication tools.

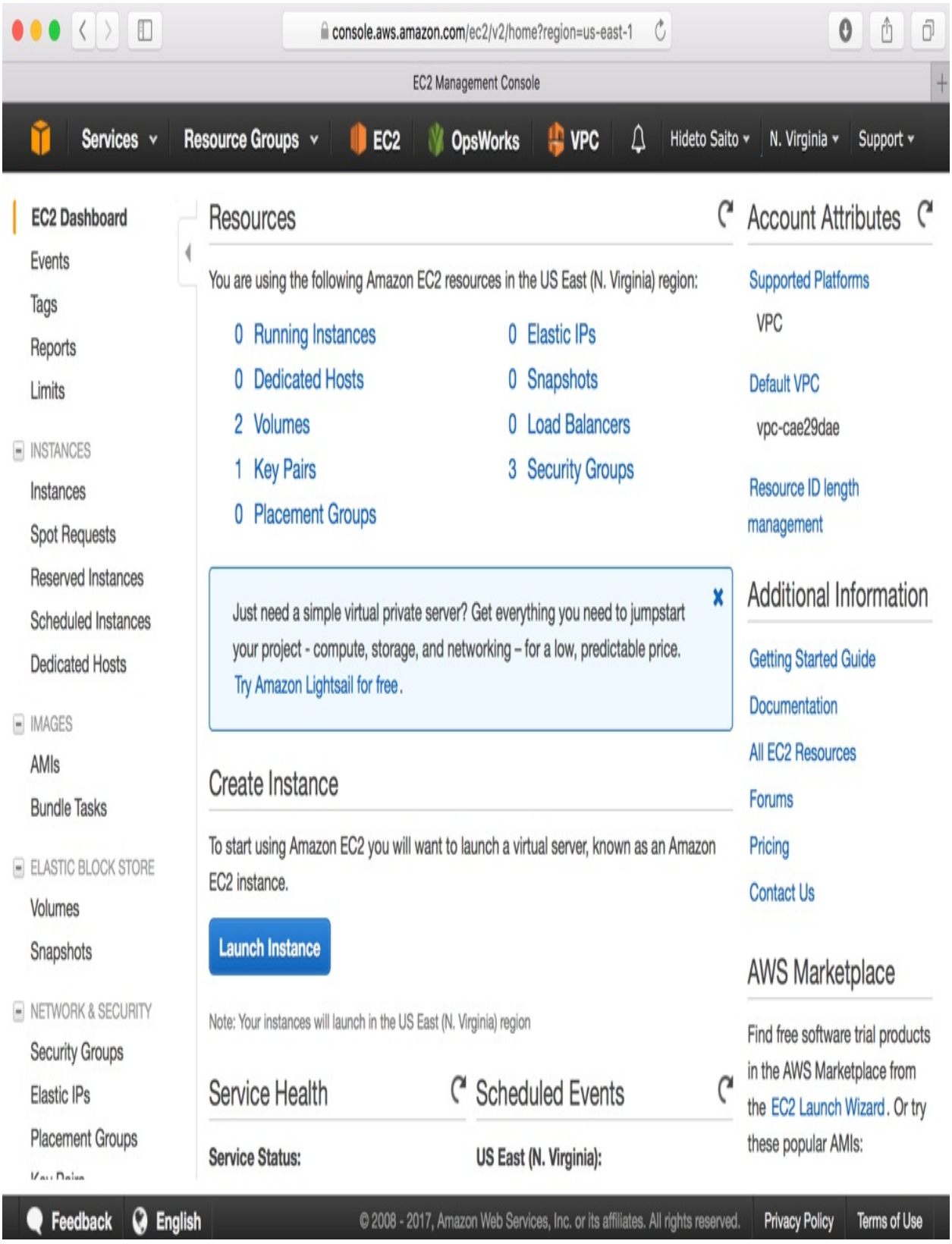
These tools also support integration with SaaS services so that DevOps teams can see events on a single chat room. The following screenshot is a Slack chat room that integrates with Jenkins:


trendmicrodcs.slack.com/messages/C0HMRN10S/
jenkins-ci | TrendMicroDCS Slack
TrendMicroDCS
hidetosaito
All Threads
CHANNELS (13)
general
github
jenkins-ci
random
DIRECT MESSAGES
slackbot
1
hidetosaito (you)
carol
cywu
jennifer
joey_hsiao
otis_lin
ryan_bot
ryan_c_chen
+ Invite people
#jenkins-ci
6
0
Add a topic
Search
January 26th, 2016
qa-vsphere-worker-staging - #26 dcsrd-docker-registry.trendmicro.com/vsphere-worker:v26 dcsrd-docker-registry.trendmicro.com/vsphere-worker:latest Failure after 1 min 19 sec (Open)
jenkins APP 9:36 PM
qa-vsphere-worker-staging - #28 Started by user ChengYang Wu (DCS-RD-TW) (Open)
qa-vsphere-worker-staging - #28 Starting... after 3 sec and counting (Open)
jenkins APP 9:43 PM
qa-vsphere-worker-staging - #28 dcsrd-docker-registry.trendmicro.com/vsphere-worker:v28 dcsrd-docker-registry.trendmicro.com/vsphere-worker:latest Back to normal after 6 min 48 sec (Open)
jenkins APP 9:50 PM
qa-ss-worker-staging - #6 Started by user ChengYang Wu (DCS-RD-TW) (Open)
qa-ss-worker-staging - #6 Started by changes from Carol Hsu (DCS-RD-TW) (2 file(s) changed) (Open)
Message #jenkins-ci

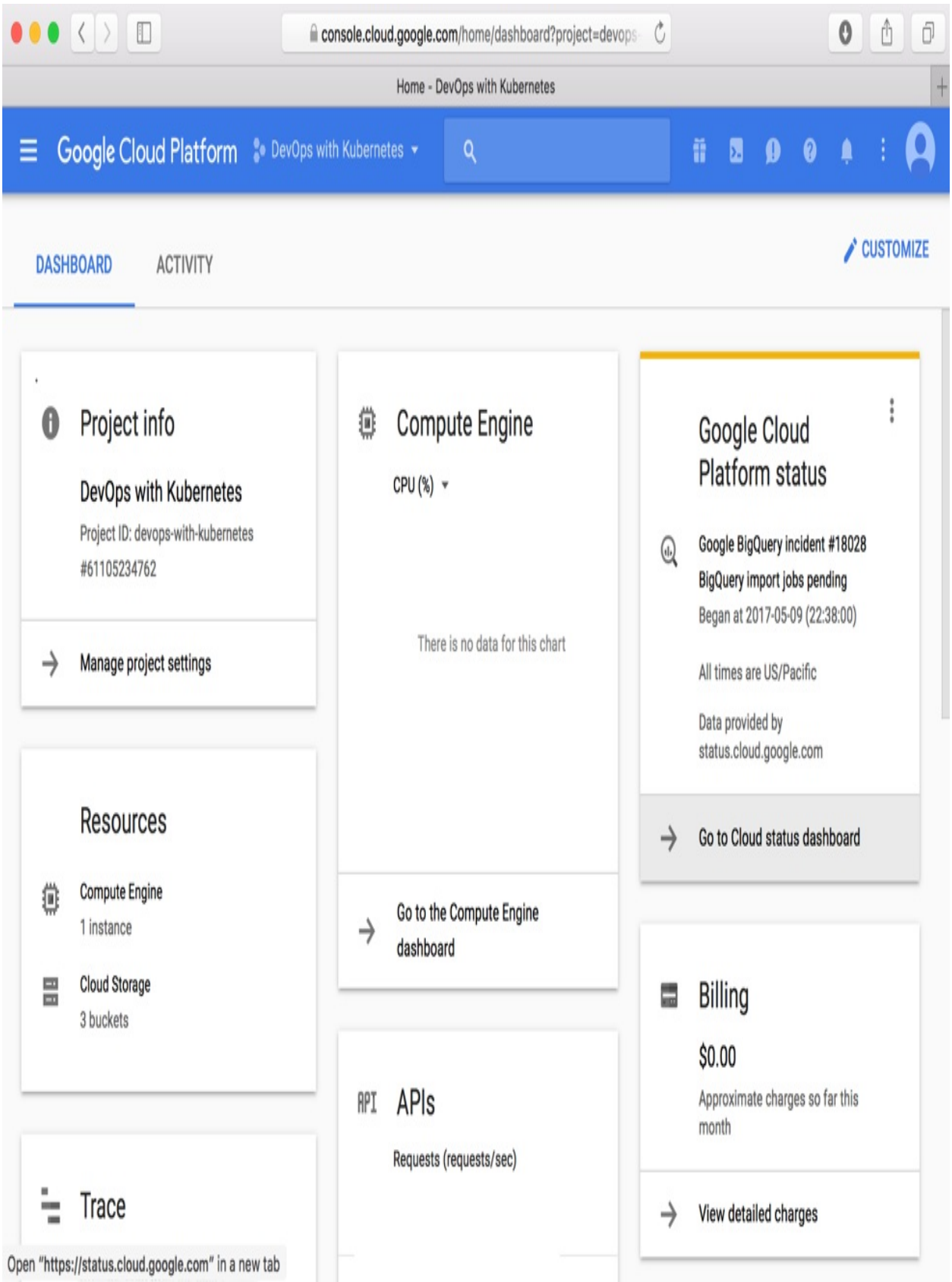
The public cloud

CI, CD, and automation work can be achieved easily when used with cloud technology. In particular, public cloud APIs help DevOps to come up with many CI and CD tools. Public clouds such as Amazon Web Services (https://aws.amazon.com), Google Cloud Platform (https://cloud.google.com), and Microsoft Azure (https://azure.microsoft.com) provide some APIs for DevOps teams to control cloud infrastructure. The DevOps can also reduce wastage of resources, because you can pay as you go whenever the resources are needed. The public cloud will continue to grow in the same way as the software development cycle and the architecture design. These are all essential in order to carry your application or service to success.

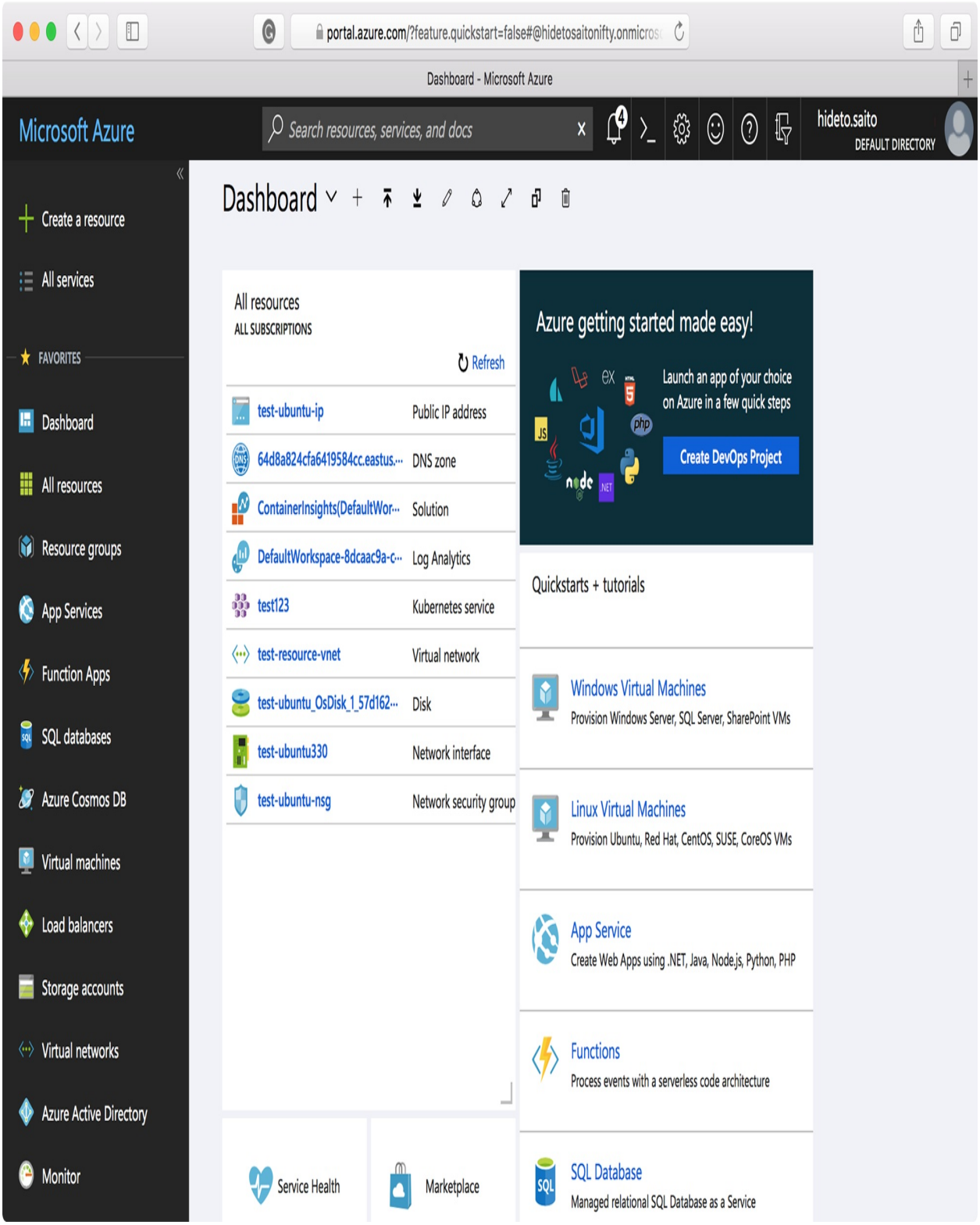
The following screenshot shows the web console for Amazon Web Services:



Google Cloud Platform also has a web console, as shown here:


console.cloud.google.com/home/dashboard?project=devops
Home - DevOps with Kubernetes
Google Cloud Platform
DevOps with Kubernetes
DASHBOARD
ACTIVITY
CUSTOMIZE
Project info
DevOps with Kubernetes
Project ID: devops-with-kubernetes
#61105234762
Manage project settings
Compute Engine
CPU (%)
There is no data for this chart
Go to the Compute Engine dashboard
Google Cloud Platform status
Google BigQuery incident #18028
BigQuery import jobs pending
Began at 2017-05-09 (22:38:00)
All times are US/Pacific
Data provided by status.cloud.google.com
Go to Cloud status dashboard
Resources
Compute Engine
1 instance
Cloud Storage
3 buckets
APIs
Requests (requests/sec)
Billing
$0.00
Approximate charges so far this month
View detailed charges
Trace
Open "https://status.cloud.google.com" in a new tab

Here's a screenshot of the Microsoft Azure console as well:



All three cloud services have a free trial period that a DevOps engineer can use

to try and understand the benefits of cloud infrastructure.

Summary

In this chapter, we've discussed the history of software development methodology, programming evolution, and DevOps tools. These methodologies and tools support a faster software delivery cycle. The microservices design also helps to produce continuous software updates. However, microservices increase the complexity of the management of an environment.

In Chapter 2, *DevOps with Containers*, we will describe the Docker container technology, which helps to compose microservice applications and manage them in a more efficient and automated way.

DevOps with Containers

We're now familiar with a wide variety of DevOps tools that can help us to automate tasks and manage configuration throughout the delivery journey of an application. Challenges still lie ahead, however, as applications have now become more diverse than ever. In this chapter, we'll add another skill to our tool belt: the container. In particular, we'll talk about the **Docker container**. In doing this, we'll seek to understand the following:

- Key concepts related to containers
- Running Docker applications
- Building Docker applications with Dockerfile
- Orchestrating multiple containers with Docker compose

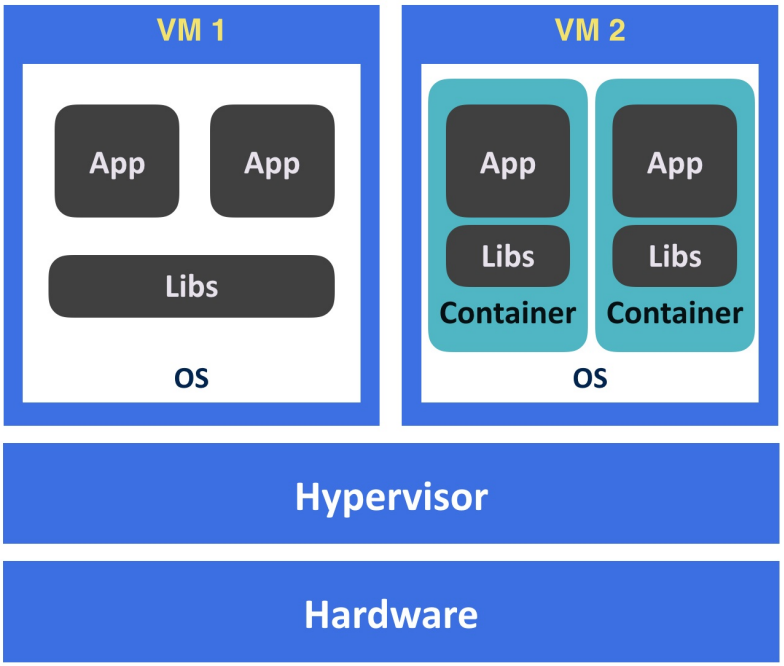
Understanding containers

One of the key features of containers is isolation. In this section, we'll establish a proper understanding of this powerful tool by looking at how a container achieves isolation and why this matters in the software development life cycle.

Resource isolation

When an application launches, it consumes CPU time, occupies memory space, links to its dependent libraries, writes to the disk, transmits packets, and may access other devices as well. Everything it uses up is a kind of resource, which is shared by all the programs on the same host. To increase the efficiency of resource utilization, we may try to put as many applications as possible on a single machine. However, the complexity involved in making every application work in a box effectively increases exponentially, even if we just want to run two applications, let alone work with tons of applications and machines. Because of this, the idea to separate the resources of a physical computing unit into isolated pieces soon became a paradigm in the industry.

You may have heard of terms such as **Virtual Machines** (**VMs**), BSD jails, Solaris containers, Linux containers, Docker, and others. All of these promise us similar isolation concepts but use fundamentally distinct mechanisms, so the actual level of isolation differs. For example, the implementation of a VM involves full virtualization of the hardware layer with a hypervisor. If you want to run an application on a VM, you have to start from a full operating system. In other words, the resources are isolated between guest operating systems running on the same hypervisor. In contrast, Linux and Docker containers are built on top of Linux primitives, which means they can only run in an operating system with those capabilities. BSD jails and Solaris containers work in a similar fashion, but on other operating systems. The following diagram illustrates the isolation relationship of the Linux container and VMs. The container isolates an application on the operating system layer, while VM-based separation is achieved by the underlying hypervisor or host operating system:


VM 1
App
App
Libs
OS
VM 2
App
Libs
Container
App
Libs
Container
OS
Hypervisor
Hardware

Linux containers

A Linux container is made up of several building blocks, the two most important of which are **namespaces** and **control groups** (**cgroups**). Both of these are Linux kernel features. Namespaces provide logical partitions of certain kinds of system resources, such as the mounting point (`mnt`), the process ID (`PID`), and the network (`net`). To further understand the concept of isolation, let's look at some simple examples on the `pid` namespace. The following examples are from Ubuntu 18.04.1 and util-linux 2.31.1.

When we type `ps axf` in our Terminal, we'll see a long list of running processes: **$ ps axf**
PID TTY STAT TIME COMMAND
2 ? S 0:00 [kthreadd]
4 ? I< 0:00 _ [kworker/0:0H]
5 ? I 0:00 _ [kworker/u2:0]
6 ? I< 0:00 _ [mm_percpu_wq]
7 ? S 0:00 _ [ksoftirqd/0]
...

`ps` is a utility that is used to report current processes on the system. `ps axf` provides a list of all processes in a forest.

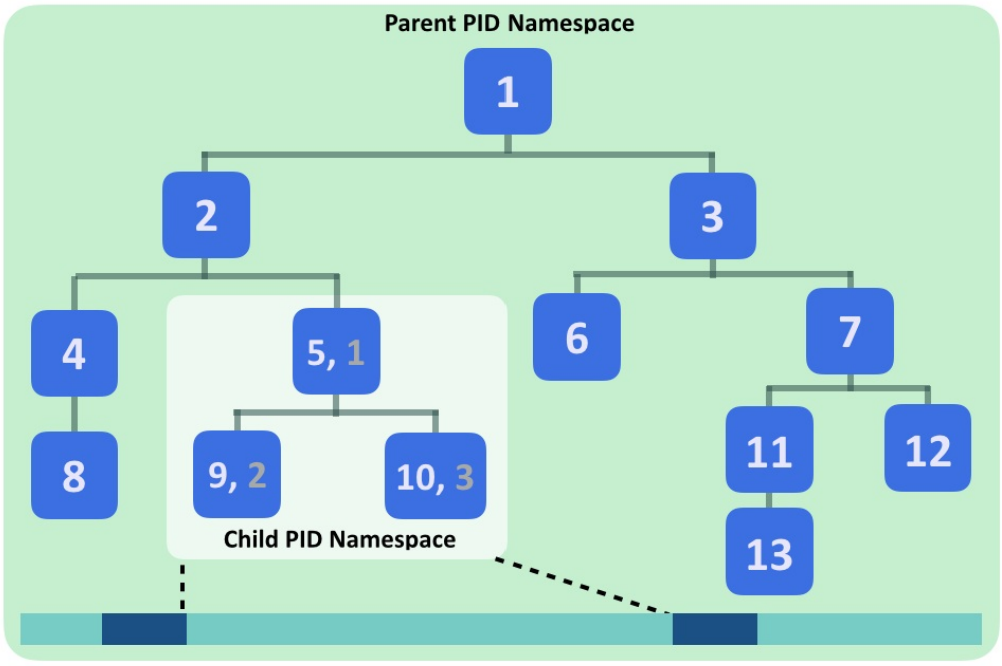
Let's now enter a new `pid` namespace with `unshare`, which is able to disassociate a process resource part by part into a new namespace. We'll then check the processes again: **$ sudo unshare --fork --pid --mount-proc=/proc /bin/sh**
 # ps axf
PID TTY STAT TIME COMMAND
1 pts/0 S 0:00 /bin/sh
2 pts/0 R+ 0:00 ps axf

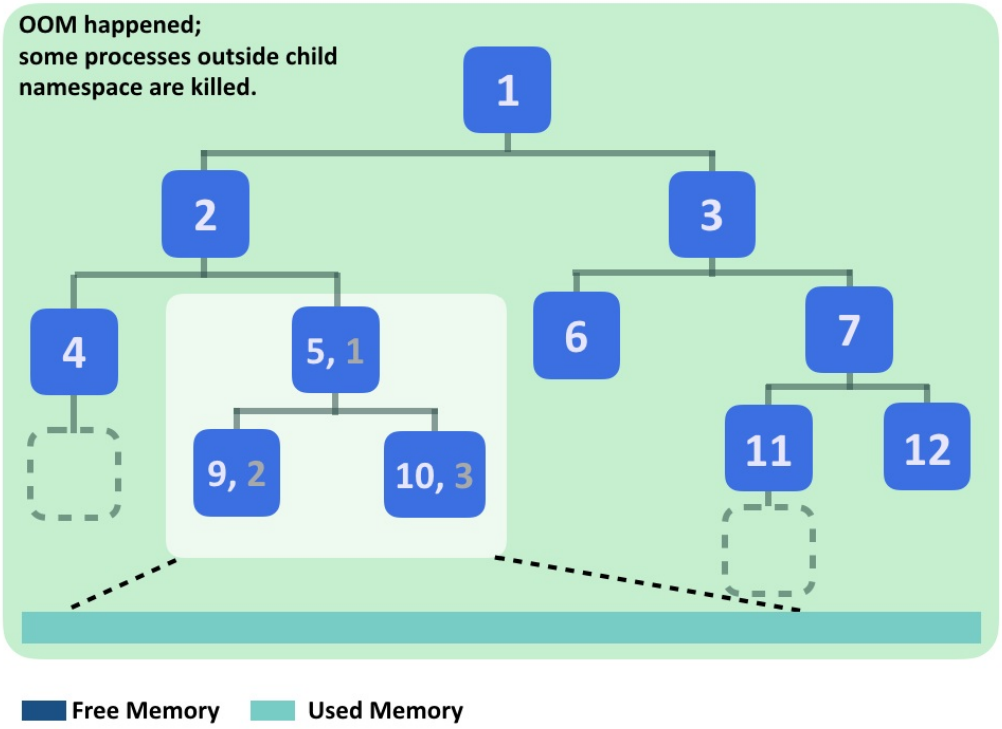
You'll find that the `pid` of the shell process at the new namespace becomes `1` and all other processes have disappeared. This means you've successfully created a `pid` container. Let's switch to another session outside the namespace and list the processes again: **$ ps axf ## from another terminal**
PID TTY STAT TIME COMMAND

...
1260 pts/0 Ss 0:00 _ -bash
1496 pts/0 S 0:00 | _ sudo unshare --fork --pid --mount-proc=/proc /bin/sh
1497 pts/0 S 0:00 | _ unshare --fork --pid --mount-proc=/proc /bin/sh
1498 pts/0 S+ 0:00 | _ /bin/sh
1464 pts/1 Ss 0:00 _ -bash
...

You can still see the other processes and your shell process within the new namespace. With the `pid` namespace's isolation, processes inhabiting different namespaces can't see each other. However, if one process uses a considerable amount of system resources, such as the memory, it could cause the system to run out of that resource and become unstable. In other words, an isolated process could still disrupt other processes or even crash the whole system if we don't impose resource usage restrictions on it.

The following diagram illustrates the `PID` namespaces and how an **Out-Of-Memory** (**OOM**) event can affect other processes outside a child namespace. The numbered blocks are the processes in the system, and the numbers are their PIDs. Blocks with two numbers are processes created with the child namespace, where the second number represents their PIDs in the child namespace. In the upper part of the diagram, there's still free memory available in the system. Later on, however, in the lower part of the diagram, the processes in the child namespace exhaust the remaining memory in the system. Due to the lack of free memory, the host kernel then starts the OOM killer to release memory, the victims of which are likely to be processes outside the child namespace. In the example here, processes **8** and **13** in the system are killed:





In light of this, `cgroups` is utilized here to limit resource usage. Like namespaces, this can impose constraints on different kinds of system resources. Let's continue

from our `pid` namespace, generate some loadon the CPU with `yes > /dev/null`, and then monitor it with `top`: **## in the container terminal**
 # yes > /dev/null & top
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
2 root 20 0 7468 788 724 R 99.7 0.1 0:15.14 yes
1 root 20 0 4628 780 712 S 0.0 0.1 0:00.00 sh
3 root 20 0 41656 3656 3188 R 0.0 0.4 0:00.00 top

Our CPU load reaches 100%, as expected. Let's now limit it with the `cgroup` CPU. `cgroups` are organized as folders under `/sys/fs/cgroup/`. First, we need to switch to the host session: **## on the host session**
$ ls /sys/fs/cgroup
blkio cpu cpuacct cpu,cpuacct cpuset devices freezer hugetlb memory net_cls net_cls,net_prio net_prio perf_event pids rdma systemd unified

Each folder represents the resources it controls. It's pretty easy to create a `cgroup` and control processes with it: just create a folder under the resource type with any name and append the process IDs you'd like to control to `tasks`. Here, we want to throttle the CPU usage of our `yes` process, so create a new folder under `cpu` and find out the `PID` of the `yes` process: **## also on the host terminal**
$ ps ax | grep yes | grep -v grep
1658 pts/0 R 0:42 yes
**$ sudo mkdir /sys/fs/cgroup/cpu/box && **
echo 1658 | sudo tee /sys/fs/cgroup/cpu/box/tasks > /dev/null

We've just added `yes` into the newly created `box` CPU group, but the policy remains unset, and the process still runs without any restrictions. Set a limit by writing the desired number into the corresponding file and check the CPU usage again: **$ echo 50000 | sudo tee /sys/fs/cgroup/cpu/box/cpu.cfs_quota_us > /dev/null**

go back to namespaced terminal, check stats with top
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
2 root 20 0 7468 748 684 R 50.3 0.1 6:43.68 yes
1 root 20 0 4628 784 716 S 0.0 0.1 0:00.00 sh
3 root 20 0 41656 3636 3164 R 0.0 0.4 0:00.08 top

The CPU usage is dramatically reduced, meaning that our CPU throttle works.

The previous two examples elucidate how a Linux container isolates system resources. By putting more confinements in an application, we can build a fully isolated box, including filesystems and networks, without encapsulating an operating system within it.

Containerized delivery

The usual way to run applications consists of the following steps:

1. Provision machines and the corresponding infrastructure resources
2. Install an operating system
3. Install system programs and application dependencies
4. Deploy the application
5. Maintain the running states of the application

The entire process is tedious and complicated, which is why we usually don't want to do it manually. The **configuration management tool**, introduced in Chapter 1, *Introduction to DevOps*, is used to eliminate most of the effort otherwise required in the delivery process. Its modular and code-based configuration design works well until application stacks grow complex and diverse. Maintaining a large configuration base is hard, especially if it's a legacy configuration that contains various hacks. Although changing configuration codes with the configuration management tool has a direct impact on the production environment, the configuration code often gets less attention than application code. Whenever we want to update an installed package, we would have to work in entangled and fragile dependencies between the system and application packages. It's not uncommon that some applications break inadvertently after upgrading an unrelated package. Moreover, upgrading the configuration management tool itself is also a challenging task.

In order to overcome this problem, immutable deployments with pre-baked VM images were introduced. This means that whenever we carry out any updates on the system or application packages, we would build a full VM image against the change and deploy it accordingly. This reduces some of the complexity, because we can test changes prior to roll-outs and we're able to customize runtimes for applications that can't share the same environments. Nevertheless, carrying out immutable deployment with VM images is costly. The overhead of booting, distributing, and running a bloated VM image is significantly larger than deploying packages.

The container, here, is a jigsaw piece that snugly fits the deployment needs. A

manifestation of a container can be managed within VCS and built into a blob image, and the image can be deployed immutably as well. This enables developers to abstract from actual resources and infrastructure engineers to avoid dependency hell. Besides, since we only need to pack up the application itself and its dependent libraries, its image size would be significantly smaller than a VM's. Consequently, distributing a container image is more economical than distributing a VM image. Additionally, we already know that running a process inside a container is basically identical to running it on its Linux host and, as such, almost no overhead will be produced. To summarize, a container is lightweight, self-contained, and almost immutable. This provides clear borders to distinguish responsibilities between applications and infrastructure.

Due to the fact that Linux containers share the same kernel, there are still potential security risks for the kernel from containers running on top of it. An emerging trend to address this concern is making running VMs as easy and efficient as operating system containers, such as ***unikernel****-based solutions or the* ***Kata container****. Another approach is inserting a mediator layer between applications and the host kernel, such as* ***gVisor****.*

Getting started with containers

There are many mature container engines such as **Docker** (https://www.docker.com) or **rkt** (https://coreos.com/rkt) that have already implemented features for production usage, so you don't need to build your own container from scratch. As well as this, the **Open Container Initiative** (https://www.opencontainers.org), an organization formed by container industry leaders, has standardized container specifications. Any implementation of standards, regardless of the underlying platform, should have similar properties, as the OCI aims to provide a seamless experience of using containers across a variety of operating systems. In fact, the core of Docker is **containerd**, which is an OCI-compatible runtime and can be used without Docker. In this book, we'll use the Docker (community edition) container engine to fabricate our containerized applications.

Installing Docker for Ubuntu

Docker requires a 64-bit version of Bionic 18.04 LTS, Artful 17.10, Xenial 16.04 LTS, or Trusty 14.04 LTS. You can install Docker with `apt-get install docker.io`, but its updates are usually slower than the Docker official repository.

Here are the installation steps from Docker (https://docs.docker.com/install/linux/docker-ce/ubuntu/):

1. Make sure you have the packages to allow `apt` repositories; if not, you can get them with the following command:

```
$ sudo apt-get update && sudo apt-get install -y apt-transport-https ca-certific
```

2. Add Docker's `gpg` key and verify whether its fingerprint matches `9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88`:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -


$ sudo apt-key fingerprint 0EBFCD88
```

3. Set up the repository of the `amd64` arch:

```
$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ub
```

4. Update the package index and install Docker CE:

```
 $ sudo apt-get update && sudo apt-get install docker-ce
```

Installing Docker for CentOS

A 64-bit version of CentOS 7 is required to run Docker. You can get the Docker package from CentOS's repository via `sudo yum install docker`, but this might be an older version. Again, the installation steps from Docker's official guide (https://docs.docker.com/install/linux/docker-ce/centos/) are as follows:

1. Install the utility to enable `yum` to use the extra repository:

```
$ sudo yum install -y yum-utils
```

2. Set up Docker's repository:

```
$ sudo yum-config-manager --add-repo https://download.docker.com/linux/centos/do
```

3. Install Docker CE and start it. If key verification is prompted, make sure it matches `060A 61C5 1B55 8A7F 742B 77AA C52F EB6B 621E 9F35`:

```
$ sudo yum install docker-ce
$ sudo systemctl start docker
```

Installing Docker for macOS

Docker wraps a micro Linux with the hypervisor framework to build a native application on macOS, which means we don't need third-party virtualization tools to use Docker on a Mac. To benefit from the hypervisor framework, you must upgrade your macOS to version 10.10.3 or more:

1. Download the Docker package and install it: `https://download.docker.com/mac/stable/Docker.dmg`.

Docker for Windows requires no third-party tools either. Check for the installation guide at the following link: *`https://docs.docker.com/docker-for-windows/install`*.

2. You're now in Docker. Try creating and running your very first Docker container. Run the command with `sudo` if you're on Linux:

```
$ docker run alpine ls
bin dev etc home lib media mnt proc root run sbin srv sys tmp usr var
```

3. You'll see that you're under a `root` directory instead of your current one. Let's check the process list again:

```
$ docker run alpine ps aux
PID   USER     TIME  COMMAND
    1 root      0:00 ps aux
```

It's isolated, as expected. You're now all ready to work with the container.

Alpine is a Linux distribution. Since it's really small in size, many people use it as their base image to build their application container. Do note, however, that it still has a few differences from mainstream Linux distributions. For example, Alpine uses *`musl libc`*, *while most distributions use* *`glibc`*.

The life cycle of a container

Using containers isn't as intuitive as most of the tools that we're used to working with, so we'll need to change the way we work. In this section, we'll go through how to use Docker so that we're able to benefit from containers.

The basics of Docker

When `docker run alpine ls` is executed, Docker carries out the following steps behind the scenes:

1. It finds the `alpine` image locally. If this is not found, Docker will try to locate and pull it from the public Docker registry to the local image storage.
2. It extracts the image and creates a container accordingly.
3. It executes the entry point defined in the image with commands, which are the arguments after the image name. In this example, the argument is `ls`. By default, the entry point is `/bin/sh -c` on Linux-based Docker.
4. When the entry point process is finished, the container then exits.

An image is an immutable bundle of code, libraries, configurations, and everything else we want to put in it. A container is an instance of an image, which is executed during runtime. You can use the `docker inspect IMAGE` and `docker inspect CONTAINER` commands to see the difference between an image and a container.

Anything launched with `docker run` would take the foreground; the `-d` option (`--detach`) enables us to run a container in the detached mode. Sometimes, we may need to enter an active container to check the image or update something inside it. To do this, we could use the `-i` and `-t` options (`--interactive` and `--tty`). If we want to interact with a detached container, we can use the `exec` and `attach` command: the `exec` command allows us to run a process in a running container, while `attach` works as its name suggests. The following example demonstrates how to use these commands:

```
$ docker run busybox /bin/sh -c "while :;do echo 'meow~';sleep 1;done"
meow~
meow~
...
```

Your Terminal should now be flooded with `meow~`. Switch to another Terminal and run `docker ps`, a command to get the status of containers, to find out the name and the ID of the container. Here, both the name and the ID are generated by Docker, and you can access a container with either of them: **$ docker ps**
CONTAINER ID IMAGE (omitted) STATUS PORTS NAMES

d51972e5fc8c busybox ... Up 3 seconds agitated_banach

$ docker exec -it d51972e5fc8c /bin/sh
/ # ps
PID USER TIME COMMAND
1 root 0:00 /bin/sh -c while :;do echo 'meow~';sleep 1;done
19 root 0:00 /bin/sh
30 root 0:00 sleep 1
31 root 0:00 ps
/ # kill -s 2 1
container terminated

As a matter of convenience, the name can be assigned upon `create` or `run` with the `--name` flag.

Once we access the container and inspect its processes, we'll see two shells: one is meowing and the other is where we are. Kill the first shell with `kill -s 2 1` inside the container and we'll see the whole container stopped as the entry point is exited. Finally, we'll list the stopped containers with `docker ps -a` and clean them up with `docker rm CONTAINER_NAME` or `docker rm CONTAINER_ID`: **$ docker ps -a**
CONTAINER ID IMAGE (omitted) STATUS PORTS NAMES
d51972e5fc8c busybox ... Exited (130) agitated_banach
$ docker rm d51972e5fc8c ## "agitated_banach" also works
d51972e5fc8c
$ docker ps -a
CONTAINER ID IMAGE (omitted) STATUS PORTS NAMES
nothing left now

Since Docker 1.13, the `docker system prune` command has been introduced, which helps us clean up stopped containers and occupied resources with ease.

The `PID 1` process is very special in UNIX-like operating systems. Regardless of what kind of process it is, it should reclaim its exited children and not take the `SIGKILL` signal. That's why the previous example uses `SIGINT` (2) instead of `SIGKILL`. Besides, most of the entry processes in a container don't handle terminated children, which may cause lots of un-reaped zombie processes in the system. If there's a need to run Docker containers in production without Kubernetes, use the `--init` flag upon `docker run`. This injects a `PID 1` process, which handles its terminated children correctly, into the container to run.

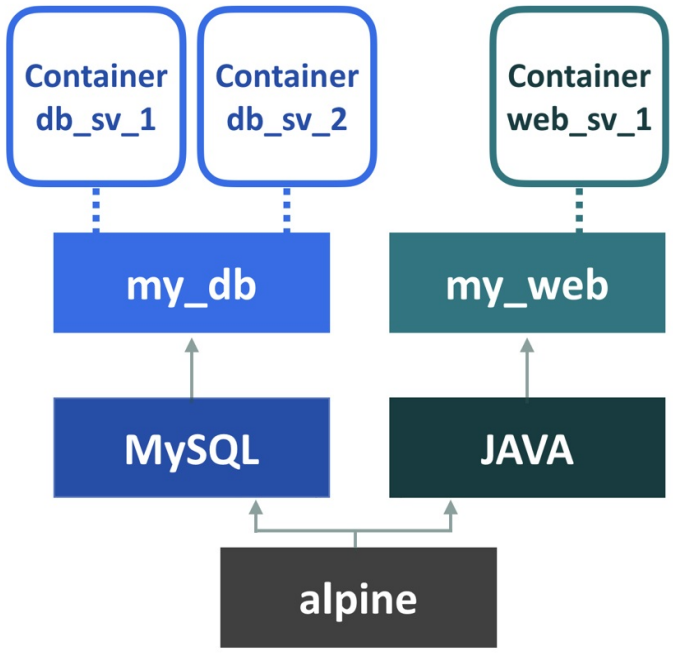
Layers, images, containers, and volumes

We know that an image is immutable and a container is ephemeral, and we know how to run an image as a container. Nevertheless, we are still missing some information with regard to packing an image.

An image is a read-only stack that consists of one or more layers, and a layer is a collection of files and directories in the filesystem. To improve disk space utilization, layers aren't locked to just one image but are shared among images, which means that Docker simply stores one copy of a base image locally, regardless of how many images are derived from it. You can utilize the `docker history [image]` command to understand how an image is built. For example, you will see that Alpine has only one layer if you check it with `docker history alpine`.

Whenever a container is created, it adds a thin, writable layer on top of the base image. Docker adopts the **Copy-On-Write** (**COW**) strategy on the thin layer. This means that a container reads the layers of the base image where the target files are stored and copies the file to its own writable layer if the file is modified. This approach prevents containers that are created from the same image from intervening with each other. The `docker diff [CONTAINER]` command shows the difference between the container and its base image in terms of their filesystem states. For example, if `/etc/hosts` in the base image is modified, Docker copies the file to the writable layer, and it'll be the only file in the output of `docker diff`.

The following diagram illustrates the hierarchical structure of Docker images:



It's important to note that data in the writable layer is deleted along with its container. To persist data, you commit the container layer as a new image with the `docker commit [CONTAINER]` command or mount data volumes into a container.

A data volume allows a container to carry out reading and writing operations, bypassing Docker's filesystem. It can either be on a host's directory or in other storage, such as Ceph or GlusterFS. Therefore, any disk I/O against the volume can operate at native speeds depending on the underlying storage. Since the data is persistent outside a container, it can be reused and shared by multiple containers. Mounting a volume is done by specifying the `-v` (`--volume`) flag with `docker run` or `docker create`. The following example mounts a volume under `/chest` in the container and leaves a file there. Afterwards, we use `docker inspect` to locate the data volume: **$ docker run --name demo -v /chest alpine touch /chest/coins**
$ docker inspect demo ## or filter outputs with --format '{{json .Mounts}}'
...
"Mounts": [
{
"Type": "volume",
"Name":"(hash-digits)",
"Source": "/var/lib/docker/volumes/(hash-digits)/_data",
"Destination": "/chest",

"Driver": "local",
"Mode": "",
"RW": true,
"Propagation": ""
}
],
...
$ ls /var/lib/docker/volumes/(hash-digits)/_data
coins

The default `tty` path of the micro Linux provided by Docker CE on macOS can be found in the following location:
`~/Library/Containers/com.docker.docker/Data/com.docker.driver.amd64-linux/tty`.
You can attach to it with `screen`.

One use case of data volumes is sharing data between containers. To do this, we first create a container and mount volumes on it, and then reference the volume with the `--volumes-from` flag when launching other containers. The following examples create a container with a data volume, `/share-vol`. Container A can put a file into it, and container B can read it: **$ docker create --name box -v /share-vol alpine nop**
7ed7c0c4426df53275c0e41798923121093b67d41bec17d50756250508f7b897
$ docker run --name AA --volumes-from box alpine touch /share-vol/wine
$ docker run --name BB --volumes-from box alpine ls /share-vol
wine

In addition, data volumes can be mounted under a given `host` path, and of course the data inside is persistent: **$ docker run --name hi -v $(pwd)/host/dir:/data alpine touch /data/hi**
$ docker rm hi
$ ls $(pwd)/host/dir
hi

Distributing images

A registry is a service that stores, manages, and distributes images. Public services, such as Docker Hub (`https://hub.docker.com`) and Quay (`https://quay.io`), collect all kinds of pre-built images of popular tools, such as Ubuntu, `nginx`, and custom images from other developers. The Alpine Linux tool we've used many times already is actually pulled from Docker Hub (`https://hub.docker.com/_/alpine`). You can upload your own tool onto these services and share them with everyone else as well.

If you need a private registry, but for some reason you don't want to subscribe to the paid plans of registry service providers, you can always set up one on your own with the Docker Registry (`https://hub.docker.com/_/registry`). Other popular registry service providers include Harbor (`https://goharbor.io/`) and Portus (`http://port.us.org/`).

Before provisioning a container, Docker will try to locate the specified image in a rule indicated in the image name. An image name consists of three sections, `[registry/]name[:tag]`, and it's resolved with the following rules:

- If the `registry` field is left out, Docker searches for the name on Docker Hub
- If the `registry` field is a registry server, Docker searches the name for it
- You can have more than one slash in a name
- The tag defaults to `latest` if it's omitted

For example, an image name such as `gcr.io/google-containers/guestbook:v3` instructs Docker to download `v3` of `google-containers/guestbook` from `gcr.io`. Likewise, if you want to push an image to a registry, tag your image in the same manner and push it with `docker push [IMAGE]`. To list the images you currently own locally, use `docker images`. You can remove an image with `docker rmi [IMAGE]`. The following example shows how to work between different registries: downloading an `nginx` image from Docker Hub, tagging it to a private registry path, and pushing it accordingly. The private registry is hosted locally with `docker run -p 5000:5000 registry`.

Here, we use the registry mentioned previously with the most basic setup. A more detailed guide about the deployment can be found at the following link: `https://docs.docker.com/registry/deploying/`.

Notice that although the default tag is `latest`, you have to tag and `push` it explicitly:

```
$ docker pull nginx
Using default tag: latest
latest: Pulling from library/nginx
802b00ed6f79: Pull complete
...
Status: Downloaded newer image for nginx:latest

$ docker tag nginx localhost:5000/comps/prod/nginx:1.15
$ docker push localhost:5000/comps/prod/nginx:1.15
The push refers to repository [localhost:5000/comps/prod/nginx]
...
8b15606a9e3e: Pushed
1.15: digest: sha256:(64-digits-hash) size: 948
$ docker tag nginx localhost:5000/comps/prod/nginx
$ docker push localhost:5000/comps/prod/nginx
The push refers to repository [localhost:5000/comps/prod/nginx]
...
8b15606a9e3e: Layer already exists
latest: digest: sha256:(64-digits-hash) size: 948
```

Most registry services ask for authentications if you're going to push images. `docker login` is designed for this purpose. For some older versions of Docker, you may sometimes receive an `image not found` error when attempting to pull an image, even though the image path is valid. This is likely to mean that you're unauthorized with the registry that keeps the image.

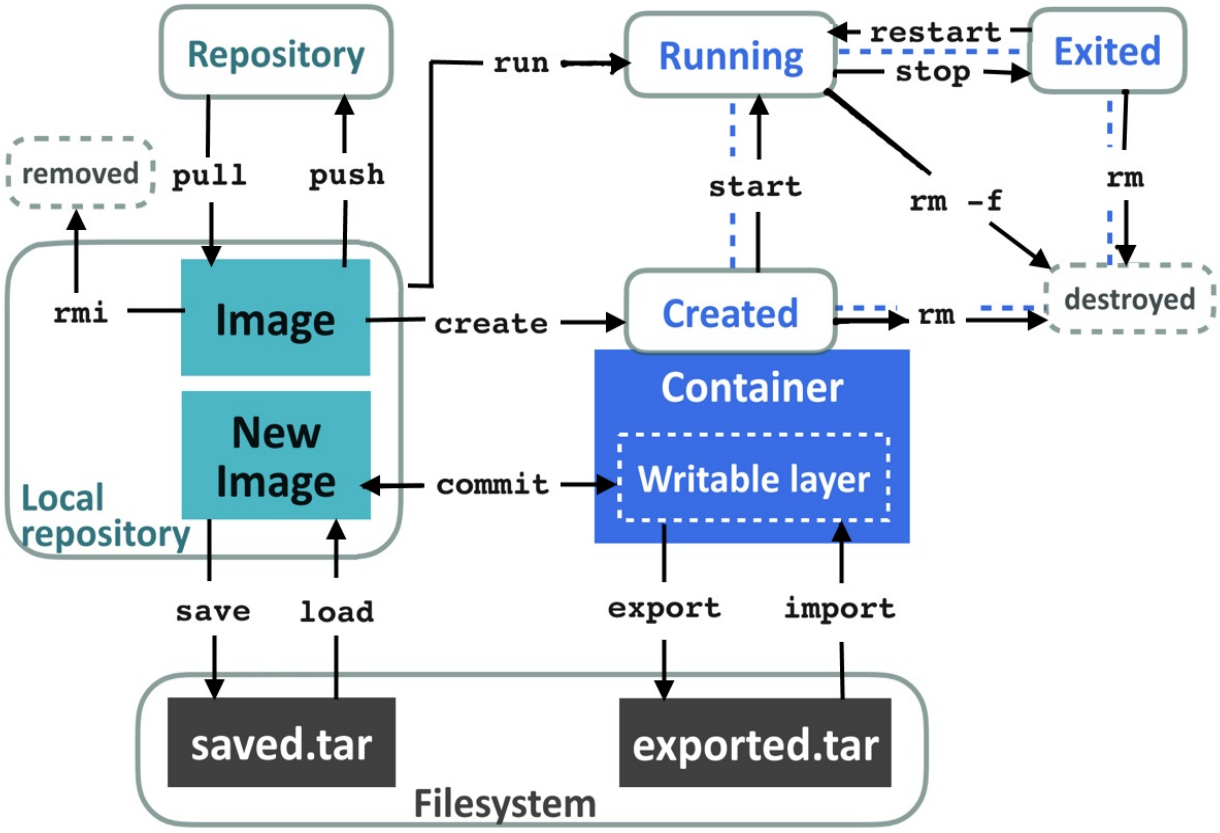
In addition to images distributed via the registry service, there are options to dump images as a TAR archive and import them back into the local repository:

- `docker commit [CONTAINER]`: Commits the changes of the container layer into a new image
- `docker save --output [filename] IMAGE1 IMAGE2 ...`: Saves one or more images to a TAR archive
- `docker load -i [filename]`: Loads a TAR image into the local repository
- `docker export --output [filename] [CONTAINER]`: Exports a container's filesystem as a TAR archive
- `docker import --output [filename] IMAGE1 IMAGE2`: Imports an exported TAR archive

The `commit`, `save`, and `export` commands look pretty much the same. The main difference is that a saved image preserves files in between layers even if they are to be deleted eventually. On the other hand, an exported image squashes all intermediate layers into one final layer. Another difference is that a saved image

keeps metadata such as layer histories, but this isn't available with an exported image. As a result, the exported image is usually smaller in size.

The following diagram depicts the relationship of states between a container and the images. The captions on the arrows are the corresponding Docker sub-commands:



The container technology is tightly bound to operating system features, which means an image built for one platform can't run on another platform without recompiling a new image on the target platform. To make this simpler, Docker introduced the Image Manifest, which supports multi-arch builds. We won't discuss multi-arch builds in this book further, but you can find more information at the following link: https://docs.docker.com/edge/engine/reference/commandline/manifest/.

Connecting containers

Docker provides three kinds of networks to manage communications between containers and the hosts, namely `bridge`, `host`, and `none`: **$ docker network ls**
NETWORK ID NAME DRIVER SCOPE
8bb41db6b13e bridge bridge local
4705332cb39e host host local
75ab6cbbbbac none null local

By default, every container is connected to the bridge network upon creation. In this mode, every container is allocated a virtual interface as well as a private IP address, and the traffic going through the interface is bridged to the host's `docker0` interface. Containers within the same bridge network can also connect to each other via their IP address. Let's run one container that's feeding a short message over port `5000`, and observe its configuration. The `--expose` flag opens the given ports to the world outside the container: **$ docker run --name greeter -d --expose 5000 busybox **
/bin/sh -c "echo Welcome stranger! | nc -lp 5000"
841138a693d220c92b8634adf97426559fd0ae622e94ac4ee9f295ab088833f5

$ docker exec greeter ifconfig
eth0 Link encap:Ethernet HWaddr 02:42:AC:11:00:02
inet addr:172.17.0.2 Bcast:172.17.255.255 Mask:255.255.0.0
...

Here, the `greeter` container is allocated to the IP address `172.17.0.2`. Now, run another container, connecting to it with this IP address:

```
$ docker run -t busybox telnet 172.17.0.2 5000
Welcome stranger!
Connection closed by foreign host
```

The `docker network inspect bridge` command provides configuration details, such as attached containers, subnet segments, and gateway information.

You can group some containers into one user-defined bridge network. This is the recommended way to connect multiple containers on a single host. The user-defined bridge network slightly differs from the default one, the major difference

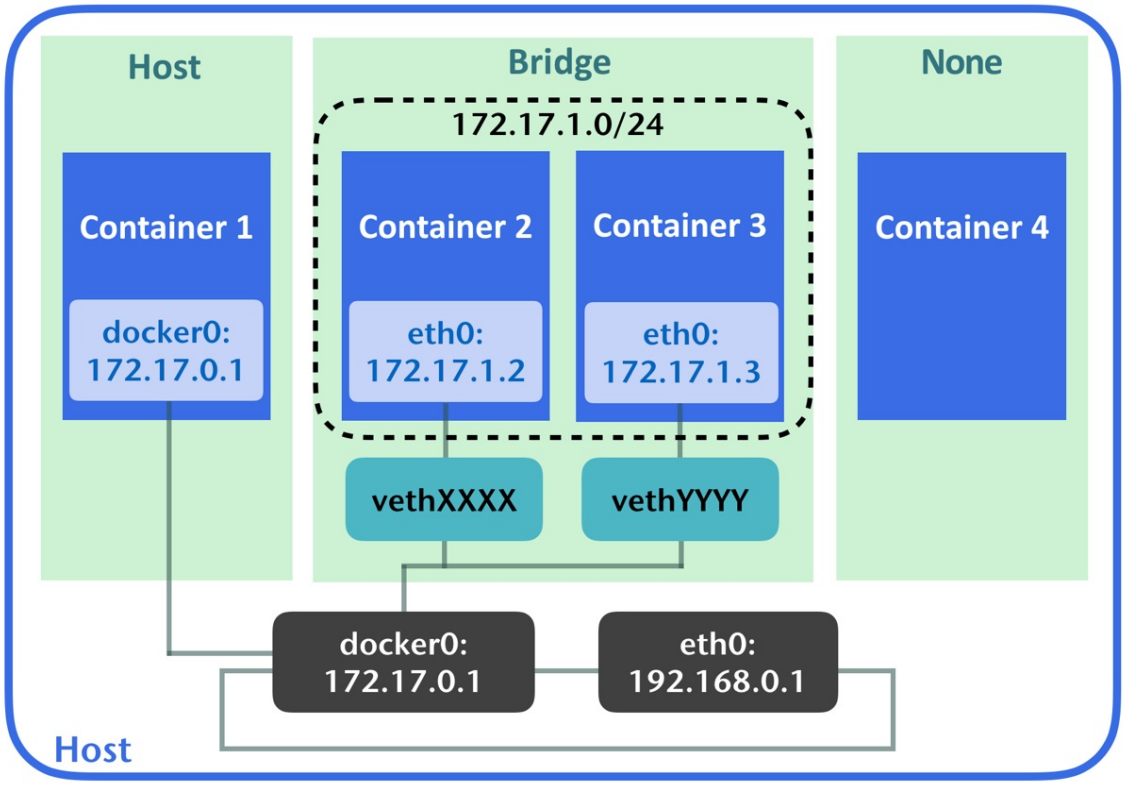
being that you can access a container from other containers with its name, rather than its IP address. Creating a network is done using the `docker network create [NW-NAME]` command, and we can attach containers to it by adding the `--network [NW-NAME]` flag at the time of creation. The network name of a container is its name by default, but it can be given another alias name with the `--network-alias` flag as well: **## create a network called "room"**
$ docker network create room
a59c7fda2e636e2f6d8af5918c9cf137ca9f09892746f4e072acd490c00c5e99
run a container with the name "sleeper" and an alias "dad" in the room
**$ docker run -d --network room **
--network-alias dad --name sleeper busybox sleep 60
56397917ccb96ccf3ded54d79e1198e43c41b6ed58b649db12e7b2ee06a69b79
ping the container with its name "sleeper" from another container
$ docker run --network room busybox ping -c 1 sleeper
PING sleeper (172.18.0.2): 56 data bytes
64 bytes from 172.19.0.2: seq=0 ttl=64 time=0.087 ms

--- sleeper ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.087/0.087/0.087 ms
ping the container with its alias "dad", it also works
$ docker run --network room alpine ping -c 1 dad
PING dad (172.18.0.2): 56 data bytes
...

The `host` network works as its name suggests; every connected container shares the host's network, but it loses the isolation property at the same time. The `none` network is a logically air-gapped box. Regardless of ingress or egress, traffic is isolated inside as there's no network interface attached to the container. Here, we attach a container that listens on port `5000` to the `host` network and communicates with it locally: **$ docker run -d --expose 5000 --network host busybox **
/bin/sh -c "echo im a container | nc -lp 5000"
a6918174c06f8f3f485913863ed69d4ae838fb550d9f4d66e24ef91534c76b3a
$ telnet localhost 5000
im a container
Connection closed by foreign host

If you are using Docker CE for macOS, the host is the micro Linux on top of the hypervisor framework.

The interaction between the host and the three network modes is shown in the following diagram. Containers in the `host` and `bridge` networks are attached with proper network interfaces and communicate with containers within the same network, as well as the outside world, but the `none` network is kept away from the host interfaces:



Other than sharing the host network, the `-p(--publish) [host]:[container]` flag, when creating a container, also allows you to map a host port to a container. We don't need attaching a `--expose` flag together with the `--publish` flag, as you'll need to open a container's port in any case. The following command launches a simple HTTP server at port `80`. You can view it with a browser as well: **$ docker run -p 80:5000 busybox /bin/sh -c **
"while :; do echo -e 'HTTP/1.1 200 OK\n\ngood day'|nc -lp 5000; done"
$ curl localhost
good day

Working with a Dockerfile

When assembling an image, whether using `docker commit` or `export`, optimizing the outcome in a managed way is a challenge, let alone integrating it with a CI/CD pipeline. A `Dockerfile` represents the building task in the form of code, which significantly reduces the difficulty of building tasks for us. In this section, we'll describe how to map Docker commands into a `Dockerfile` and take a step towards optimizing it.

Writing your first Dockerfile

A `Dockerfile` consists of a series of text instructions to guide the Docker daemon to form an image, and a `Dockerfile` must start with the `FROM` directive. For example, we may have an image built from the following one-liner: **docker commit $(**
**docker start $(**
**docker create alpine /bin/sh -c **
**"echo My custom build > /etc/motd" **
))

This roughly equates to the following `Dockerfile`: **FROM alpine**
RUN echo "My custom build" > /etc/motd

Obviously, building with a `Dockerfile` is much more concise and precise.

The `docker build [OPTIONS] [CONTEXT]` command is the only command associated with building tasks. A context can be a local path, URL, or `stdin`, which denotes the location of the `Dockerfile`. Once a build is triggered, the `Dockerfile`, alongside everything under the context, will be sent to the Docker daemon beforehand and then the daemon will start to execute instructions in the `Dockerfile` sequentially. Every execution of the instructions results in a new cache layer, and the ensuing instruction is executed at the new cache layer in the cascade. Since the context will be sent somewhere that isn't guaranteed to be a local path, and sending too many irrelevant files takes time, it's a good practice to put the `Dockerfile`, code, necessary files, and a `.dockerignore` file in an `empty` folder to make sure the resultant image contains only the desired files.

The `.dockerignore` file is a list indicating which files under the same directory can be ignored at build time. It typically looks as follows: **$ cat .dockerignore**
 # ignore .dockerignore, .git
.dockerignore
.git
 # exclude all *.tmp files and vim swp file recursively
****/*.tmp**
****/[._]*.s[a-w][a-z]**

 # exclude all markdown files except README*.md
!README*.md

Generally, `docker build` will try to locate a file named `Dockerfile` under the context to start a build. Sometimes, however, we may want to give it another name, which we can do using the `-f` (`--file`) flag. Another useful flag, `-t` (`--tag`), is able to give an image one or more repository tags after an image is built. Let's say we want to build a `Dockerfile` named `builder.dck` under `./deploy` and label it with the current date and the latest tag. The command to do this is as follows: **## We can assign the image with more than one tag within a build command**
**$ docker build -f deploy/builder.dck **
**-t my-reg.com/prod/teabreaker:$(date +"%g%m%d") **
-t my-reg.com/prod/teabreaker:latest .

The syntax of a Dockerfile

The building blocks of a `Dockerfile` are a dozen directives. Most of these are made up of functions of the `docker run/create` flags. Let's take a look at the most essential ones:

- `FROM <IMAGE>[:TAG|[@DIGEST]`: This is to tell the Docker daemon which image the current `Dockerfile` is based on. It's also the one and only instruction that has to be in a `Dockerfile`; you can have a `Dockerfile` that contains only this line. Like all of the other image-relevant commands, the tag defaults to the latest if unspecified.
- `RUN`:

```
RUN <commands>
RUN ["executable", "params", "more params"]
```

  The `RUN` instruction runs one line of a command at the current cache layer and commits the outcome. The main discrepancy between the two forms is with regards to how the command is executed. The first form is called **shell form**. This actually executes commands in the form of `/bin/sh -c <commands>`. The other form is **exec form**. This treats the command with `exec` directly.

Using the shell form is similar to writing shell scripts, hence concatenating multiple commands by shell operators and line continuation, condition tests, or variable substitutions is completely valid. Bear in mind, however, that commands aren't processed by `bash` but by `sh`.

The `exec` form is parsed as a JSON array, which means that you have to wrap texts with double quotes and escape the reserved characters. Besides, as the command is not processed by any shell, the shell variables in the array will not be evaluated. On the other hand, if the shell doesn't exist in the base image, you can still use the `exec` form to invoke executables.

- `CMD`:

```
CMD ["executable", "params", "more params"]
CMD ["param1","param2"]
```

```
CMD command param1 param2 ...
```

The `CMD` it to set default commands for the built image, but it doesn't run the command at build time. If arguments are supplied upon executing `docker run`, the `CMD` configurations here are overridden. The syntax rules of `CMD` are almost identical to `RUN`; the previous two forms are the `exec` form, and the third one is the shell form, which prepends `/bin/sh -c` to the parameters as well. There's another `ENTRYPOINT` directive that would interact with `CMD`;the parameter of `ENTRYPOINT` would prepend to the three forms of `CMD` when a container starts. There can be many `CMD` directives in a `Dockerfile`, but only the last one will take effect.

- `ENTRYPOINT`:

```
ENTRYPOINT ["executable", "param1", "param2"]
ENTRYPOINT command param1 param2
```

These two forms are, respectively, the `exec` form and the shell form, and the syntax rules are the same as `RUN`. The entry point is the default executable for an image. This means that when a container spins up, it runs the executable configured by `ENTRYPOINT`. When `ENTRYPOINT` is combined with the `CMD` and `docker run` arguments, writing it in a different form would lead to very different behavior. Here are the rules regarding their combinations:

- If the `ENTRYPOINT` is in shell form, then the `CMD` and `docker run` arguments would be ignored. The runtime command would be as follows:

```
/bin/sh -c entry_cmd entry_params ...
```

- If the `ENTRYPOINT` is in `exec` form and the `docker run` arguments are specified, then the `CMD` commands are overridden. The runtime command would be as follows:

```
entry_cmd entry_params run_arguments
```

- If the `ENTRYPOINT` is in `exec` form and only `CMD` is configured, the runtime command would become the following for the three forms:

```
entry_cmd entry_parms CMD_exec CMD_parms
entry_cmd entry_parms CMD_parms
entry_cmd entry_parms /bin/sh -c CMD_cmd CMD_parms
```

- `ENV`:

```
ENV key value
ENV key1=value1 key2=value2 ...
```

The `ENV` instruction sets environment variables for the consequent instructions and the built image. The first form sets the key to the string after the first space, including special characters, except the line continuation character. The second form allows us to set multiple variables in a line, separated with spaces. If there are spaces in a value, either enclose them with double quotes or escape the space character. Moreover, the key defined with `ENV` also takes effect on variables in the same document. See the following examples to observe the behavior of `ENV`:

```
FROM alpine
# first form
ENV k1 wD # aw
# second form, line continuation character also works
ENV k2=v2 k3=v\ 3 \
k4="v 4"
# ${k2} would be evaluated, so the key is "k_v2" in this case
ENV k_${k2}=$k3 k5=\"K\=da\"
# show the variables
RUN env | grep -Ev '(HOSTNAME|PATH|PWD|HOME|SHLVL)' | sort
```

The output during the `docker build` would be as follows: **...**

```
---> Running in c5407972c5f5
k1=wD # aw
k2=v2
k3=v 3
k4=v 4
k5="K=da"
k_v2=v 3
...
```

- `ARG key[=<default value>]`: The `ARG` instruction can pass our arguments as environment variables into the building container via the `--build-arg` flag of `docker build`. For instance, building the following file using `docker build --build-arg FLAGS=--static` would result in `RUN ./build/dev/run --static` on the last line:

```
FROM alpine
ARG TARGET=dev
ARG FLAGS
```

```
RUN ./build/$TARGET/run $FLAGS
```

Unlike `ENV`, only one argument can be assigned per line. If we are using `ARG` together with `ENV`, then the value of `ARG`, no matter where it is (either by `--build-arg` or the default value), would be overwritten by the value of `ENV`. Due to the frequent use of the proxy environment variables, these are all supported as arguments by default, including `HTTP_PROXY`, `http_proxy`, `HTTPS_PROXY`, `https_proxy`, `FTP_PROXY`, `ftp_proxy`, `NO_PROXY`, and `no_proxy`. This means we can pass these building arguments without defining them in the `Dockerfile` beforehand. One thing worth noting is that the value of `ARG` would remain in both the shell history on the building machine and the Docker history of the image, which means it's wise not to pass sensitive data via `ARG`:

- `LABEL key1=value1 key2=value2 ...`: The use of `LABEL` resembles that of `ENV`, but a label is only stored in the metadata section of an image and is used by other host programs instead of programs in a container. For example, if we attach the maintainer of our image in the form `LABEL maintainer=johndoe@example.com`, we can filter the annotated image with the `-f(--filter)` flag in this query: `docker images --filter label=maintainer=johndoe@example.com`.
- `EXPOSE <port> [<port> ...]`: This instruction is identical to the `--expose` flag used with `docker run/create`, exposing ports in the container created by the resulting image.
- `USER <name|uid>[:<group|gid>]`: The `USER` instruction switches the user to run the subsequent instructions, including the ones in `CMD` or `ENTRYPOINT`. However, it can't work properly if the user doesn't exist in the image. If you want to run instructions using a user that doesn't exist, you have to run `adduser` before using the `USER` directive.

- `WORKDIR <path>`: This instruction sets the working directory to a certain path. Environment variables set with `ENV` take effect on the path. The path would be created automatically if it doesn't already exist. It works like `cd` in a `Dockerfile`, as it takes both relative and absolute paths and can be used multiple times. If an absolute path is followed by a relative path, the result would be relative to the previous path:

```
WORKDIR /usr
WORKDIR src
WORKDIR app
RUN pwd
# run docker build
...
---> Running in 73aff3ae46ac
```

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

- `COPY`:

```
        COPY [--chown=<user>:<group>] <src> ... <dest>
        COPY [--chown=<user>:<group>] ["<src>", ..., "<dest>"]
```

This directive copies the source to a file or a directory in the building container. The source as well as the destination could be files or directories. The source must be within the context path and not excluded by `.dockerignore`, as only those will be sent to the Docker daemon. The second form is for cases in which the path contains spaces. The `--chown` flag enables us to set the file owner on the fly without running additional `chown` steps inside containers. It also accepts numeric user IDs and group IDs:

- `ADD`:

```
            ADD [--chown=<user>:<group>] <src > ... <dest>
            ADD [--chown=<user>:<group>] ["<src>", ..., "<dest>"]
```

`ADD` is quite similar to `COPY` in terms of its functionality: it moves files into an image. The major differences are that `ADD` supports downloading files from a remote address and extracting compressed files from the container in one line. As such, `<src>` can also be a URL or compressed file. If `<src>` is a URL, `ADD` will download it and copy it into the image; if `<src>` is inferred as a compressed file, it'll be extracted into the `<dest>` path:

- `VOLUME`:

```
        VOLUME mount_point_1 mount_point_2 ...
        VOLUME ["mount point 1", "mount point 2", ...]
```

The `VOLUME` instruction creates data volumes at the given mount points. Once it's been declared during build time, any change in the data volume at consequent directives would not persist. Besides, mounting host directories in a `Dockerfile` or `docker build` isn't doable because of portability concerns: there's no guarantee that the specified path would exist in the host. The effect of both syntax forms is identical; they only differ with regard to syntax parsing. The second form is a JSON array, so characters such as `\` should be escaped.

- `ONBUILD [Other directives]`: `ONBUILD` allows you to postpone some instructions to later builds that happen in the derived image. For example, suppose we

have the following two Dockerfiles:

```
--- baseimg.dck ---
FROM alpine
RUN apk add --no-cache git make
WORKDIR /usr/src/app
ONBUILD COPY . /usr/src/app/
ONBUILD RUN git submodule init \
       && git submodule update \
       && make
--- appimg.dck ---
FROM baseimg
EXPOSE 80
CMD ["/usr/src/app/entry"]
```

The instruction then would be evaluated in the following order when running `docker build`:

```
$ docker build -t baseimg -f baseimg.dck .
---
FROM alpine
RUN apk add --no-cache git make
WORKDIR /usr/src/app
---
$ docker build -t appimg -f appimg.dck .
---
COPY . /usr/src/app/
RUN git submodule init   \
 && git submodule update \
 && make
EXPOSE 80
CMD ["/usr/src/app/entry"]
```

Organizing a Dockerfile

Even though writing a `Dockerfile` is pretty much the same as composing a build script, there are some more factors that we should consider to build efficient, secure, and stable images. Moreover, a `Dockerfile` itself is also a document. Keeping it readable makes it easier to manage.

Let's say we have an application stack that consists of application code, a database, and a cache. The initial `Dockerfile` of our stack could be the following:

```
FROM ubuntu
ADD . /proj
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y redis-server python python-pip mysql-server
ADD /proj/db/my.cnf /etc/mysql/my.cnf
ADD /proj/db/redis.conf /etc/redis/redis.conf
ADD https://example.com/otherteam/dep.tgz /tmp/
RUN -zxf /tmp/dep.tgz -C /usr/src
RUN pip install -r /proj/app/requirements.txt
RUN cd /proj/app ; python setup.py
CMD /proj/start-all-service.sh
```

The first suggestion is to make sure a container is dedicated to one thing and one thing only. This gives our system better transparency since it helps us clarify the boundaries between components in the system. Also, packing unnecessary packages is discouraged, as it increases the image size, which could slow down the time it takes to build, distribute, and launch the image. We'll remove the installation and configuration of both `mysql` and `redis` in our `Dockerfile` in the beginning. Next, the code is moved into the container with `ADD .`, which means that we're very likely to move the whole code repository into the container. Usually, there're lots of files that aren't directly relevant to the application, including VCS files, CI server configurations, or even build caches, and we probably wouldn't like to pack them into an image. For this reason, it is suggested to use `.dockerignore` to filter out these files as well. Lastly, using `COPY` is preferred over `ADD` in general, unless we want to extract a file in one step. This is

because it is easier to predict the outcome when we use `COPY`. Now our `Dockerfile` is simpler, as shown in the following code snippet: **FROM ubuntu**
ADD proj/app /app
RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y python python-pip
ADD https://example.com/otherteam/dep.tgz /tmp/
RUN tar -zxf /tmp/dep.tgz -C /usr/src
RUN pip install -r /app/requirements.txt
RUN cd /app ; python setup.py
CMD python app.py

While building an image, the Docker engine will try to reuse the cache layer as much as possible, which notably reduces the build time. In our `Dockerfile`, we have to go through all the updating and dependency installation processes if any package to be installed needs updating. To benefit from building caches, we'll re-order the directives based on a rule of thumb: run less frequent instructions first.

Additionally, as we've described before, any changes made to the container filesystem result in a new image layer. To be more specific, `ADD`, `RUN`, and `COPY` create layers. Even though we deleted certain files in the consequent layer, these files still occupy image layers as they're still being kept at intermediate layers. Therefore, our next step is to minimize the image layers by simply compacting multiple `RUN` instructions and cleaning the unused files at the end of the `RUN`. Moreover, to keep the readability of the `Dockerfile`, we tend to format the compacted `RUN` with the line continuation character, `\`. Although `ADD` can fetch a file from a remote location to the image, it's still not a good idea to do this as this would still occupy a layer in order to store the downloaded file. Downloading files with `RUN` and `wget/curl` is more common.

In addition to working with the building mechanisms of Docker, we'd also like to write a maintainable `Dockerfile` to make it clearer, more predictable, and more stable. Here are some suggestions:

- Use `WORKDIR` instead of the inline `cd`, and use the absolute path for `WORKDIR`
- Explicitly expose the required ports
- Specify a tag for the base image
- Separate and sort packages line by line

- Use the `exec` form to launch an application

The first four suggestions are pretty straightforward, aimed at eliminating ambiguity. The last one refers to how an application is terminated. When a stop request from the Docker daemon is sent to a running container, the main process (`PID 1`) will receive a stop signal (`SIGTERM`). If the process is not stopped after a certain period of time, the Docker daemon will send another signal (`SIGKILL`) to kill the container. The `exec` form and shell form differ here. In the shell form, the `PID 1` process is `/bin/sh -c`, not the application. Furthermore, different shells don't handle signals in the same way. Some forward the stop signal to the child processes, while some do not. The shell at Alpine Linux doesn't forward them. As a result, to stop and clean up our application properly, using the `exec` form is encouraged.

Combining those principles, we have the following `Dockerfile`: **FROM ubuntu:18.04**

**RUN apt-get update && apt-get upgrade -y **
**&& apt-get install -y --no-install-recommends **
**curl **
**python3.6 **
**python-pip=9.* **
**&& curl -SL https://example.com/otherteam/dep.tgz **
**| tar -zxC /usr/src **
&& rm -rf /var/lib/apt/lists/*

ENTRYPOINT ["python"]
CMD ["entry.py"]
EXPOSE 5000
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . /app

There are other practices that we can follow to make our `Dockerfile` better, including starting from a dedicated and smaller base image rather than general-purpose distributions, using users other than `root` for better security, and removing unnecessary files in the `RUN` in which they are joined.

Multi-stage builds

The principles we've discussed so far are all about how to make builds fast and how to make the final image smaller while keeping the maintainability of the `Dockerfile`. Instead of striving to optimize a `Dockerfile`, writing one to build the artifacts we need and then moving them to another image with runtime dependencies only makes it much easier to sort the different logic out. In the building stage, we can forget about minimizing the layers so that the build cache can be reused efficiently; when it comes to the release image, we can follow the previously recommended techniques to make our image clean and small. Before Docker CE 17.05, we had to write two Dockerfiles to implement this build pattern. Now, Docker has built-in support to define different stages in a single `Dockerfile`.

Take a `golang` build as an example: this requires lots of dependencies and a compiler, but the artifact can merely be a single binary. Let's look at the following example: **FROM golang:1.11 AS builder**
ARG GOOS=linux
ARG GOARCH=amd64
ARG CGO_ENABLED=0
WORKDIR /go/src/app
COPY main.go .
RUN go get .
RUN go build -a -tags netgo -ldflags '-w -s -extldflags "-static"'

FROM scratch
COPY --from=builder /go/src/app/app .
ENTRYPOINT ["/app"]
CMD ["--help"]

The delimiter for the different stages is the `FROM` directive, and we can name the stages with the `AS` keyword after the image name. At the `builder` stage, Docker starts a `golang` base image, and then builds the target binary as usual. Afterwards, during the second build, it copies the binary from the `builder` container with `--from=[stage name|image name]` to a `scratch` image—a reserved name for an entirely

empty image. As there's only one binary file and one layer in the resultant image, its size is dramatically smaller than the `builder` one.

The number of stages isn't limited to two, and the source of the `COPY` directive can either be a previously defined stage or a built image. The `ARG` directive works against `FROM`, which is also the only exception that can be written before a `FROM` directive, as they belong to different stages. In order to use it, the `ARG` directive to be consumed in `FROM` should be declared before `FROM`, as shown here: **ARG TAGS=latest**
FROM ubuntu:$TAGS
...

Multi-container orchestration

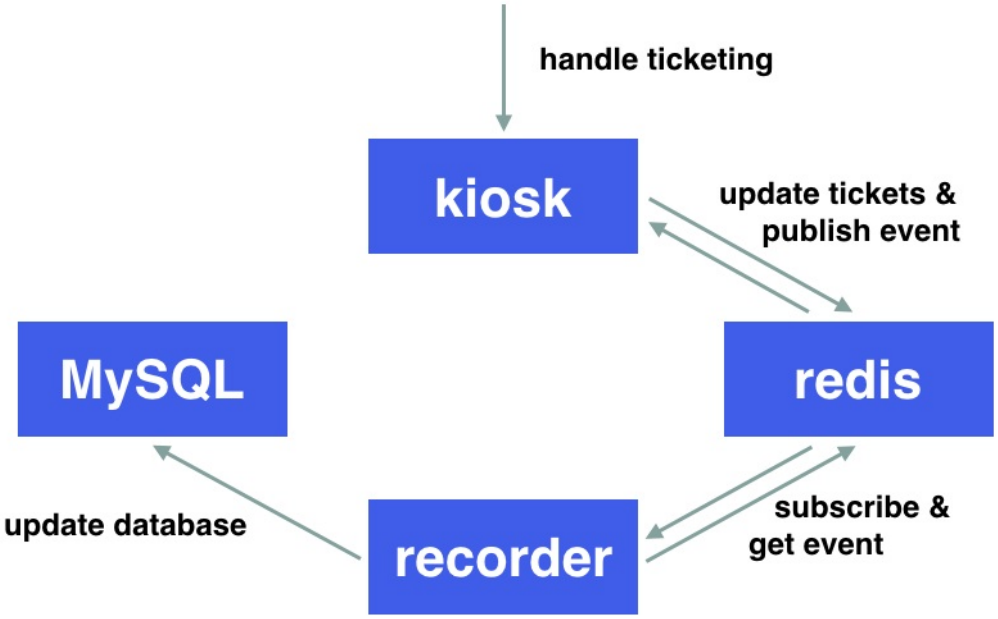
As we pack more and more applications into isolated boxes, we'll soon realize that we need a tool that is able to help us tackle many containers simultaneously. In this section, we'll move a step up from spinning up just one single container to orchestrating multiple containers in a band.

Piling up containers

Modern systems are usually built as stacks made up of multiple components that are distributed over networks, such as application servers, caches, databases, and message queues. Each individual component is also a self-contained system with many sub-components. What's more, the rising trend of microservices introduces additional degrees of complexity into these entangled relationships between systems. Because of this, even though container technology gives us a certain degree of relief regarding deployment tasks, coordinating components in a system is still difficult.

Let's say we have a simple application called `kiosk`, which connects to a `redis` to manage how many tickets we currently have. Once a ticket is sold, it publishes an event through a `redis` channel. The **recorder** subscribes the `redis` channel and writes a timestamp log into a MySQL database upon receiving any event.

For the `kiosk` and the `recorder`, you can find the code as well as their Dockerfiles here: https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter2. The architecture is as follows:



We know how to start those containers separately and connect them with each other. Based on what we've discussed before, we would first create a bridge network and run the containers inside:

```
$ docker network create kiosk
$ docker run -d --network-alias lcredis --network=kiosk redis
$ docker run -d --network-alias lmysql -e MYSQL_ROOT_PASSWORD=$MYPS \
                --network=kiosk mysql:5.7
$ docker run -d -p 5000:5000 \
                -e REDIS_HOST=lcredis --network=kiosk kiosk-example
$ docker run -d -e REDIS_HOST=lcredis -e MYSQL_HOST=lmysql      \
                -e MYSQL_ROOT_PASSWORD=$MYPS -e MYSQL_USER=root \
                --network=kiosk recorder-example
```

Because our `kiosk` and `recorder` are much lighter than the database, our applications are very likely to start up earlier than the database's. In this case, our `kiosk` might fail if any incoming connection requests changes to the databases or to Redis. In other words, we have to consider the startup order in our startup scripts. We also have to deal with problems such as how to deal with random components crashing, how to manage variables, how to scale out certain components, and how to manage the states of every moving part.

An overview of Docker compose

Docker compose is a tool that enables us to run multiple containers with ease. It's a built-in tool in the Docker CE distribution. All it does is read `docker-compose.yml` (or `.yaml`) to run the defined containers. A `docker-compose` file is a YAML-based template, and it typically looks like this: version: '3'
services:
hello-world:
image: hello-world

Launching it is pretty simple: save the template to `docker-compose.yml` and use the `docker-compose up` command to start it: **$ docker-compose up**
Creating network "user_default" with the default driver
Pulling hello-world (hello-world:)...
...
Creating user_hello-world_1 ... done
Attaching to user_hello-world_1
hello-world_1 |
hello-world_1 | Hello from Docker!
hello-world_1 | This message shows that your installation appears to be working correctly.
...
user_hello-world_1 exited with code 0

Let's take a look at what `docker-compose` did when the `up` command was executed.

Docker compose is basically a medley of Docker functions for multiple containers. For example, the counterpart of `docker build` is `docker-compose build`; the former builds a Docker image and the latter builds Docker images listed in `docker-compose.yml`. Remember, however, that the `docker-compose run` command doesn't correspond to `docker run`; it's actually used to run a specific container from the configuration in `docker-compose.yml`. In fact, the closest command to `docker run` is `docker-compose up`.

The `docker-compose.yml` file consists of different configurations of volumes,

networks, and services. There should be a version definition to indicate which version of the `docker-compose` format should be used. With this understanding of the template structure, what the previous `hello-world` example does is quite clear; it creates a service called `hello-world` that uses the `hello-world:latest` image.

Since there's no network defined, `docker-compose` will create a new network with a default driver and connect services to that network, as shown at the start of the output of the example.

The network name of a container will be the name of the service. You may notice that the name displayed in the console differs slightly from its original one in `docker-compose.yml`. This is because Docker compose tries to avoid name conflicts between containers. As a result, Docker compose runs the container with the name it generated and makes a network alias with the service name. In this example, both `hello-world` and `user_hello-world_1` are resolvable to other containers within the same network.

*Docker compose is the easiest option to run multiple containers on a single machine, but it's not designed to orchestrate containers across networks. Other major container orchestration engines such as **Kubernetes**, **Docker Swarm**, **Mesos** (with **Marathon** or **Aurora**), or **Nomad** are better choices to run containers across multiple nodes.*

Composing containers

As Docker compose is the same as Docker in many aspects, it's more efficient to understand how to write `docker-compose.yml` with examples than start from `docker-compose` syntax. Let's now go back to the `kiosk-example` we looked at earlier and start with a `version` definition and four `services`: **version: '3'**
services:
kiosk-example:
recorder-example:
lcredis:
lmysql:

The `docker run` arguments for `kiosk-example` are pretty simple. They include a publishing port and an environment variable. On the Docker compose side, we fill the source image, the publishing port, and environment variables accordingly. Because Docker compose is able to handle `docker build`, it can build images if those images can't be found locally. We want to use this to decrease the effort of image management: **kiosk-example:**
image: kiosk-example
build: ./kiosk
ports:
- "5000:5000"
environment:
REDIS_HOST: lcredis

Converting the Docker run of the `recorder-example` and `redis` in the same manner, we have a template that looks as follows: **version: '3'**
services:
kiosk-example:
image: kiosk-example
build: ./kiosk
ports:
- "5000:5000"
environment:
REDIS_HOST: lcredis

recorder-example:
image: recorder-example
build: ./recorder
environment:
REDIS_HOST: lcredis
MYSQL_HOST: lmysql
MYSQL_USER: root
MYSQL_ROOT_PASSWORD: mysqlpass
lcredis:
image: redis
ports:
- "6379"

For the MySQL part, MySQL requires a data volume to keep its data as well as its configurations. In addition to the `lmysql` section, we add `volumes` at the level of `services` and an empty map called `mysql-vol` to claim a data volume: **lmysql:**
image: mysql:5.7
environment:
MYSQL_ROOT_PASSWORD: mysqlpass
MYSQL_DATABASE: db
MYSQL_USER: user
MYSQL_PASSWORD: pass
volumes:
- mysql-vol:/var/lib/mysql
ports:
- "3306"
volumes:
mysql-vol: {}

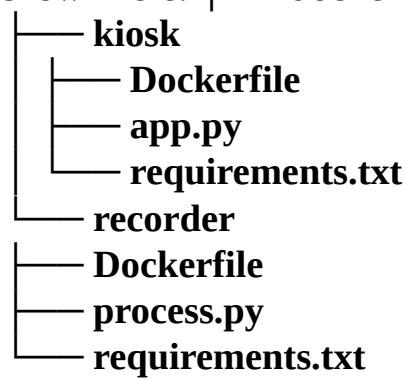
One of the benefits of this is that we can manage the launching order between the components with `depends_on`. What this does is maintain the order; it can't detect whether the components that it will use are ready. This means our application could still connect and write to the database before the database is ready. All in all, as our program is a part of a distributed system with many moving parts, it's a good idea to make it resilient to the changes of its dependencies.

Combining all of the preceding configurations, including `depends_on`, we have the

final template, as follows: **version: '3'**
services:
kiosk-example:
image: kiosk-example
build: ./kiosk
ports:
- "5000:5000"
environment:
REDIS_HOST: lcredis
depends_on:
- lcredis
recorder-example:
image: recorder-example
build: ./recorder
environment:
REDIS_HOST: lcredis
MYSQL_HOST: lmysql
MYSQL_USER: root
MYSQL_ROOT_PASSWORD: mysqlpass
depends_on:
- lmysql
- lcredis
lcredis:
image: redis
ports:
- "6379"
lmysql:
image: mysql:5.7
environment:
MYSQL_ROOT_PASSWORD: mysqlpass
MYSQL_DATABASE: db
MYSQL_USER: user
MYSQL_PASSWORD: pass
volumes:
- mysql-vol:/var/lib/mysql
ports:
- "3306"

volumes:
mysql-vol: {}

This file is put in the `root` folder of a project. The corresponding file tree is shown here:



Finally, run `docker-compose up` to check everything is fine. We can check every component is linked nicely using `kiosk`:

```
$ curl localhost:5000
Gooday!
 Import tickets with "curl -XPOST -F 'value=<int>' /tickets"
 Purchase a ticket with "curl -XPOST /buy
 Get current tickets with "curl -XGET /tickets"
$ curl -XGET localhost:5000/tickets
0
$ curl -XPOST -F 'value=10' localhost:5000/tickets
SUCCESS
$ curl -XGET localhost:5000/tickets
10
$ curl -XPOST localhost:5000/buy
SUCCESS
$ docker exec chapter2_lmysql_1  mysql -u root -pmysqlpass \
              -e "select * from kiosk.sellinglog;"
id  ts
1   1536704902204
```

Writing a template for Docker compose is as simple as this. We're now able to run an application in the stack with ease.

Summary

Starting from the very primitive elements of Linux containers and Docker tool stacks, we went through every aspect of containerizing an application, including packing and running a Docker container, writing a `Dockerfile` for code-based immutable deployment, and manipulating multiple containers with Docker compose. However, the abilities we gained in this chapter only allow us to run and connect containers within the same host, which limits our ability to build larger applications.

In `Chapter 3`, *Getting Started with Kubernetes*, we'll meet Kubernetes, unleashing the power of containers beyond the limits of scale.

Getting Started with Kubernetes

So far, we've learned about the benefits that containers can bring us, but what if we need to scale out our services to meet the needs of our business? Is there a way to build services across multiple machines without dealing with cumbersome network and storage settings? Is there an easy way to manage and roll out our microservices with a different service cycle? This is where Kubernetes comes into play. In this chapter, we'll look at the following concepts:

- Understanding Kubernetes
- Components of Kubernetes
- Kubernetes resources and their configuration files
- How to launch the kiosk application using Kubernetes

Understanding Kubernetes

Kubernetes is a platform for managing containers across multiple hosts. It provides lots of management features for container-oriented applications, such as auto-scaling, rolling deployments, compute resources, and storage management. Like containers, it's designed to run anywhere, including on bare metal, in our data center, in the public cloud, or even in the hybrid cloud.

Kubernetes fulfills most application container operational needs. Its highlights include the following:

- Container deployment
- Persistent storage
- Container health monitoring
- Compute resource management
- Auto-scaling
- High availability by cluster federation

With Kubernetes, we can manage containerized applications easily. For example, by creating `Deployment`, we can roll out, roll over, or roll back selected containers (`Chapter 9`, *Continuous Delivery*) with just a single command. Containers are considered ephemeral. If we only have one host, we could mount host volumes into containers to preserve data. In the cluster world, however, a container might be scheduled to run on any host in the cluster. How do we mount a volume without specifying which host it's run on? Kubernetes **volumes** and **persistent volumes** were introduced to solve this problem (`Chapter 4`, *Managing Stateful Workloads*).

The lifetime of containers might be short; they may be killed or stopped anytime when they exceed the resource limit. How do we ensure our services are always on and are served by a certain number of containers? Deployment in Kubernetes ensures that a certain number of groups of containers are up and running. Kubernetes also supports **liveness probes** to help you define and monitor your application's health. For better resource management, we can define the

maximum capacity for Kubernetes nodes and the resource limit for each group of containers (also known as **pods**). The Kubernetes scheduler will select a node that fulfills the resource criteria to run the containers. We'll learn about this further in Chapter 8, *Resource Management and Scaling*. Kubernetes also provides an optional horizontal pod auto-scaling feature, which we can use to scale a pod horizontally by core or custom metrics. Kubernetes is also designed to have **high availability** (**HA**). We're able to create multiple master nodes and prevent single points of failure.

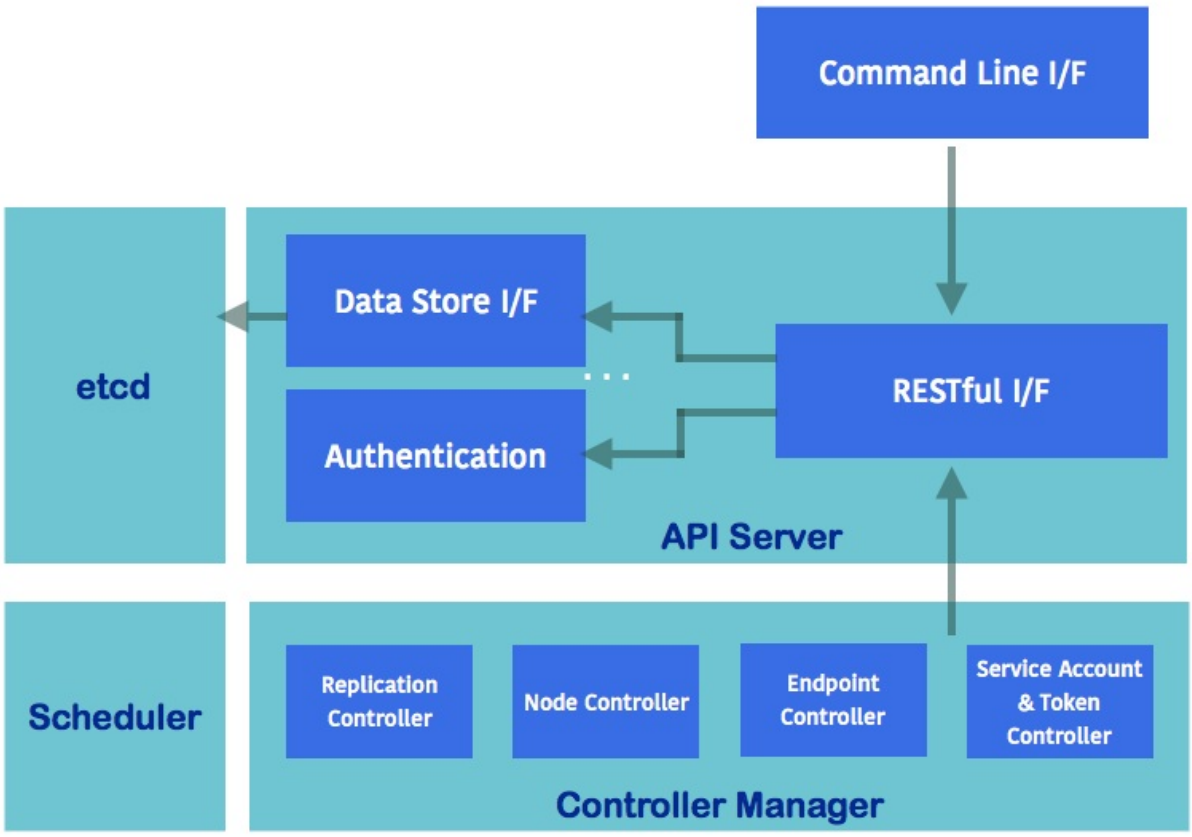
Kubernetes components

Kubernetes includes two major players:

- **Masters**: The master is the heart of Kubernetes; it controls and schedules all of the activities in the cluster
- **Nodes**: Nodes are the workers that run our containers

Master components

The master includes the **API Server**, the **Controller Manager**, the **Scheduler**, and **etcd**. All components can run on different hosts with clustering. However, in this case, we'll make all of the components run on the same node as shown in the following diagram:



Master components

API server (kube-apiserver)

The API server provides an HTTP/HTTPS server, which provides a RESTful API for the components in the Kubernetes master. For example, we could use `GET` to get the resource status or `POST` to create a new resource. We can also watch for updates for resources. The API server stores the object information into etcd, which is Kubernetes' backend data store.

Controller manager (kube-controller-manager)

The controller manager is a set of control loops that watch the changes in the API server and ensure the cluster is in the desired state. For example, the deployment controller ensures that the whole deployment is run on the desired amount of containers. The node controller responds and evicts the pod when the nodes go down. The endpoint controller is used to create a relationship between services and pods. The service account and the token controller are used to create a default account and API access tokens.

To accommodate different development paces and release cycles from different cloud providers, from Kubernetes version 1.6, cloud provider-specific logic was moved from `kube-controller-manager` *to the cloud controller manager (*`cloud-controller-manager`*). This was promoted to beta in version 1.11.*

etcd

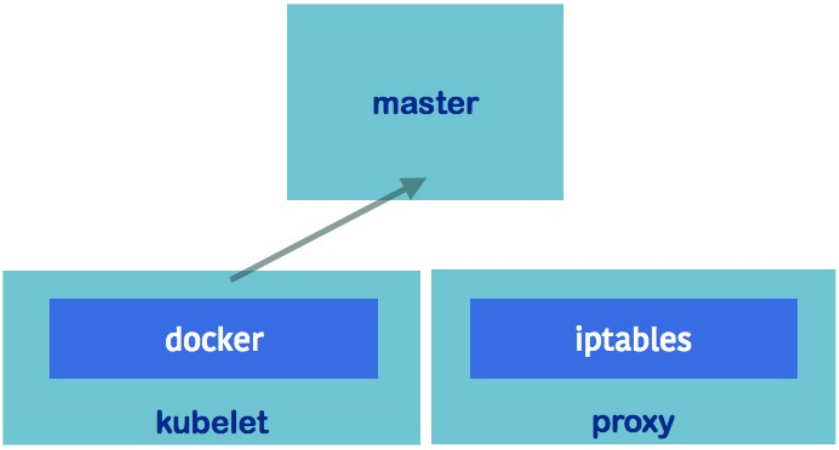
etcd is an open source distributed key-value store (https://coreos.com/etcd). Kubernetes stores all of the RESTful API objects here. etcd is responsible for storing and replicating data.

Scheduler (kube-scheduler)

The scheduler determines which nodes are suitable candidates for pods to run on. It uses not only resource capacity and the balance of the resource utilization on the node but also node affinity, taints, and toleration. For more information, refer to Chapter 8, *Resource Management and Scaling*.

Node components

Node components are provisioned and run on every node, which report the runtime status of the pod to the **master**:



Node components

Kubelet

Kubelet is a major process in the nodes. It reports node activities back to `kube-apiserver` periodically, including pod health, node health, and liveness probe. As the preceding diagram shows, it runs containers via container runtimes, such as Docker or rkt.

Proxy (kube-proxy)

The proxy handles the routing between a pod load balancer (also known as a **service**) and pods. It also provides routing from external internet to services. There are three proxy modes: `userspace`, `iptables`, and `ipvs`. The `userspace` mode creates a large overhead by switching the kernel space and user space. The `iptables` mode, on the other hand, is the latest default proxy mode. It changes the `iptables` Network Address Translation (NAT: https://en.wikipedia.org/wiki/Network_address_translation) in Linux to achieve routing TCP and UDP packets across all containers. **IP Virtual Servers** (**IPVS**) was **general available** (**GA**) in Kubernetes 1.11 and is used to address performance degradations when running 1,000+ services in a cluster. It runs on a host and acts as a load balancer, forwarding the connection to real servers. IPVS mode will fall back to `iptables` in some scenarios; please refer to https://github.com/kubernetes/kubernetes/tree/master/pkg/proxy/ipvs for more detailed information.

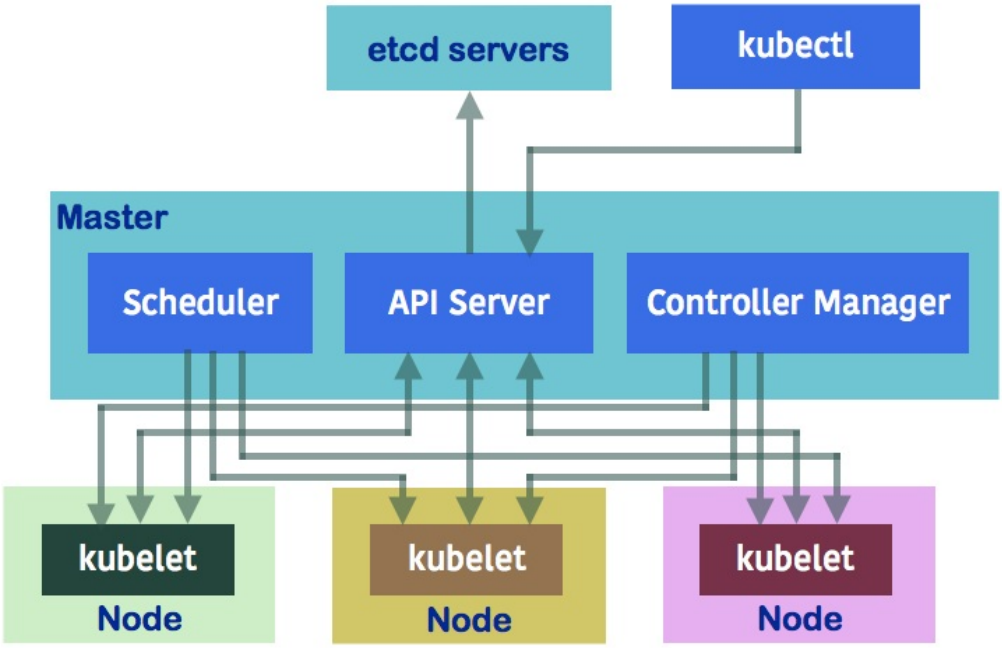
Docker

As described in Chapter 2, *DevOps with Containers*, Docker is a container runtime implementation. Kubernetes uses Docker as a default container engine, and it also supports other container runtimes, such as rkt (https://coreos.com/rkt/) and runc (https://github.com/opencontainers/runc).

The interaction between the Kubernetes master and nodes

As we can see in the following diagram, the client uses **kubectl**, which is a command-line interface, to send requests to the **API Server**. The **API Server**, a hub between the master components, will respond to the client requests and push and pull the object information from etcd. If a new task is created, such as run pods, the scheduler will determine which node should be assigned to do that task. The **Controller Manager** monitors the running tasks and responds if any undesired state occurs.

The **API Server** fetches the logs from pods via the **kubelet**:



Interaction between master and nodes

Getting started with Kubernetes

In this section, we'll learn how to set up a single-node cluster. Then, we'll learn how to interact with Kubernetes via its command-line tool: kubectl. We'll go through all of the important Kubernetes API objects and their expressions in YAML format, which is the input to kubectl. We'll then see how kubectl sends requests to the API server to create the desired objects accordingly.

Preparing the environment

Firstly, kubectl has to be installed. In major Linux distributions (such as Ubuntu or CentOS), you can just search for and install the package named `kubectl` via the package manager. In macOS, we can choose to use Homebrew (`https://brew.sh/`) to install it. Homebrew is a useful package manager in macOS. We can easily install Homebrew via the `/usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"` command. Then, we can run brew install kubernetes-cli to install kubectl via Homebrew.

Let's now start to provision a Kubernetes cluster. The easiest way to do this is to run minikube (`https://github.com/kubernetes/minikube`), which is a tool to run Kubernetes on a single-node locally. It can be run on Windows, Linux, and macOS. In the following example, we'll run on macOS. Minikube will launch a VM with Kubernetes installed. We'll then be able to interact with it via kubectl.

Note that minikube isn't suitable for production or any heavy-load environment. It has some limitations due to its single node nature. We'll learn how to run a real cluster in `Chapter 10`, *Kubernetes on AWS*, `Chapter 11`, *Kubernetes on GCP*, and `Chapter 12`, *Kubernetes on Azure*, instead.

Before installing minikube, we have to install some dependencies first. Minikube's official GitHub repository (`https://github.com/kubernetes/minikube`) lists the dependencies and drivers for the different platforms. In our case, we're using VirtualBox (`https://www.virtualbox.org/`) as the driver in macOS. You're free to use other drivers; visit the preceding minikube GitHub link to find more options.

After downloading and installing VirtualBox from its official website, we're ready to go. We can install minikube via `brew cask install minikube`: **// install minikube**
 # brew cask install minikube
==> Tapping caskroom/cask
==> Linking Binary 'minikube-darwin-amd64' to '/usr/local/bin/minikube'.
...
minikube was successfully installed!

After minikube is installed, we can start the cluster via the `minikube start` command. This will launch a Kubernetes cluster locally. At the time of writing, the default Kubernetes version that minikube v0.30.0 supports is `v.1.12.0`. You also can add the `--kubernetes-version` argument to specify the particular Kubernetes version you'd like to run. For example, assume we want to run a cluster with the version v1.12.1:

```
// start the cluster (via minikube v0.30.0)
#
Starting local Kubernetes v1.12.1 cluster...
Starting VM...
Getting VM IP address...
Moving files into cluster...
Downloading kubeadm v1.12.1
Downloading kubelet v1.12.1
Finished Downloading kubeadm v1.12.1
Finished Downloading kubelet v1.12.1
Setting up certs...
Connecting to cluster...
Setting up kubeconfig...
Starting cluster components...
Kubectl is now configured to use the cluster.
Loading cached images from config file.
```

This will then proceed to start a VM named `minikube` in VirtualBox and set up the cluster via `kubeadm`, a Kubernetes provisioning tool. It'll also set up `kubeconfig`, which is a configuration file to define the context and authentication settings of the cluster. With `kubeconfig`, we're able to switch to different clusters via the `kubectl` command. We could use the `kubectl config view` command to see the current settings in `kubeconfig`: **apiVersion: v1**
 # cluster and certificate information
clusters:
- cluster:
certificate-authority: /Users/chloelee/.minikube/ca.crt
server: https://192.168.99.100:8443
name: minikube
 # context is the combination of cluster, user and namespace
contexts:
- context:
cluster: minikube
user: minikube
name: minikube
current-context: minikube
kind: Config

preferences: {} # user information

users:
- name: minikube
user:
client-certificate: /Users/chloelee/.minikube/client.crt
client-key: /Users/chloelee/.minikube/client.key

Here, we're currently using the `minikube` context. The context is a combination of the authentication information and the cluster connection information. You could use `kubectl config` use-context `$context` to forcibly switch the context if you have more than one context.

kubectl

`kubectl` is the command-line tool to manage Kubernetes clusters. The most general usage of `kubectl` is to check the `version` of the cluster: **// check Kubernetes version**
 # kubectl version

Client Version: version.Info{Major:"1", Minor:"12", GitVersion:"v1.12.0", GitCommit:"0ed33881dc4355495f623c6f22e7dd0b7632b7c0", GitTreeState:"clean", BuildDate:"2018-10-01T00:59:42Z", GoVersion:"go1.11", Compiler:"gc", Platform:"darwin/amd64"}
Server Version: version.Info{Major:"1", Minor:"11", GitVersion:"v1.11.3", GitCommit:"a4529464e4629c21224b3d52edfe0ea91b072862", GitTreeState:"clean", BuildDate:"2018-09-09T17:53:03Z", GoVersion:"go1.10.3", Compiler:"gc", Platform:"linux/amd64"}

We then know our server version is upto date, which is the latest at the time of writing—version 1.12.0. The general syntax of `kubectl` is as follows: **kubectl [command] [type] [name] [flags]**

`command` indicates the operation you want to perform. If you type `kubectl help` in Terminal, it'll show the supported commands. `type` means the resource type. We'll learn about the major resource types in the next section. `name` is how we name our resources. It's always good practice to have clear and informative names throughout. For the `flags`, if you type `kubectl options`, the `stdout` will show all of the flags you could pass on.

We can always add `--help` to get more detailed information on specific commands, as in the example: **// show detailed info for logs command**
kubectl logs --help
Print the logs for a container in a pod or specified resource. If the pod has only one container, the container name is optional.

Aliases:
logs, log

Examples:
 # Return snapshot logs from pod nginx with only one container
kubectl logs nginx

 # Return snapshot logs for the pods defined by label app=nginx
kubectl logs -lapp=nginx
...
Options
...

We can then get examples and supported options in the `kubectl logs` command.

Kubernetes resources

Kubernetes objects are the entries in the cluster, which are stored in etcd. They represent the desired state of your cluster. When we create an object, we send the request to the API server by kubectl or a RESTful API. The API server will check whether the request is valid, store the state in etcd, and interact with other master components to ensure the object exists. Kubernetes uses namespaces to isolate the objects virtually, so we could create different namespaces for different teams, usages, projects, or environments. Every object has its own name and unique ID. Kubernetes also supports labels and annotations to let us tag our objects. Labels in particular can be used to group the objects together.

apiVersion: Kubernetes API version
kind: object type
metadata:
 spec metadata, i.e. namespace, name, labels and annotations
spec:
 the spec of Kubernetes object

Namespaces

Kubernetes namespaces allow us to implement isolation of multiple virtual clusters. Objects in different namespaces are invisible to each other. This is useful when different teams or projects share the same cluster. Most resources come under a namespace (these are known as namespaced resources); however, some generic resources, such as nodes or namespaces themselves, don't belong to any namespace. Kubernetes has three namespaces:

- `default`
- `kube-system`
- `kube-public`

If we don't explicitly assign a namespace to a namespaced resource, it'll be located in the namespace of the current context. If we never add a new namespace, a default namespace will be used.

Kube-system namespaces are used by objects created by the Kubernetes system, such as addon, which are the pods or services that implement cluster features, such as dashboard. Kube-public namespace was introduced in Kubernetes version 1.6, which is used by a beta controller manager (BootstrapSigner: `https://kubernetes.io/docs/admin/bootstrap-tokens`), putting the signed cluster location information into the `kube-public` namespace. This information could be viewed by authenticated or unauthenticated users.

In the following sections, all of the namespaced resources are located in a default namespace. Namespaces are also very important for resource management and roles. We'll provide further information in `Chapter 8`, *Resource Management and Scaling*.

Let's see how to create a namespace. A namespace is a Kubernetes object. We can specify the type to be a namespace, just like other objects. An example of how to create a namespace called `project1` follows: **// configuration file of namespace**
 # cat 3-2-1_ns1.yml
apiVersion: v1

kind: Namespace
metadata:
name: project1

// create namespace for project1
 # kubectl create -f 3-2-1_ns.yaml
namespace/project1 created

// list namespace, the abbreviation of namespaces is ns. We could use `kubectl get ns` to list it as well.
 # kubectl get namespaces
NAME STATUS AGE
default Active 1d
kube-public Active 1d
kube-system Active 1d
project1 Active 11s

Let's now try to start two nginx containers via deployment in the `project1` namespace: **// run a nginx deployment in project1 ns**
 # kubectl run nginx --image=nginx:1.12.0 --replicas=2 --port=80 --namespace=project1

deployment.apps/nginx created

When we list pods by `kubectl get pods`, we'll see nothing in our cluster. This is because Kubernetes uses the current context to decide which namespace is current. If we don't explicitly specify namespaces in the context or the `kubectl` command line, the `default` namespace will be used: **// We'll see the Pods if we explicitly specify --namespace # kubectl get pods --namespace=project1**
NAME READY STATUS RESTARTS AGE
nginx-8cdc99758-btgzj 1/1 Running 0 22s
nginx-8cdc99758-xpk58 1/1 Running 0 22s

You could use `--namespace <namespace_name>`, `--namespace=<namespace_name>`, `-n <namespace_name>`, or `-n=<namespace_name>` to specify the namespace for a command. To list the resources across namespaces, use the `--all-namespaces` parameter.

Another way to do this is to change the current context to point to the desired namespace rather than the `default` namespace.

Name

Every object in Kubernetes owns its own name that's uniquely identified within the same namespace. Kubernetes uses object names as part of a resource's URL for the API server, so it has be a combination of lowercase alphanumeric characters and dashes and dots, and it has to be less than 254 characters long. Besides the object name, Kubernetes also assigns a **Unique ID** (**UID**) to every object to distinguish historical occurrences of similar entities.

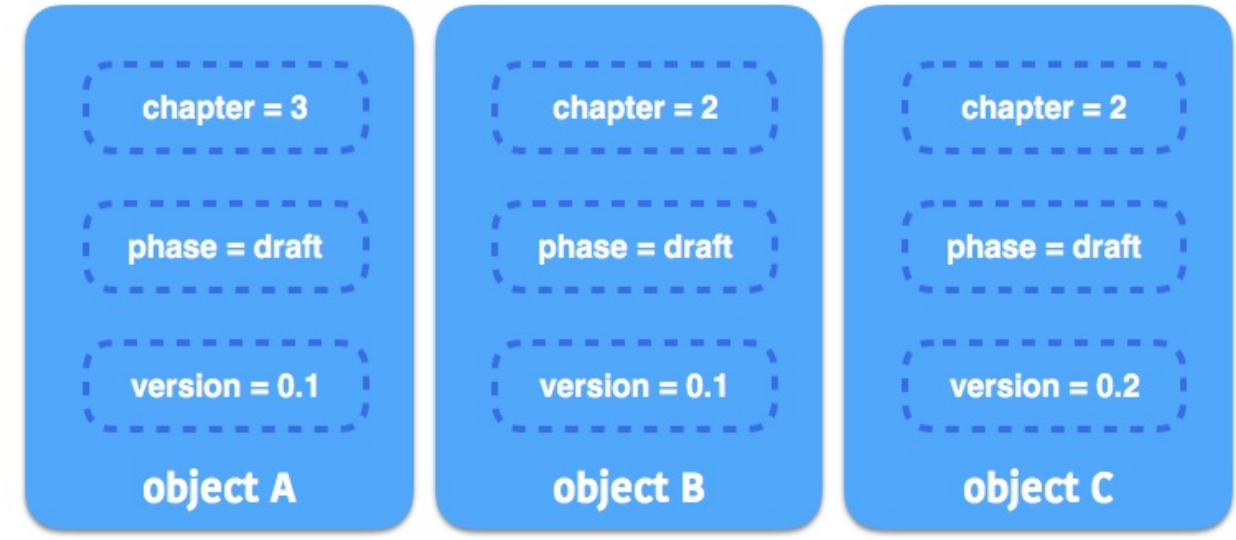
Label and selector

Labels are sets of key/pair values that attach to objects. They're designed to provide meaningful, identifying information about objects. Common usages are to indicate the name of the micro-service, the tier, the environment, and the software version. Users can define meaningful labels that could be used with selectors later. The syntax of labels in an object `spec` is as follows: **labels:**
$key1: $value1
$key2: $value2

Along with labels, label selectors are used to filter sets of objects. Separated by commas, multiple requirements will be joined by the `AND` logical operator. There are two ways to filter:

- Equality-based requirements
- Set-based requirements

Equality-based requirements support the following operators: `=`, `==`, and `!=`. Take the following diagram as an example: if the selector is `chapter=2,version!=0.1`, the result will be **object C**. If the requirement is `version=0.1`, the result will be **object A** and **object B**:



Selector example

If we write the requirement in the supported object `spec`, it'll be as follows:
selector:
$key1: $value1

Set-based requirement supports `in`, `notin`, and `exists` (for `key` only). For example, if the requirement is `chapter in (3, 4),version`, then **object A** will be returned. If the requirement is `version notin (0.2), !author_info`, the result will be **object A** and **object B**. The following example shows an object `spec` that uses set-based requirements:

```
selector:
  matchLabels:
    $key1: $value1
  matchExpressions:
  - {key: $key2, operator: In, values: [$value1, $value2]}
```

The requirements of `matchLabels` and `matchExpressions` are combined together. This means that the filtered objects need to be true for both requirements.

annotations:
 $key1: $value1
 $key2: $value2

Namespace, name, label, and annotation are located in the metadata section of the object `spec`. Selector is located in the `spec` section of selector-supported resources, such as pod, service, ReplicaSet, and deployment.

Pods

A pod is the smallest deployable unit in Kubernetes. It can contain one or more containers. Most of the time, we just need one container per pod. In some special cases, more than one container is included in the same pod, such as sidecar containers (http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html). Containers in the same pod run in a shared context, on the same node, sharing the network namespace and shared volumes. Pods are also designed as mortal. When a pod dies for some reason, for example, if it's killed by Kubernetes controller if resources are lacking, it won't recover by itself. Instead, Kubernetes uses controllers to create and manage the desired state of pods for us.

We can use `kubectl explain <resource>` to get the detailed description of the resource by the command line. This will show the fields that the resource supports: **// get detailed info for `pods`**
 # kubectl explain pods
KIND: Pod
VERSION: v1

DESCRIPTION:
Pod is a collection of containers that can run on a host. This resource is created by clients and scheduled onto hosts.

FIELDS:
apiVersion <string>
APIVersion defines the versioned schema of this representation of an object. Servers should convert recognized schemas to the latest internal value, and may reject unrecognized values. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#resources

kind <string>
Kind is a string value representing the REST resource this object represents. Servers may infer this from the endpoint the client submits requests to. Cannot be updated. In CamelCase. More info:

https://git.k8s.io/community/contributors/devel/api-conventions.md#types-kinds

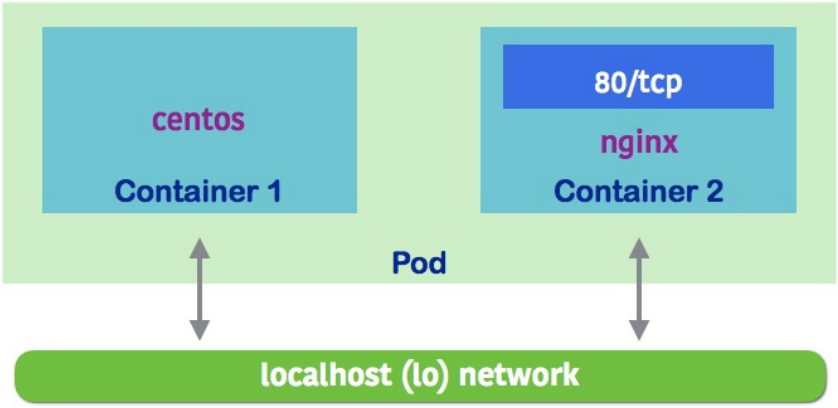
metadata <Object>
Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#metadata

spec <Object>
Specification of the desired behavior of the pod. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status

status <Object>
Most recently observed status of the pod. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/api-conventions.md#spec-and-status

In the following example, we'll show how to create two containers in a pod, and demonstrate how they access each other. Please note that this is neither a meaningful nor a classic sidecar pattern example. Instead, it's just an example of how we can access other containers within a pod: **// an example for creating co-located and co-scheduled container by pod**
 # cat 3-2-1_pod.yaml
apiVersion: v1
kind: Pod
metadata:
name: example
spec:
containers:
- name: web
image: nginx
- name: centos
image: centos
command: ["/bin/sh", "-c", "while : ;do curl http://localhost:80/; sleep 10; done"]

The following diagram shows the relationship between containers in a **Pod**. They share the same network namespace:



Containers inside a pod are visible via localhost This spec will create two containers, `web` and `centos`. Web is an `nginx` container (`https://hub.docker.com/_/nginx/`). The container port `80` is exposed by default. Since `centos` shares the same context as nginx, when using `curl` in `http://localhost:80/`, it should be able to access nginx.

Next, use the `kubectl create` command to launch the pod. The `-f` argument allows us to feed a configuration file to the `kubectl` command and creates the desired resources specified in the file: **// create the resource by `kubectl create` - Create a resource by filename or stdin**
 # kubectl create -f 3-2-1_pod.yaml pod "example" created

If we add `--record=true` at the end of the `kubectl` command when we create the resources, Kubernetes will add the latest command while creating or updating this resource. Therefore, we won't forget which resources are created by which spec.

We can use the `kubectl get <resource>` command to get the current status of the object. In this case, we use the `kubectl get pods` command: **// get the current running pods**
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
example 0/2 ContainerCreating 0 1s

Adding `--namespace=$namespace_name` allows us to access the object in different namespaces. The following is an example of how to check the pods in the `kube-system` namespace, which is used by system-type pods:

```
// list pods in kube-system namespace
# kubectl get pods --namespace=kube-system
NAME READY STATUS RESTARTS AGE
coredns-99b9bb8bd-p2dvw 1/1 Running 0 1m
etcd-minikube 1/1 Running 0 47s
kube-addon-manager-minikube 1/1 Running 0 13s
kube-apiserver-minikube 1/1 Running 0 38s
kube-controller-manager-minikube 1/1 Running 0 32s
kube-proxy-pvww2 1/1 Running 0 1m
kube-scheduler-minikube 1/1 Running 0 26s
kubernetes-dashboard-7db4dc666b-f8b2w 1/1 Running 0 1m
storage-provisioner 1/1 Running 0 1m
```

The status of our example pod is `ContainerCreating`. In this phase, Kubernetes has accepted the request and is trying to schedule the pod and pull down the image. Zero containers are currently running.

Most objects have short names, which come in handy when we use `kubectl get <object>` to list their status. For example, pods could be called `po`, services could be called `svc`, and deployment could be called `deploy`. Type `kubectl get` to know more. Alternatively, the `kubectl api-resources` command could list all resources with their short names and attributes.

After waiting a moment, we could get the status again: **// get the current running pods**
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
example 2/2 Running 0 3s

We can see that two containers are currently running. The uptime is three seconds. Using `kubectl logs <pod_name> -c <container_name>` gets `stdout` for the container, similar to `docker logs <container_name>`: **// get stdout for centos**
 # kubectl logs example -c centos
<!DOCTYPE html>
<html>
<head>

<title>Welcome to nginx!</title>
...

`centos` in the pod shares the same networking with nginx via localhost. Kubernetes creates a network container along with the pod. One of the functions in the network container is to forward the traffic between containers within a pod. We'll learn more about this in `Chapter 6`, *Kubernetes Network*.

If we specify labels in the pod spec, we could use the `kubectl get pods -l <requirement>` command to get the pods that satisfy the requirements, for example, `kubectl get pods -l 'tier in (frontend, backend)'`. Additionally, if we use `kubectl pods -o wide`, this will list which pods are running on which nodes.

We could use `kubectl describe <resource> <resource_name>` to get detailed information about a resource: **// get detailed information for a pod**
 # kubectl describe pods example Name: example
Namespace: default
Priority: 0
PriorityClassName: <none>
Node: minikube/10.0.2.15
Start Time: Sun, 07 Oct 2018 15:15:36 -0400
Labels: <none>
Annotations: <none>
Status: Running
IP: 172.17.0.4
Containers: ...

At this point, we know which node this pod is running on. In `minikube`, we only get a single node so it won't make any difference. In the real cluster environment, knowing which node the pod is running on is useful for troubleshooting. We haven't associated any labels, annotations, or controllers for it: **web:**
Container ID: docker://d8284e14942cbe0b8a91f78afc132e09c0b522e8a311e44f6a9a60ac2ca7103a
Image: nginx
Image ID: docker-pullable://nginx@sha256:9ad0746d8f2ea6df3a17ba89eca40b48c47066dfab55a75e08e2b70fc80d929e
Port: <none>
Host Port: <none>
State: Running
Started: Sun, 07 Oct 2018 15:15:50 -0400
Ready: True
Restart Count: 0
Environment: <none>
Mounts:
/var/run/secrets/kubernetes.io/serviceaccount from default-token-bm6vn (ro)

In the containers section, we'll see there are two containers included in this pod. We can see their states, source images, port mappings, and restart count: **Conditions:**
Type Status
Initialized True
Ready True
ContainersReady True
PodScheduled True

A pod has `PodStatus`, which includes a map of array representations as `PodConditions`. The possible types of `PodConditions` are `PodScheduled`, `Ready`, `Initialized`, `Unschedulable`, and `ContainersReady`. The value will be `true`, `false`, or unknown. If the pod isn't created accordingly, `PodStatus` will give us a brief overview of which part failed. In the preceding example, we launched the pod successfully in each phase without any errors: **Volumes:**
default-token-bm6vn:
Type: Secret (a volume populated by a Secret)
SecretName: default-token-bm6vn
Optional: false

A pod is associated with a service account that provides an identity for processes that are running the pod. It's controlled by service account and a token controller in the API Server.

It'll mount a read-only volume to each container under `/var/run/secrets/kubernetes.io/serviceaccount` in a pod that contains a token for API access. Kubernetes creates a default service account. We can use the `kubectl get serviceaccounts` command to list the service accounts: **QoS Class: BestEffort**
Node-Selectors: <none>
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
node.kubernetes.io/unreachable:NoExecute for 300s

We don't assign any selectors to this pod yet. Toleration is used to restrict how many pods a node can use. We'll learn more about this in `Chapter 8`, *Resource Management and Scaling*: **Events:**
Type Reason Age From Message
---- ------ ---- ---- -------
Normal Scheduled 2m22s default-scheduler Successfully assigned default/example to minikube
Normal Pulling 2m21s kubelet, minikube pulling image "nginx"
Normal Pulled 2m8s kubelet, minikube Successfully pulled image "nginx"
Normal Created 2m8s kubelet, minikube Created container
Normal Started 2m8s kubelet, minikube Started container
Normal Pulling 2m8s kubelet, minikube pulling image "centos"
Normal Pulled 93s kubelet, minikube Successfully pulled image "centos"
Normal Created 92s kubelet, minikube Created container
Normal Started 92s kubelet, minikube Started container

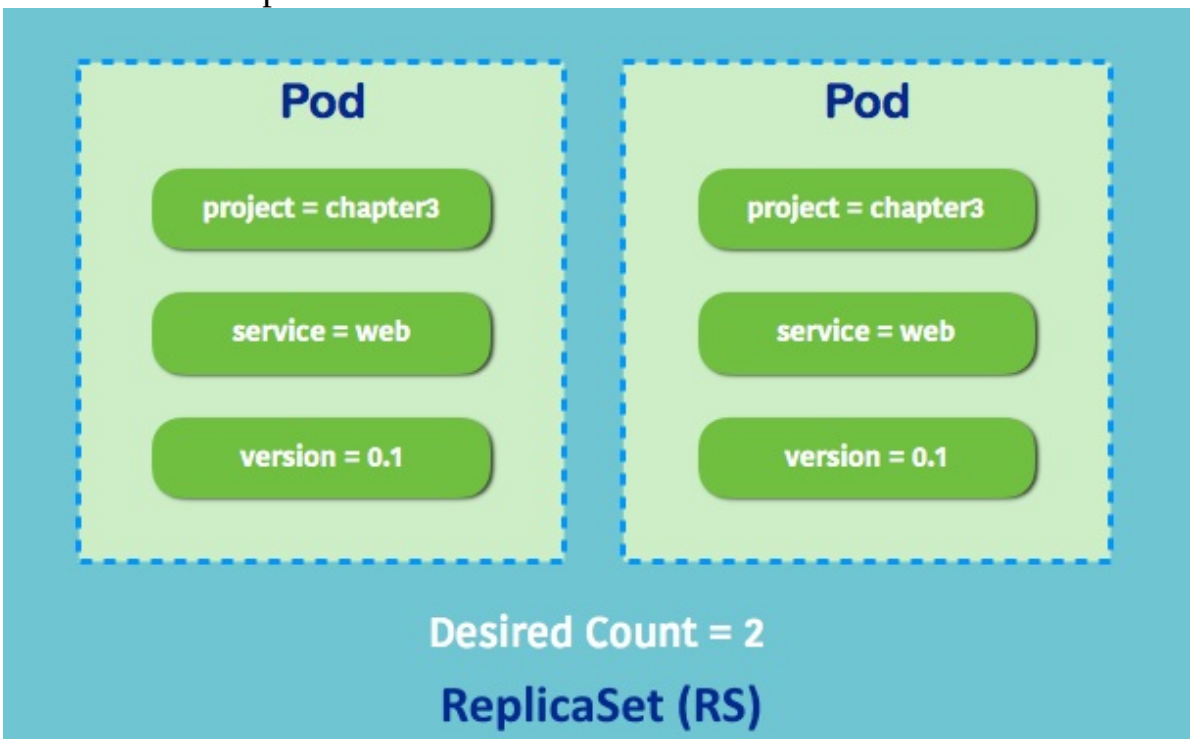
By seeing the events, we can identify the steps required for Kubernetes to run a node. First, the scheduler assigns the task to a node, which here is called `minikube`. Then, kubelet starts pulling the first image and creates a container accordingly. After that, kubelet pulls down the second container and starts the container.

ReplicaSet

A pod isn't self-healing. When a pod encounters failure, it won't recover on its own. This is where **ReplicaSet** (**RS**) comes into play. ReplicaSet ensures that the specified number of replica pods are always up and running in the cluster. If a pod crashes for any reason, ReplicaSet will send a request to spin up a new pod.

*ReplicaSet is similar to **ReplicationController** (**RC**), which was used in older versions of Kubernetes. Unlike ReplicaSet, which uses set-based selector requirement, ReplicationController used equality-based selector requirements. It has now been completely replaced by ReplicaSet.*

Let's see how ReplicaSet works:



ReplicaSet with a desired count of 2

Let's say that we want to create a `ReplicaSet` object, with a desired count of 2. This means that we'll always have two pods in the service. Before we write the spec for ReplicaSet, we'll have to decide on the pod template first. This is similar to the spec of a pod. In a ReplicaSet, labels are required in the metadata section. A ReplicaSet uses a pod selector to select which pods it manages. Labels allow

ReplicaSet to distinguish whether all of the pods matching the selectors are all on track.

In this example, we'll create two pods, each with the labels `project`, `service`, and `version`, as shown in the preceding diagram: **// an example for RS spec**

```
# cat 3-2-2_rs.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: nginx
spec:
replicas: 2
selector:
matchLabels:
project: chapter3
matchExpressions:
- {key: version, operator: In, values: ["0.1", "0.2"]}
template:
metadata:
name: nginx
labels:
project: chapter3
service: web
version: "0.1"
spec:
containers:
- name: nginx
image: nginx
ports:
- containerPort: 80

// create the RS
# kubectl create -f 3-2-2_rs.yaml
replicaset.apps/nginx created
```

Then, we can use `kubectl` to get the current RS status: **// get current RSs**

```
# kubectl get rs
```

NAME DESIRED CURRENT READY AGE
nginx 2 2 2 29s

This shows that we desire two pods, we currently have two pods, and two pods are ready. How many pods do we have now? Let's check it out via the `kubectl` command: **// get current running pod**
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-l5mdn 1/1 Running 0 11s
nginx-pjjw9 1/1 Running 0 11s

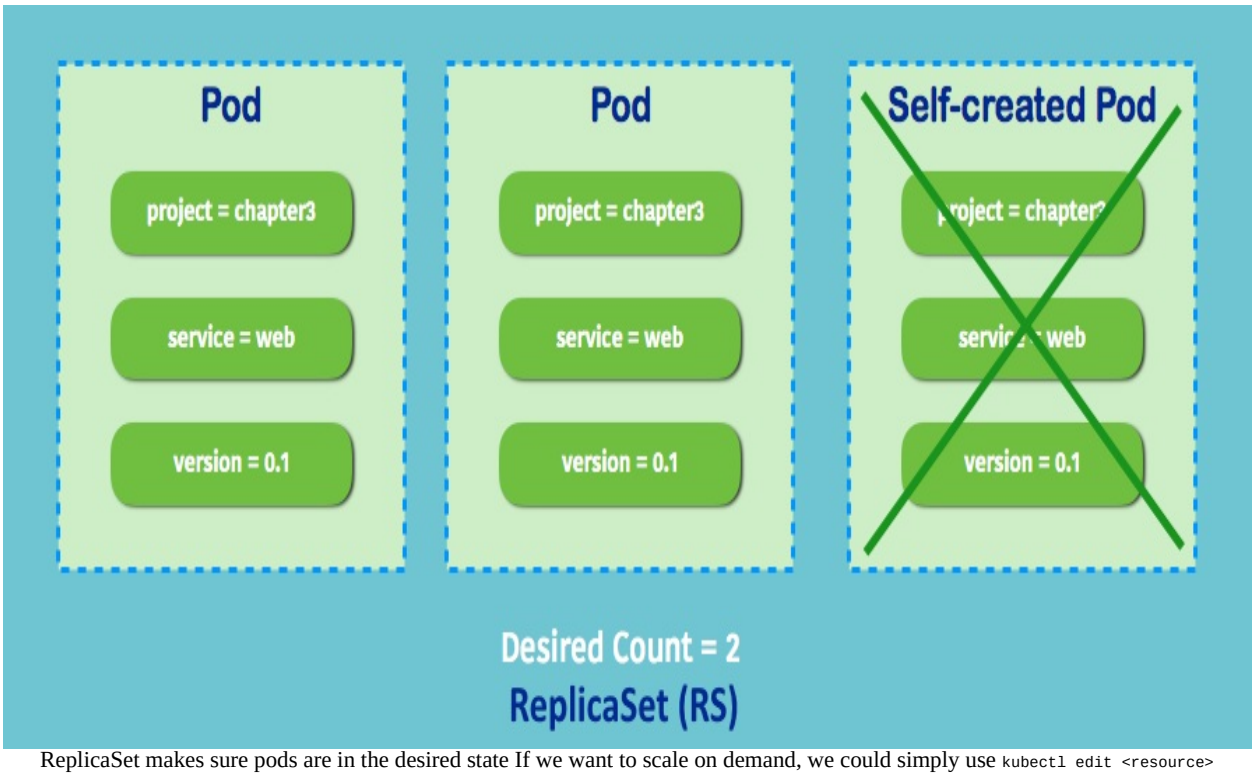
This shows we have two pods up and running. As described previously, ReplicaSet manages all of the pods matching the selector. If we create a pod with the same label manually, in theory, it should match the pod selector of the RS we just created. Let's try it out: **// manually create a pod with same labels**
 # cat 3-2-2_rs_self_created_pod.yaml
apiVersion: v1
kind: Pod
metadata:
name: our-nginx
labels:
project: chapter3
service: web
version: "0.1"
spec:
containers:
- name: nginx
image: nginx
ports:
- containerPort: 80
// create a pod with same labels manually
 # kubectl create -f 3-2-2_rs_self_created_pod.yaml
pod "our-nginx" created

Let's see if it's up and running: **// get pod status**
 # kubectl get pods

NAME READY STATUS RESTARTS AGE
nginx-l5mdn 1/1 Running 0 4m
nginx-pjjw9 1/1 Running 0 4m
our-nginx 0/1 Terminating 0 4s

It's scheduled, and ReplicaSet catches it. The amount of pods becomes three, which exceeds our desired count. The pod is eventually killed: **// get pod status**
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-l5mdn 1/1 Running 0 5m
nginx-pjjw9 1/1 Running 0 5m

The following diagram is an illustration of how our self-created pod was evicted. The labels are matched with ReplicaSet, but the desired count is 2. Therefore, the additional pod was evicted:



ReplicaSet makes sure pods are in the desired state If we want to scale on demand, we could simply use `kubectl edit <resource> <resource_name>` to update the spec. Here, we'll change the replica count from `2` to `5`: **// change replica count from 2 to 5, default system editor will pop out.**
Change `replicas` number
 # kubectl edit rs nginx
replicaset.extensions/nginx edited

Let's check the RS information: **// get RS information**
 # kubectl get rs
NAME DESIRED CURRENT READY AGE

nginx 5 5 5 5m

We now have five pods. Let's check how RS works: **// describe RS resource `nginx`**
 # kubectl describe rs nginx
Name: nginx
Namespace: default
Selector: project=chapter3,version in (0.1,0.2)
Labels: project=chapter3
service=web
version=0.1
Annotations: <none>
Replicas: 5 current / 5 desired
Pods Status: 5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
Labels: project=chapter3
service=web
version=0.1
Containers:
nginx:
Image: nginx
Port: 80/TCP
Host Port: 0/TCP
Environment: <none>
Mounts: <none>
Volumes: <none>
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal SuccessfulCreate 3m34s replicaset-controller Created pod: nginx-l5mdn
Normal SuccessfulCreate 3m34s replicaset-controller Created pod: nginx-pjjw9
Normal SuccessfulDelete 102s replicaset-controller Deleted pod: our-nginx
Normal SuccessfulCreate 37s replicaset-controller Created pod: nginx-v9trs
Normal SuccessfulCreate 37s replicaset-controller Created pod: nginx-n95mv
Normal SuccessfulCreate 37s replicaset-controller Created pod: nginx-xgdhq

By describing the command, we can learn the spec of RS and the events. When we created the `nginx` RS, it launched two containers by spec. Then, we created another pod manually by another spec, named `our-nginx`. RS detected that the pod matches its pod selector. After the amount exceeded our desired count, it evicted it. Then, we scaled out the replicas to five. RS detected that it didn't fulfill our desired state and launched three pods to fill the gap.

If we want to delete an RC, simply use the `kubectl` command: `kubectl delete <resource> <resource_name>`. Since we have a configuration file on hand, we could also use `kubectl delete -f <configuration_file>` to delete the resources listing in the file: **// delete a rc**
 # kubectl delete rs nginx
replicaset.extensions/nginx deleted

// get pod status
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-pjjw9 0/1 Terminating 0 29m

Deployments

Deployments are the best primitive to manage and deploy our software in Kubernetes after version 1.2. They allow us to deploy pods, carry out rolling updates, and roll back pods and ReplicaSets. We can define our desired software updates declaratively using Deployments and then Deployments will do them for us progressively.

Before Deployments, ReplicationController and kubectl rolling-update were the major ways to implement rolling updates for software. These methods were much more imperative and slower. Deployment is now the main high-level object used to manage our application.

Let's take a look at how it works. In this section, we'll get a taste of how a Deployment is created, how to perform rolling updates, and rollbacks. `Chapter 9`, *Continuous Delivery*, has more information with practical examples about how we can integrate Deployments into our continuous delivery pipeline.

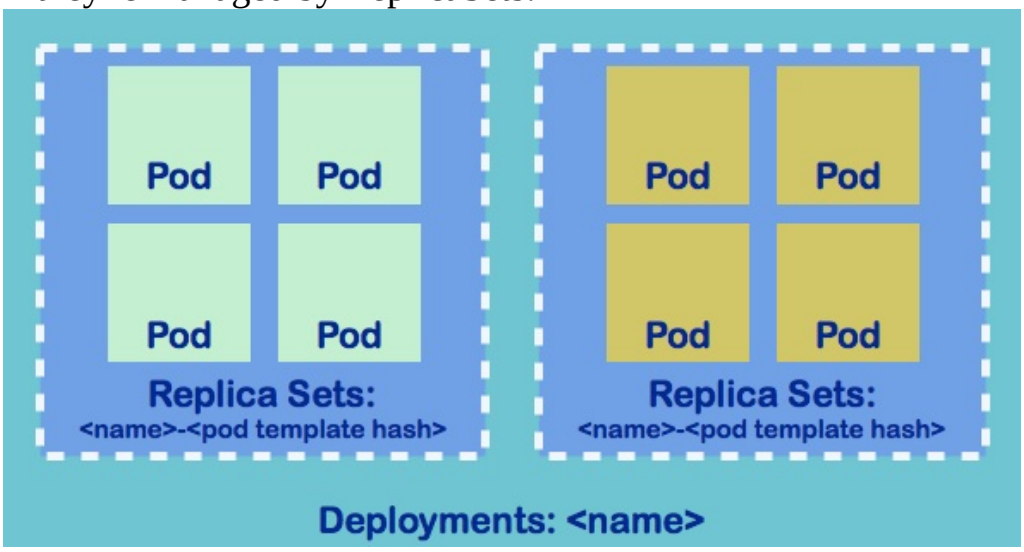
First, we use the `kubectl run` command to create `deployment` for us: **// using kubectl run to launch the Pods**
 # kubectl run nginx --image=nginx:1.12.0 --replicas=2 --port=80
deployment "nginx" created
// check the deployment status
 # kubectl get deployments
NAME DESIRED CURRENT UP-TO-DATE AVAILABLE AGE
nginx 2 2 2 2 4h

Before Kubernetes 1.2, the `kubectl run` command would create pods instead.

There are two pods that are deployed by `deployment`: **// check if pods match our desired count**
 # kubectl get pods
NAME READY STATUS RESTARTS AGE
nginx-2371676037-2brn5 1/1 Running 0 4h
nginx-2371676037-gjfhp 1/1 Running 0 4h

The following is a diagram of the relationship between Deployments,

ReplicaSets, and pods. In general, Deployments manage ReplicaSets and ReplicaSets manage pods. Note that we shouldn't manipulate ReplicaSets that are managed by Deployments, just like there's no reason to directly change pods if they're managed by ReplicaSets:



The relationship between Deployments, ReplicaSets, and pods If we delete one of the pods, the replaced pod will be scheduled and launched immediately. This is because Deployments create a ReplicaSet behind the scenes, which will ensure that the number of replicas matches our desired count: **// list replica sets**
 # kubectl get rs
NAME DESIRED CURRENT READY AGE
nginx-2371676037 2 2 2 4h

We could also expose the port for deployment using the `kubectl` command: **// expose port 80 to service port 80**
 # kubectl expose deployment nginx --port=80 --target-port=80
service "nginx" exposed
// list services
 # kubectl get services
NAME CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes 10.0.0.1 <none> 443/TCP 3d
nginx 10.0.0.94 <none> 80/TCP 5s

Deployments can be created by spec as well. The previous Deployments and Service launched by kubectl can be converted into the following spec: **// create deployments by spec**
 # cat 3-2-3_deployments.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx
spec:
replicas: 2
template:
metadata:
labels:
run: nginx
spec:
containers:
- name: nginx
image: nginx:1.12.0
ports:
- containerPort: 80
---

kind: Service
apiVersion: v1
metadata:
name: nginx
labels:
run: nginx
spec:
selector:
run: nginx
ports:
- protocol: TCP
port: 80
targetPort: 80
name: http
// create deployments and service
 # kubectl create -f 3-2-3_deployments.yaml
deployment "nginx" created
service "nginx" created

In order to perform rolling updates, we'll need to add a rolling update strategy. There are three parameters used to control the process:

Parameters	Description	Default value
`minReadySeconds`	This is the warm-up time and indicates how long a newly created pod is considered to be available. By default, Kubernetes assumes the application will be available once it's successfully launched.	0
`maxSurge`	This indicates how many pods can be surged when carrying out rolling update processes.	25%
`maxUnavailable`	This indicates how many pods can be unavailable when carrying out rolling update processes.	25%

`minReadySecond` is an important setting. If our application isn't available immediately when the pod is up, the pods will roll too fast without proper waiting. Although all of the new pods are up, the application might be still warming up; there's a chance that a service outage might occur. In the following example, we'll add the configuration into the `Deployment.spec` section: **// add to Deployments.spec, save as 3-2-3_deployments_rollingupdate.yaml**
minReadySeconds: 3
strategy:
type: RollingUpdate
rollingUpdate:

maxSurge: 1
maxUnavailable: 1

This indicates that we allow only one of the pods to be unavailable at any time and one pod to be launched when rolling the pods. The warm-up time before proceeding to the next operation is three seconds. We can use either `kubectl edit deployments nginx` (edit directly) or `kubectl replace -f 3-2-3_deployments_rollingupdate.yaml` to update the strategy.

Let's say we want to simulate a new software rollout from nginx 1.12.0 to 1.13.1. We can still use the preceding two commands to change the image version or use `kubectl set image deployment nginx nginx=nginx:1.13.1` to trigger the update. If we use `kubectl describe` to check what's going on, we'll see that Deployments have triggered rolling updates on ReplicaSets by deleting/creating pods: **// list rs**
 # kubectl get rs
NAME DESIRED CURRENT READY AGE
nginx-596b999b89 2 2 2 2m

// check detailed rs information
 # kubectl describe rs nginx-596b999b89
Name: nginx-596b999b89
Namespace: default
Selector: pod-template-hash=1526555645,run=nginx
Labels: pod-template-hash=1526555645
run=nginx
Annotations: deployment.kubernetes.io/desired-replicas: 2
deployment.kubernetes.io/max-replicas: 3
deployment.kubernetes.io/revision: 1
Controlled By: Deployment/nginx
Replicas: 2 current / 2 desired
Pods Status: 2 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
Labels: pod-template-hash=1526555645
run=nginx
Containers:
nginx:
Image: nginx:1.12.0
Port: 80/TCP
Host Port: 0/TCP
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
Normal SuccessfulCreate 3m41s replicaset-controller Created pod: nginx-596b999b89-th9rx
Normal SuccessfulCreate 3m41s replicaset-controller Created pod: nginx-596b999b89-2pp7b

The following is a diagram of how rolling update works in a Deployment:

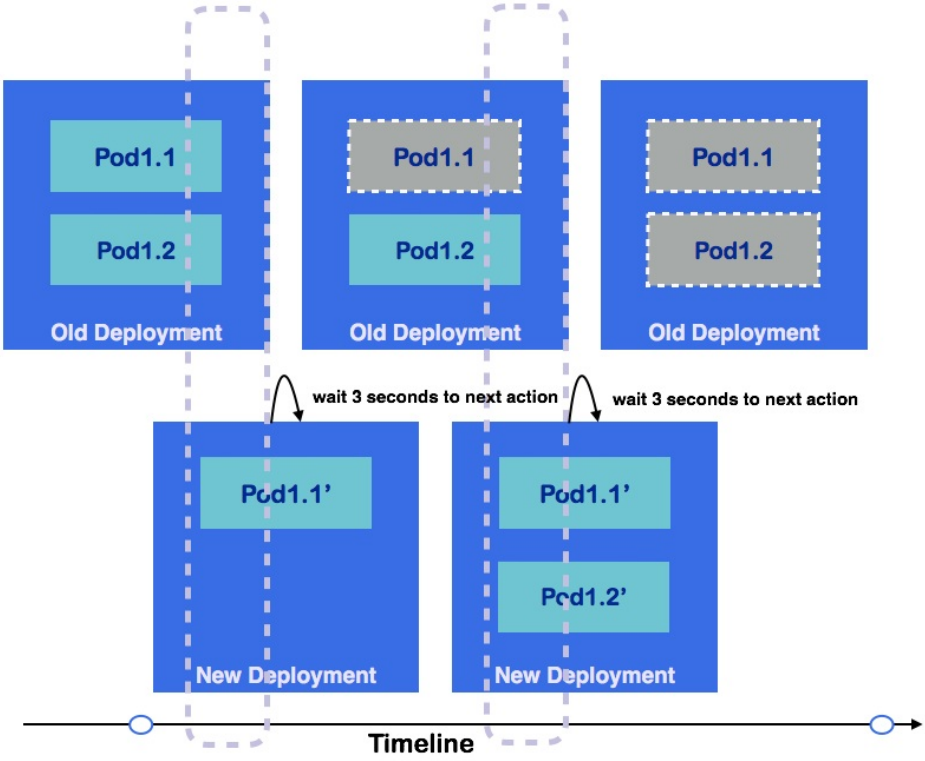


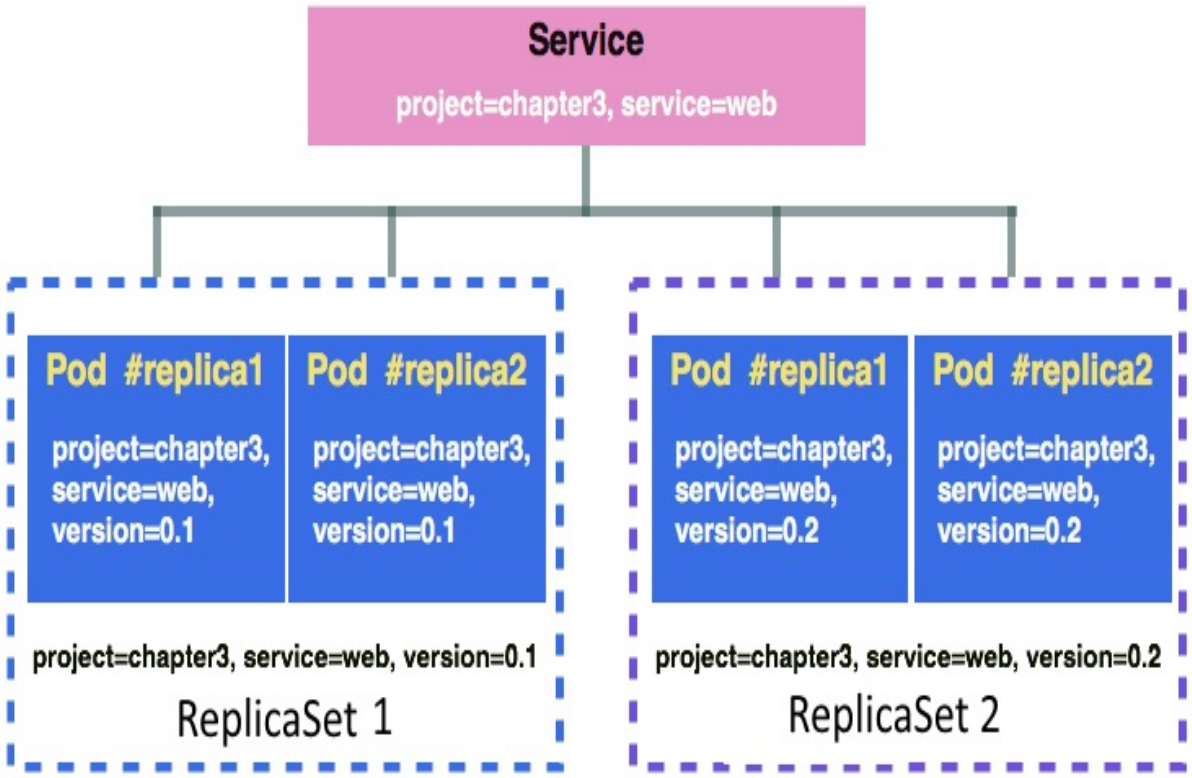
Illustration of a Deployment The preceding diagram shows an illustration of a Deployment. At a certain point in time, our desired count is 2 and we have one maxSurge pod. After launching each new pod, Kubernetes will wait three seconds (`minReadySeconds`) and then perform the next action.

If we use the `kubectl set image deployment nginx nginx=nginx:1.12.0` command to roll back to the previous version 1.12.0, Deployments will do the rollback for us.

Services

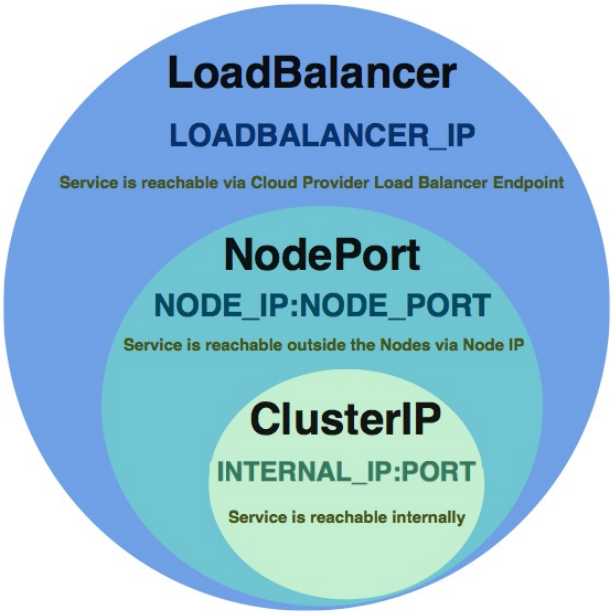
Services in Kubernetes are abstraction layers for routing traffic to a logical set of pods. With Services, we don't need to trace the IP address of each pod. Services usually use the label selector to select the pods that they need to route to while, in some cases, Services are created without a selector on purpose. The Service abstraction is powerful. It enables decoupling and makes communication between micro-services possible. Currently, Kubernetes Services support TCP, UDP, and SCTP.

Services don't care about how we create the pod. Just like ReplicaSet, it only cares that the pods match its label selectors, so the pods could belong to different ReplicaSets:



Service maps pods via label selector In the preceding diagram, all of the pods match the service selector, `project=chapter3, service=web`, so the Service will be responsible for distributing the traffic into all of the pods without explicit assignment.

There are four types of Services: `ClusterIP`, `NodePort`, `LoadBalancer`, and `ExternalName`:



LoadBalancer includes the features of NodePort and ClusterIP

ClusterIP

`ClusterIP` is the default Service type. It exposes the Service on a cluster-internal IP. Pods in the cluster could reach the Service via the IP address, environment variables, or DNS. In the following example, we'll learn how to use both native Service environment variables and DNS to access the pods behind Services in the cluster.

Before starting a Service, we'd like to create two sets of RS with different version labels, as follows: **// create RS 1 with nginx 1.12.0 version**
 # cat 3-2-3_rs1.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: nginx-1.12
spec:
replicas: 2
selector:
matchLabels:
project: chapter3
service: web
version: "0.1"
template:
metadata:
name: nginx
labels:
project: chapter3
service: web
version: "0.1"
spec:
containers:
- name: nginx
image: nginx:1.12.0
ports:
- containerPort: 80

// create RS 2 with nginx 1.13.1 version
 # cat 3-2-3_rs2.yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
name: nginx-1.13
spec:
replicas: 2
selector:
matchLabels:
project: chapter3
service: web
version: "0.2"
template:
metadata:
name: nginx
labels:
project: chapter3
service: web
version: "0.2"
spec:
containers:
- name: nginx
image: nginx:1.13.1
ports:
- containerPort: 80

Then, we could make our pod selector, targeting project and service labels: **// simple nginx service**
 # cat 3-2-3_service.yaml
kind: Service
apiVersion: v1
metadata:
name: nginx-service
spec:
selector:

project: chapter3
service: web
ports:
- protocol: TCP
port: 80
targetPort: 80
name: http

// create the RSs
 # kubectl create -f 3-2-3_rs1.yaml
replicaset.apps/nginx-1.12 created
 # kubectl create -f 3-2-3_rs2.yaml
replicaset.apps/nginx-1.13 created
// create the service
 # kubectl create -f 3-2-3_service.yaml
service "nginx-service" created

Since a `service` object might create a DNS label, the service name must be a combination of alphanumeric characters and hyphens. A hyphen at the beginning or end of a label isn't allowed.

We can then use `kubectl describe service <service_name>` to check the Service information: **// check nginx-service information**
 # kubectl describe service nginx-service
Name: nginx-service
Namespace: default
Labels: <none>
Annotations: <none>
Selector: project=chapter3,service=web
Type: ClusterIP
IP: 10.0.0.188
Port: http 80/TCP

```
Endpoints:    172.17.0.5:80,172.17.0.6:80,172.17.0.7:80 + 1 more...
Session Affinity:  None
Events:      <none>
```

One Service could expose multiple ports. Just extend the `.spec.ports` list in the `service` spec.

We can see that it's a `ClusterIP` type Service and its assigned internal IP is

`10.0.0.188`. The endpoints show that we have four IPs behind the Service. The pod IPs can be found by the `kubectl describe pods <pod_name>` command. Kubernetes creates an `endpoints` object along with a `service` object to route the traffic to the matching pods.

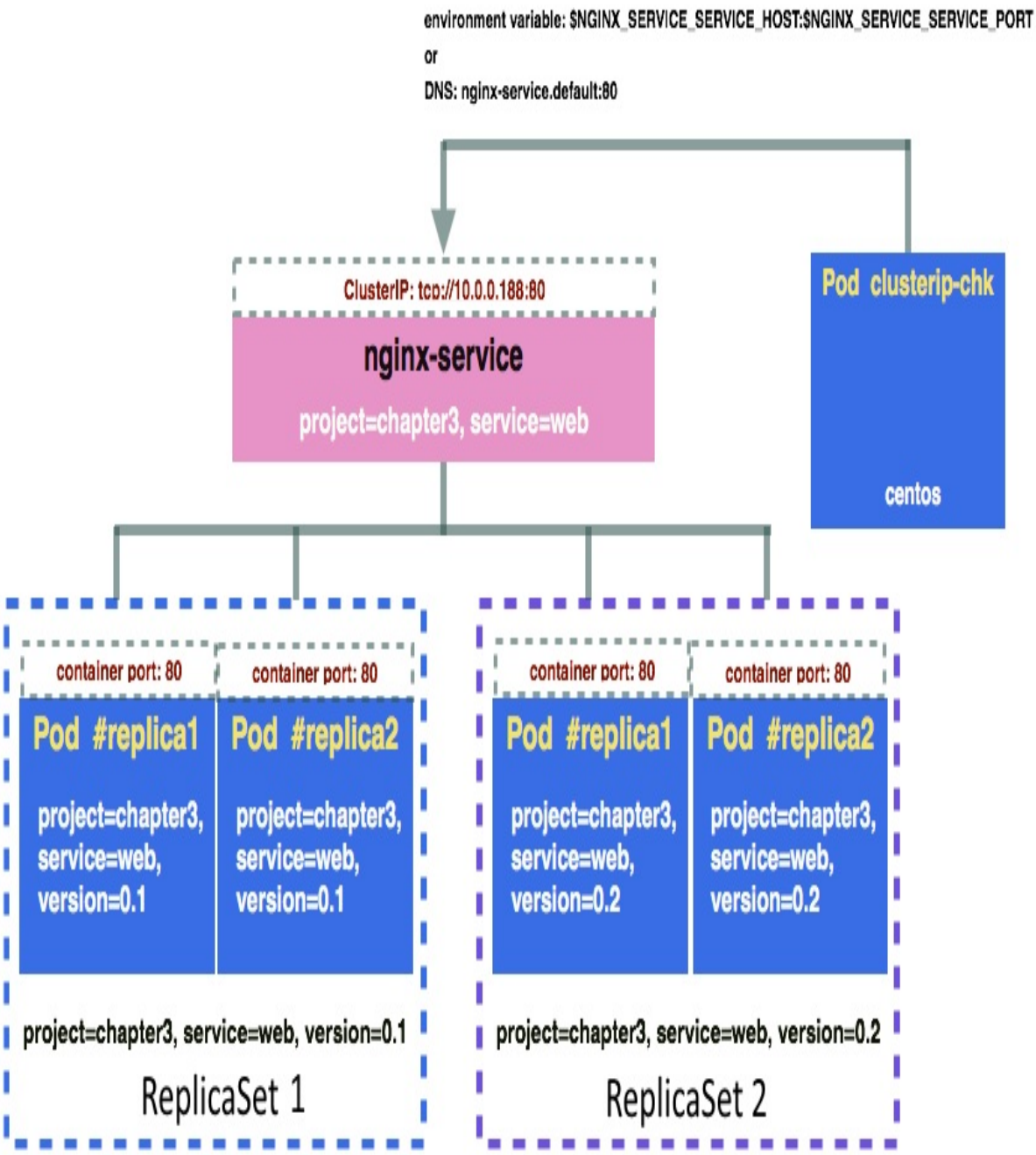
When the Service is created with selectors, Kubernetes will create corresponding endpoint entries and keep updating, which will indicate the destination that the Service routes to: **// list current endpoints. Nginx-service endpoints are created and pointing to the ip of our 4 nginx pods.**
 # kubectl get endpoints
NAME ENDPOINTS AGE
kubernetes 10.0.2.15:8443 2d nginx-service
172.17.0.5:80,172.17.0.6:80,172.17.0.7:80 + 1 more... 10s

The ClusterIP could be defined within your cluster, though most of the time we don't explicitly use the IP address to access clusters. Using `.spec.clusterIP` can do this for us.

By default, Kubernetes will expose seven environment variables for each Service. In most cases, the first two allow us to use the `kube-dns` add-on to carry out service discovery for us:

- `${SVCNAME}_SERVICE_HOST`
- `${SVCNAME}_SERVICE_PORT`
- `${SVCNAME}_PORT`
- `${SVCNAME}_PORT_${PORT}_${PROTOCAL}`
- `${SVCNAME}_PORT_${PORT}_${PROTOCAL}_PROTO`
- `${SVCNAME}_PORT_${PORT}_${PROTOCAL}_PORT`
- `${SVCNAME}_PORT_${PORT}_${PROTOCAL}_ADDR`

In the following example, we'll use `${SVCNAME}_SERVICE_HOST` in another pod to check whether we can access our nginx pods:



Accessing ClusterIP via environment variables and DNS names We'll then create a pod called `clusterip-chk` to access nginx containers via `nginx-service`: // **access nginx service via ${NGINX_SERVICE_SERVICE_HOST}**

 # cat 3-2-3_clusterip_chk.yaml
apiVersion: v1
kind: Pod
metadata:
name: clusterip-chk
spec:
containers:
- name: centos
image: centos
command: ["/bin/sh", "-c", "while : ;do curl

http://${NGINX_SERVICE_SERVICE_HOST}:80/; sleep 10; done"]

We can check `stdout` of the `cluserip-chk` pod via the `kubectl logs` command: **// check stdout, see if we can access nginx pod successfully**
 # kubectl logs -f clusterip-chk
% Total % Received % Xferd Average Speed Time Time Time Current Dload Upload Total Spent Left Speed
100 612 100 612 0 0 156k 0 --:--:-- --:--:-- --:--:-- 199k ... <title>Welcome to nginx!</title> ...

This abstraction level decouples the communication between pods. Pods are mortal. With RS and Services, we can build robust services without worrying about whether one pod will influence all microservices.

With the DNS server enabled, the pods in the same cluster and the same namespace as the Services can access Services via their DNS records.

CoreDNS GA was introduced in Kubernetes 1.11 and is now the default option in Kubernetes. Before this, the kube-dns add-on was in charge of DNS-based service discovery The DNS server creates DNS records for newly created services by watching the Kubernetes API. The DNS format for the cluster IP is `$servicename.$namespace` and the port is `_$portname_$protocal.$servicename.$namespace`. The spec of the `clusterip_chk` pod will be similar to the environment variables one. Change the URL to `http://nginx-service.default:_http_tcp.nginx-service.default/` in our previous example, and they should work exactly the same.

NodePort

If the Service is set as `NodePort`, Kubernetes will allocate a port within a certain range on each node. Any traffic going to the nodes on that port will be routed to the Service port. The port number may be user-specified. If not, Kubernetes will randomly choose a port between 30,000 and 32,767 that doesn't cause any collision. On the other hand, if it's specified, the user should be responsible for managing the collision by themselves. `NodePort` includes a `ClusterIP` feature. Kubernetes assigns an internal IP to the Service.

In the following example, we'll see how we can create a `NodePort` Service and use it:

```
// write a nodeport type service
# cat 3-2-3_nodeport.yaml
kind: Service
apiVersion: v1
metadata:
  name: nginx-nodeport
spec:
  type: NodePort
  selector:
    project: chapter3
    service: web
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

// create a nodeport service
# kubectl create -f 3-2-3_nodeport.yaml
service "nginx-nodeport" created
```

You should then be able to access the Service via `http://${NODE_IP}:80`. The node could be any node. The `kube-proxy` watches for any updates by the Service and the endpoints, and updates the iptable rules accordingly (if using default `iptables` proxy-mode).

If you're using minikube, you can access the Service via the `minikube service [-n NAMESPACE] [--url] NAME` command. In this example, this is `minikube service nginx-nodeport`.

LoadBalancer

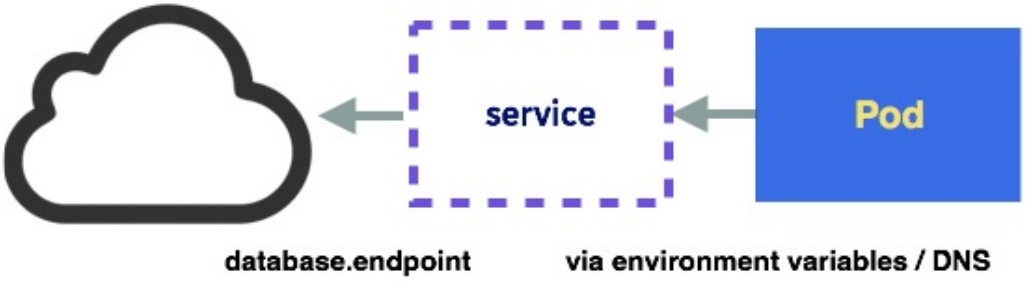
This type is only usable with cloud provider support, such as Amazon Web Services (Chapter 10, *Kubernetes on AWS*), Google Cloud Platform (Chapter 11, *Kubernetes on GCP*), and Azure (Chapter 12, *Kubernetes on Azure*). If we create a LoadBalancer Service, Kubernetes will provision a load balancer by the cloud provider to the Service.

ExternalName (kube-dns version >= 1.7)

Sometimes, we use different services in the cloud. Kubernetes is flexible enough to be hybrid. We can use ExternalName to create a CNAME for the external endpoints in the cluster.

Service without selectors

Services use selectors to match the pods to direct the traffic. However, sometimes you need to implement a proxy to be the bridge between the Kubernetes cluster and another namespace, another cluster, or an external resource. In the following example, we'll demonstrate how to implement a proxy for `http://www.google.com` in your cluster. This is just an example; the source of the proxy in your case might be the endpoint of your database or another resource in the cloud:



How a Service without a selector works The configuration file is similar to the previous one, just without the selector section: **// create a service without selectors**
 # cat 3-2-3_service_wo_selector_srv.yaml
kind: Service
apiVersion: v1
metadata:
name: google-proxy
spec:
ports:
- protocol: TCP
port: 80
targetPort: 80 // create service without selectors
 # kubectl create -f 3-2-3_service_wo_selector_srv.yaml
service "google-proxy" created

No Kubernetes endpoint will be created, since there's no selector. Kubernetes doesn't know where to route the traffic, since no selector can match the pods. We'll have to create the endpoints manually.

In the `Endpoints` object, the source addresses can't be the DNS name, so we'll use `nslookup` to find the current Google IP from the domain, and add it to `Endpoints.subsets.addresses.ip`: **// get an IP from google.com**
 # nslookup www.google.com
Server: 192.168.1.1
Address: 192.168.1.1#53 Non-authoritative answer:
Name: google.com
Address: 172.217.0.238
// create endpoints for the ip from google.com
 # cat 3-2-3_service_wo_selector_endpoints.yaml
kind: Endpoints
apiVersion: v1
metadata:
name: google-proxy
subsets:

- addresses:
- ip: 172.217.0.238
ports:
- port: 80 // create Endpoints
 # kubectl create -f 3-2-3_service_wo_selector_endpoints.yaml
endpoints "google-proxy" created

Let's create another pod in the cluster to access our Google proxy: **// pod for accessing google proxy**
 # cat 3-2-3_proxy-chk.yaml
apiVersion: v1
kind: Pod
metadata:
name: proxy-chk
spec:
containers:
- name: centos
image: centos
command: ["/bin/sh", "-c", "while : ;do curl -L http://${GOOGLE_PROXY_SERVICE_HOST}:80/; sleep 10; done"]

// create the pod
 # kubectl create -f 3-2-3_proxy-chk.yaml
pod "proxy-chk" created

Let's check `stdout` from the pod: **// get logs from proxy-chk**
 # kubectl logs proxy-chk
% Total % Received % Xferd Average Speed Time Time Time Current Dload Upload Total Spent Left Speed
100 219 100 219 0 0 2596 0 --:--:-- --:--:-- --:--:-- 2607
100 258 100 258 0 0 1931 0 --:--:-- --:--:-- --:--:-- 1931
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="en-CA">

...

Hurray! We can now confirm that the proxy works. The traffic to the Service will be routed to the endpoints we specified. If it doesn't work, make sure you add the proper inbound rules to the network of your external resources.

Endpoints don't support DNS as a source. Alternatively, we can use the ExternalName, which doesn't have selectors either. This requires kube-dns version >= 1.7.

In some use cases, users need neither load balancing nor proxy functionalities for the Service. In those cases, we can set `CluterIP = "None"` as a so-called headless service. For more information, please refer to `https://kubernetes.io/docs/concepts/services-networking/service/#headless-services`.

Volumes

A container is ephemeral and so is its disk. We either use the `docker commit [CONTAINER]` command or mount data volumes into a container (`Chapter 2`, *DevOps with Containers*). In the Kubernetes domain, volume management is critical, since pods might run on any node. Also, ensuring that containers in the same pod can share the same files becomes extremely hard. This is an important topic in Kubernetes. `Chapter 4`, *Managing Stateful Workloads*, introduces volume management.

Secrets

A secret, as its name suggests, is an object that stores secrets in key-value format for providing sensitive information to pods. It might be a password, an access key, or a token. A secret isn't stored in the disk; instead, it's stored in a per-node `tmpfs` filesystem. Kubelet on the node will create a `tmpfs` filesystem to store the secret. A secret isn't designed to store large amounts of data due to storage management considerations. The current size limit of one secret is 1 MB.

We can create a secret based on a file, a directory, or a specified literal value by launching kubectl to create a secret command or by the spec. There are three types of secret format: generic (or opaque, if encoded), docker registry, and TLS.

We'll use either the generic or opaque type in our application. The docker registry type is used to store the credentials of a private docker registry. A TLS secret is used to store the CA certificate bundle for cluster administration.

The `docker-registry` type of secret is also called `imagePullSecrets` and is used to pass the password of a private Docker registry via kubelet when pulling the image. This means we don't have to enter `docker login` for each provisioned node. The command is as follows: `kubectl create secret docker-registry <registry_name> --docker-server=<docker_server> --docker-username=<docker_username> --docker-password=<docker_password> --docker-email=<docker_email>`.

We'll start with a generic example to show how it works: **// create a secret by command line**
 # kubectl create secret generic mypassword --from-file=./mypassword.txt
secret "mypassword" created

The options for creating secrets based on a directory and a literal value are pretty similar to the file ones. If we specify a directory after `--from-file`, the files in the directory will be iterated. The filename will be the secret key if it's a legal secret name. Non-regular files, such as subdirectories, symlinks, devices, or pipes, will be ignored. On the other hand, `--from-literal=<key>=<value>` is the option to use if you want to specify plain text directly from the command, for example, `--from-literal=username=root`.

Here, we create a secret name, `mypassword`, from the `mypassword.txt` file. By default, the key of the secret is the filename, which is equivalent to the `--from-file=mypassword=./mypassword.txt` option. We could append multiple `--from-file` instances as well. We can use the `kubectl get secret -o yaml` command to see more

detailed information about the secret:

```
// get the detailed info of the secret
# kubectl get secret mypassword -o yaml
apiVersion: v1
data:
  mypassword: bXlwYXNzd29yZA==
kind: Secret
metadata:
  creationTimestamp: 2017-06-13T08:09:35Z
  name: mypassword
  namespace: default
  resourceVersion: "256749"
  selfLink: /api/v1/namespaces/default/secrets/mypassword
  uid: a33576b0-500f-11e7-9c45-080027cafd37
type: Opaque
```

We can see that the type of the secret becomes `Opaque` since the text has been encrypted by kubectl. It's `base64` encoded. We can use a simple `bash` command to decode it: **# echo "bXlwYXNzd29yZA==" | base64 --decode mypassword**

There are two ways for a pod to retrieve the secret. The first one is by a file, and the second one is by an environment variable. The first method is implemented by the volume. The syntax involves adding `containers.volumeMounts` in container specs and adding a volumes section with the secret configuration.

Retrieving secrets via files

Let's see how to read secrets from files inside a pod first: **// example for how a Pod retrieve secret**
 # cat 3-2-3_pod_vol_secret.yaml
apiVersion: v1
kind: Pod
metadata:
name: secret-access
spec:
containers:
- name: centos
image: centos
command: ["/bin/sh", "-c", "while : ;do cat /secret/password-example; sleep 10; done"]
volumeMounts:
- name: secret-vol
mountPath: /secret
readOnly: true
volumes:
- name: secret-vol
secret:
secretName: mypassword
items:
- key: mypassword
path: password-example

// create the pod
 # kubectl create -f 3-2-3_pod_vol_secret.yaml
pod "secret-access" created

The secret file will be mounted in `/<mount_point>/<secret_name>` without specifying `itemskey`, `path`, or `/<mount_point>/<path>` in the pod. In this case, the file path is `/secret/password-example`. If we describe the pod, we find that there are two mount points in this pod: the read-only volume that stores our secret and the one that

stores the credentials to communicate with the API servers, which is created and managed by Kubernetes. We'll learn more about this in `Chapter 6`, *Kubernetes Network*: **# kubectl describe pod secret-access**
...
Mounts:
/secret from secret-vol (ro)
/var/run/secrets/kubernetes.io/serviceaccount from default-token-jd1dq (ro)
...

We can delete a secret using the `kubectl delete secret <secret_name>` command.

After describing the pod, we can find a `FailedMount` event, since the volume no longer exists: **# kubectl describe pod secret-access**
...
FailedMount MountVolume.SetUp failed for volume "kubernetes.io/secret/28889b1d-5015-11e7-9c45-080027cafd37-secret-vol" (spec.Name: "secret-vol") pod "28889b1d-5015-11e7-9c45-080027cafd37" (UID: "28889b1d-5015-11e7-9c45-080027cafd37") with: secrets "mypassword" not found
...

If the pod is generated before a secret is created, the pod will encounter failure as well.

We'll now learn how to create a secret using the command line. We'll briefly introduce its spec format: **// secret example**
 # cat 3-2-3_secret.yaml
apiVersion: v1
kind: Secret
metadata:
name: mypassword
type: Opaque
data:
mypassword: bXlwYXNzd29yZA==

Since the spec is plain text, we need to encode the secret by our own `echo -n <password> | base64` command. Please note that the type here becomes `Opaque`. This should work in the same way as the one we create via the command line.

// example to use environment variable to retrieve the secret
cat 3-2-3_pod_ev_secret.yaml
apiVersion: v1
kind: Pod
metadata:
 name: secret-access-ev
spec: <br/> containers:
 - name: centos
 image: centos
 command: ["/bin/sh", "-c", "while : ;do echo $MY_PASSWORD; sleep 10; done"]
 env:
 - name: MY_PASSWORD
 valueFrom:
 secretKeyRef:
 name: mypassword
 key: mypassword
// create the pod
kubectl create -f 3-2-3_pod_ev_secret.yaml
pod "secret-access-ev" created

A secret should always be created before the pods that need it. Otherwise, the pods won't be launched successfully.

The declaration is under `spec.containers[].env[]`. We'll need the secret name and the key name. Both are `mypassword` in this case. The example should work the same as the one we looked at previously.

ConfigMap

ConfigMap is a resource that allows you to leave your configuration outside a Docker image. It injects the configuration data as key-value pairs into pods. Its properties are similar to secrets, but, whereas secrets are used to store sensitive data, such as passwords, ConfigMaps are used to store insensitive configuration data.

Like secrets, ConfigMaps could be based on files, directories, or specified literal value. They also have a similar syntax to secrets but use `kubectl create configmap`: **// create configmap**
 # kubectl create configmap example --from-file=config/app.properties --from-file=config/database.properties
configmap "example" created

Since two `config` files are located in the same folder name, `config`, we could pass a `config` folder instead of specifying the files one by one. The equivalent command to the preceding command is `kubectl create configmap example --from-file=config` in this case.

If we describe the ConfigMap, it'll show the current information: **// check out detailed information for configmap**
 # kubectl describe configmap example
Name: example
Namespace: default
Labels: <none>
Annotations: <none>
Data
====
app.properties:
----
name=DevOps-with-Kubernetes
port=4420
database.properties:
----

endpoint=k8s.us-east-1.rds.amazonaws.com
port=1521

We could use `kubectl edit configmap <configmap_name>` to update the configuration after creation.

We also could use `literal` as the input. The equivalent commands for the preceding example would be `kubectl create configmap example --from-literal=app.properties.name=name=DevOps-with-Kubernetes`. This isn't always very practical when we have many configurations in an app.

Let's see how to use this inside a pod. There are two ways to use ConfigMap inside a pod: by volume or environment variables.

cat 3-2-3_pod_vol_configmap.yaml
apiVersion: v1
kind: Pod
metadata:
 name: configmap-vol
spec:
 containers:
 - name: configmap
 image: centos
 command: ["/bin/sh", "-c", "while : ;do cat /src/app/config/database.properties; sleep 10; done"]
 volumeMounts:
 - name: config-volume
 mountPath: /src/app/config
 volumes:
 - name: config-volume
 configMap:
 name: example

// create configmap
kubectl create -f 3-2-3_pod_vol_configmap.yaml
pod "configmap-vol" created

// check out the logs
# kubectl logs -f configmap-vol
endpoint=k8s.us-east-1.rds.amazonaws.com
port=1521

We then could use this method to inject our non-sensitive configuration into the pod.

cat 3-2-3_pod_ev_configmap.yaml
apiVersion: v1
kind: Pod
metadata:
 name: configmap-ev
spec:
 containers:
 - name: configmap
 image: centos
 command: ["/bin/sh", "-c", "while : ;do echo $DATABASE_ENDPOINT; sleep 10; done"]
 env:
 - name: DATABASE_ENDPOINT
 valueFrom:
 configMapKeyRef:
 name: example
 key: database.properties
// create configmap
kubectl create -f 3-2-3_pod_ev_configmap.yaml
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

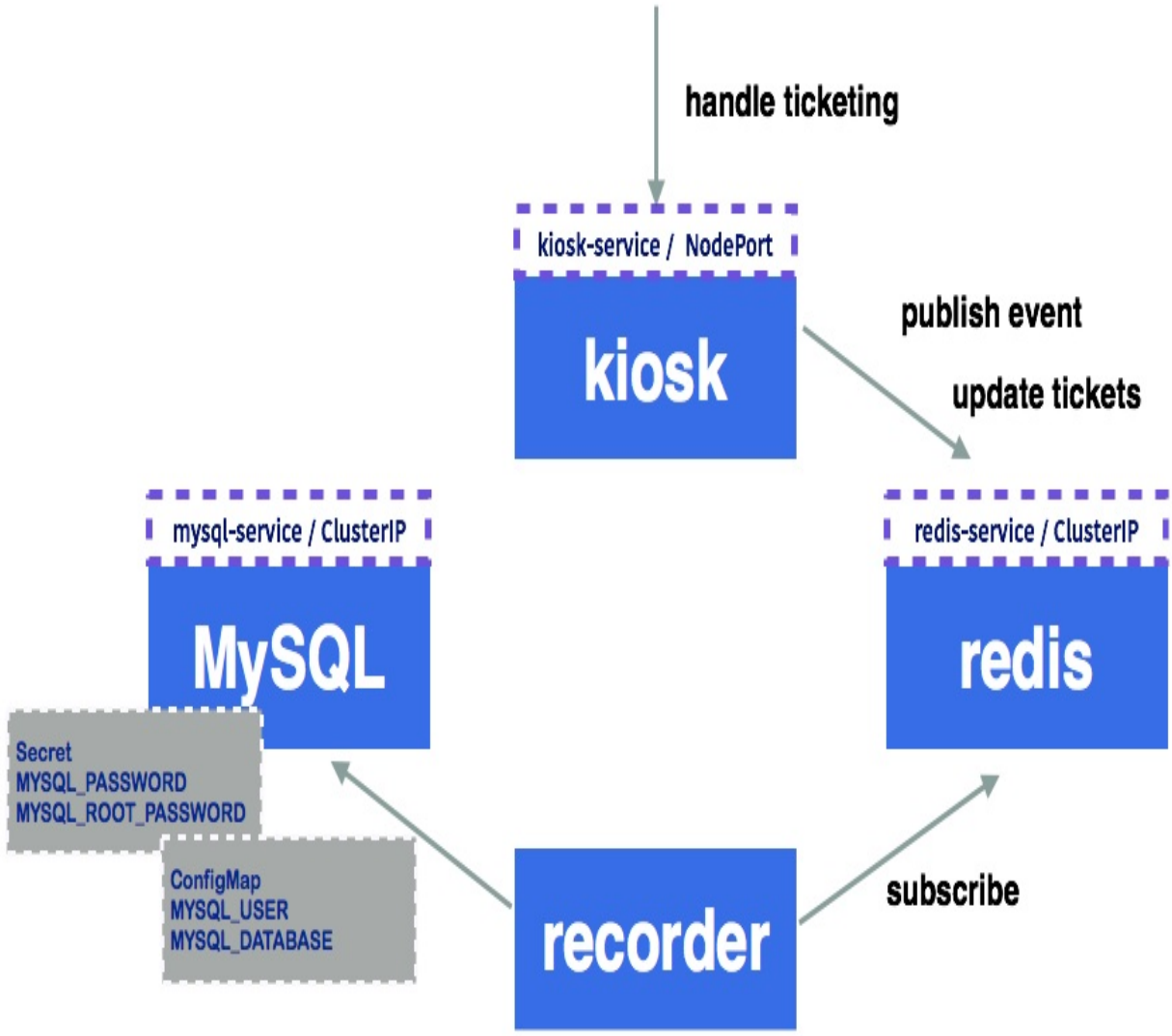
id="kobo.25.1">pod/configmap-ev created
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.26.1">// check out the logs
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.27.1"># kubectl logs configmap-ev
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.28.1">endpoint=k8s.us-east-1.rds.amazonaws.com port=1521

The Kubernetes system itself also uses ConfigMap to do some authentication. Check out the system ConfigMap by adding `--namespace=kube-system` in the `kubectl describe configmap` command.

Multi-container orchestration

In this section, we'll revisit our ticketing service: a kiosk web service as a frontend that provides an interface for get/put tickets. There is a Redis acting as cache to manage how many tickets we have. Redis also acts as a publisher/subscriber channel. Once a ticket is sold, the kiosk will publish an event into it. The subscriber is called recorder and will write a timestamp and record it to the MySQL database. Please refer to the last section in `Chapter 2`, *DevOps with Containers*, for a detailed Dockerfile and Docker compose implementation. We'll use `Deployment`, `Service`, `Secret`, `Volume`, and `ConfigMap` objects to implement this example in Kubernetes. The source code can be found at the following link: `https://github.com/DevOps-with-Kubernetes/examples/tree/master/chapter3/3-3_kiosk`.

The service architecture with Kubernetes resources is shown in the following diagram:



An example of a kiosk in the Kubernetes world We'll need four kinds of pods. Deployment is the best choice to manage or deploy the pods. This will reduce the effort required when we carry out deployments in the future thanks to its deployment strategy feature. Since the kiosk, Redis, and MySQL will be accessed by other components, we'll associate services to their pods. MySQL acts as a datastore and, for simplicity, we'll mount a local volume to it. Note that Kubernetes offers a bunch of choices. Check out the details and examples in `Chapter 4`, *Managing Stateful Workload*s. We'll want to store sensitive information such as our MySQL root and user password in secrets. The other insensitive configuration, such as the DB name or username, we'll leave to ConfigMap.

We'll launch MySQL first, as the recorder depends on it. Before creating MySQL, we'll have to create a corresponding `secret` and `ConfigMap` first. To create a `secret`, we need to generate `base64` encrypted data: **// generate base64 secret for MYSQL_PASSWORD and MYSQL_ROOT_PASSWORD**
 # echo -n "pass" | base64
cGFzcw==
 # echo -n "mysqlpass" | base64
bXlzcWxwYXNz

Then, we're able to create the secret: **# cat secret.yaml**
apiVersion: v1
kind: Secret
metadata:
name: mysql-user
type: Opaque
data:
password: cGFzcw==

 # MYSQL_ROOT_PASSWORD
apiVersion: v1
kind: Secret
metadata:
name: mysql-root
type: Opaque
data:
password: bXlzcWxwYXNz
// create mysql secret
 # kubectl create -f secret.yaml --record
secret "mysql-user" created
secret "mysql-root" created

After that, we come to our ConfigMap. Here, we put the database user and the database name as an example: **# cat config.yaml**
kind: ConfigMap
apiVersion: v1
metadata:
name: mysql-config
data:
user: user
database: db
// create ConfigMap
 # kubectl create -f config.yaml --record
configmap "mysql-config" created

It's then time to launch MySQL and its service: **# cat mysql.yaml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: lmysql
spec:
replicas: 1
selector:
matchLabels:
tier: database
version: "5.7"
template:
metadata:
labels:
tier: database
version: "5.7"
spec:
containers:
- name: lmysql
image: mysql:5.7
volumeMounts:
- mountPath: /var/lib/mysql
name: mysql-vol
ports:
- containerPort: 3306
env:
- name: MYSQL_ROOT_PASSWORD
valueFrom:
secretKeyRef:
name: mysql-root
key: password
- name: MYSQL_DATABASE
valueFrom:
configMapKeyRef:
name: mysql-config
key: database
- name: MYSQL_USER

valueFrom:
configMapKeyRef:
name: mysql-config
key: user
- name: MYSQL_PASSWORD
valueFrom:
secretKeyRef:
name: mysql-user
key: password
volumes:
- name: mysql-vol
hostPath:
path: /mysql/data
minReadySeconds: 10
strategy:
type: RollingUpdate
rollingUpdate:
maxSurge: 1
maxUnavailable: 1
---
kind: Service
apiVersion: v1
metadata:
name: lmysql-service
spec:
selector:
tier: database
ports:
- protocol: TCP
port: 3306
targetPort: 3306
name: tcp3306

We can put more than one spec into a file by adding three dashes as separation. Here, we mount `hostPath /mysql/data` into pods with the path `/var/lib/mysql`. In the environment section, we use the secret and ConfigMap syntax with `secretKeyRef` and `configMapKeyRef`.

After creating MySQL, Redis would be the next best candidate, since other services are dependent on it but it doesn't have any prerequisites:

```
// create Redis deployment
      # cat redis.yaml
     apiVersion: apps/v1
       kind: Deployment
           metadata:
          name: lcredis
             spec:
           replicas: 1
            selector:
           matchLabels:
             tier: cache
           version: "3.0"
            template:
             metadata:
               labels:
              tier: cache
            version: "3.0"
               spec:
             containers:
           - name: lcredis
            image: redis:3.0
                 ports:
       - containerPort: 6379
      minReadySeconds: 1
          strategy:
       type: RollingUpdate
          rollingUpdate:
             maxSurge: 1
        maxUnavailable: 1
             ---
        kind: Service
       apiVersion: v1
          metadata:
     name: lcredis-service
           spec:
          selector:
          tier: cache
            ports:
        - protocol: TCP
            port: 6379
```

```
                                        targetPort: 6379
                                          name: tcp6379

                                // create redis deployements and service
                                     # kubectl create -f redis.yaml
                                       deployment "lcredis" created
                                    service "lcredis-service" created
```

It would then be a good time to start the kiosk: **# cat kiosk-example.yaml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: kiosk-example
spec:
replicas: 5
selector:
matchLabels:
tier: frontend
version: "3"
template:
metadata:
labels:
tier: frontend
version: "3"
annotations:
maintainer: cywu
spec:
containers:
- name: kiosk-example
image: devopswithkubernetes/kiosk-example
ports:
- containerPort: 5000
env:
- name: REDIS_HOST
value: lcredis-service.default
minReadySeconds: 5
strategy:
type: RollingUpdate
rollingUpdate:
maxSurge: 1
maxUnavailable: 1
---
kind: Service
apiVersion: v1
metadata:
name: kiosk-service
spec:
type: NodePort
selector:
tier: frontend
ports:
- protocol: TCP
port: 80
targetPort: 5000
name: tcp5000
// launch the spec
 # kubectl create -f kiosk-example.yaml
deployment "kiosk-example" created
service "kiosk-service" created

Here, we expose `lcredis-service.default` to environment variables to kiosk pods. This is the DNS name that kube-dns creates for `Service` objects (referred to as Services in this chapter). Hence, the kiosk can access the Redis host via environment variables.

In the end, we'll create a recorder. This doesn't expose any interface to others, so it doesn't need a `Service` object: **# cat recorder-example.yaml**
apiVersion: apps/v1
kind: Deployment

metadata:
name: recorder-example
spec:
replicas: 3
selector:
matchLabels:
tier: backend
version: "3"
template:
metadata:
labels:
tier: backend
version: "3"
annotations:
maintainer: cywu
spec:
containers:
- name: recorder-example
image: devopswithkubernetes/recorder-example
env:
- name: REDIS_HOST
value: lcredis-service.default
- name: MYSQL_HOST
value: lmysql-service.default
- name: MYSQL_USER
value: root
- name: MYSQL_ROOT_PASSWORD
valueFrom:
secretKeyRef:
name: mysql-root
key: password
minReadySeconds: 3
strategy:
type: RollingUpdate
rollingUpdate:
maxSurge: 1
maxUnavailable: 1
// create recorder deployment
 # kubectl create -f recorder-example.yaml
deployment "recorder-example" created

The recorder needs to access both Redis and MySQL. It uses root credentials that are injected via a secret. Both endpoints for Redis and MySQL are accessed via a service DNS name, `<service_name>.<namespace>`.

We could then check the `Deployment` objects:

```
                                  // check deployment details
                                    # kubectl get deployments
NAME               DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
 kiosk-example      5         5         5            5           1h
 lcredis            1         1         1            1           1h
 lmysql             1         1         1            1           1h
recorder-example   3         3         3            3           1h
```

As expected, we have four `Deployment` objects with a different desired count of pods.

As we expose the kiosk as a NodePort, we should be able to access its service endpoint and see whether it works properly. Assume we have a node, the IP of which is `192.168.99.100`, and the NodePort that Kubernetes allocates is `30520`.

If you're using minikube, `minikube service [-n NAMESPACE] [--url] NAME` could help you access service NodePort via your default browser: ***// open kiosk console***

 # minikube service kiosk-service

Opening kubernetes service default/kiosk-service in default browser...

This will allow us to find out the IP and the port.

We could then create and get a ticket using `POST` and `GET /tickets`: **// post ticket**
 # curl -XPOST -F 'value=100' http://192.168.99.100:30520/tickets

SUCCESS
// get ticket
 # curl -XGET http://192.168.99.100:30520/tickets
100

Summary

In this chapter, we learned the basic concepts of Kubernetes. We learned that a Kubernetes master has kube-apiserver to handle requests and controller managers are the control center of Kubernetes. These ensure our desired container amount is fulfilled, they control the endpoint to associate pods and services, and they control API access tokens. We also have Kubernetes nodes, which are the workers to host the containers, receive the information from the master, and route the traffic based on the configuration.

We then used minikube to demonstrate basic Kubernetes objects, including pods, ReplicaSets, Deployments, Services, secrets, and ConfigMaps. Finally, we demonstrated how to combine all of the concepts we've learned into our kiosk application.

As we mentioned previously, the data inside containers will disappear when a container is gone. Therefore, volume is extremely important to persist the data in the container world. In `Chapter 4`, *Managing Stateful Workloads*, we'll be learning how volume works and how to use persistent volume.

Managing Stateful Workloads

In `Chapter 3`, *Getting Started with Kubernetes*, we introduced the basic functions of Kubernetes. Once you start to deploy containers with Kubernetes, you'll need to consider the application's data life cycle and CPU/memory resource management.

In this chapter, we will discuss the following topics:

- How a container behaves with volumes
- Introducing Kubernetes' volume functionalities
- Best practices and pitfalls of Kubernetes' persistent volume
- Submitting a short-lived application as a Kubernetes Job

Kubernetes volume management

Kubernetes and Docker use a local disk by default. The Docker application may store and load any data onto the disk, for example, log data, temporary files, and application data. As long as the host has enough space and the application has the necessary permission, the data will exist as long as a container exists. In other words, when a container terminates, exits, crashes, or is reassigned to another host, the data will be lost.

//run CentOS Container
$ docker run -it centos
# ls
anaconda-post.log dev home lib64 media opt root sbin sys usr
bin etc lib lost+found mnt proc run srv tmp var

//create one file (/I_WAS_HERE) at root directory
touch /I_WAS_HERE
ls /
I_WAS_HERE bin etc lib lost+found mnt proc run srv tmp
var
anaconda-post.log dev home lib64 media opt root sbin sys usr

//Exit container
exit
exit

//re-run CentOS Container
docker run -it centos

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.19.1">//previous file (/I_WAS_HERE) was disappeared
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.20.1"># ls /
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.21.1">anaconda-post.log dev home lib64 media opt root sbin sys
usr
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.22.1">bin
etc lib lost+found mnt proc run srv tmp var

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.25.1">//there are 2 pod on the same Node
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.26.1">$
kubectl get pods
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.27.1">NAME READY STATUS RESTARTS
AGE
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.28.1">Besteffort 1/1 Running 0 1h
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.29.1">guaranteed 1/1 Running 0 1h

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.30.1">//when application consumes a lot of memory, one Pod has been
killed
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.31.1">$ kubectl get pods
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.32.1">NAME READY STATUS RESTARTS AGE
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.33.1">Besteffort 0/1 Error 0 1h
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.34.1">guaranteed 1/1 Running 0 1h

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.35.1">//clashed Pod is restarting
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.36.1">$
kubectl get pods
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.37.1">NAME READY STATUS RESTARTS
AGE
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.38.1">Besteffort 0/1 CrashLoopBackOff 0
1h
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.39.1">guaranteed 1/1 Running 0 1h

<span

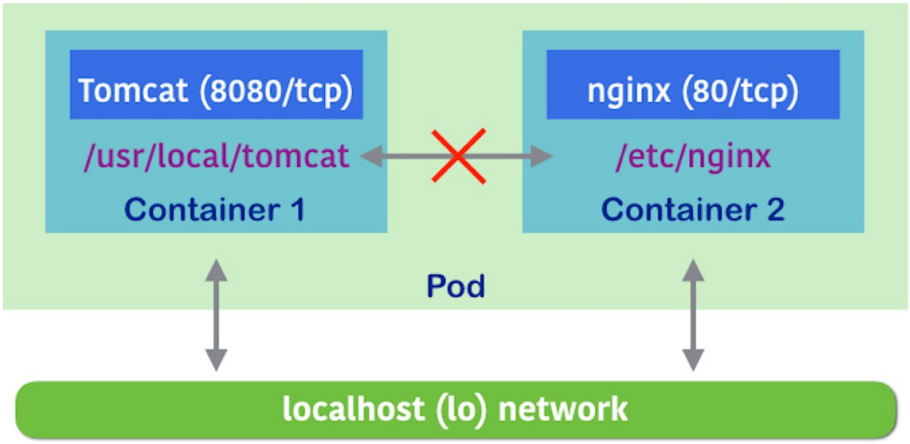
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.40.1">//few moment later, Pod has been restarted

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.41.1">$ kubectl get pods
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.42.1">NAME READY STATUS RESTARTS AGE
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.43.1">Besteffort 1/1 Running 1 1h
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.44.1">guaranteed 1/1 Running 0 1h

Sharing volume between containers within a pod

`Chapter 3`, *Getting Started with Kubernetes*, stated that multiple containers within the same Kubernetes pod can share the same pod IP address, network port, and IPC. Therefore, applications can communicate with each other through a localhost network. However, the filesystem is segregated.

The following diagram shows that **Tomcat** and **nginx** are in the same pod. Those applications can communicate with each other via localhost. However, they can't access each other's `config` files:



Some applications won't affect these scenarios and behavior, but some applications may have some use cases that require them to use a shared directory or file. Consequently, developers and Kubernetes administrators need to be aware of the different types of stateless and stateful applications.

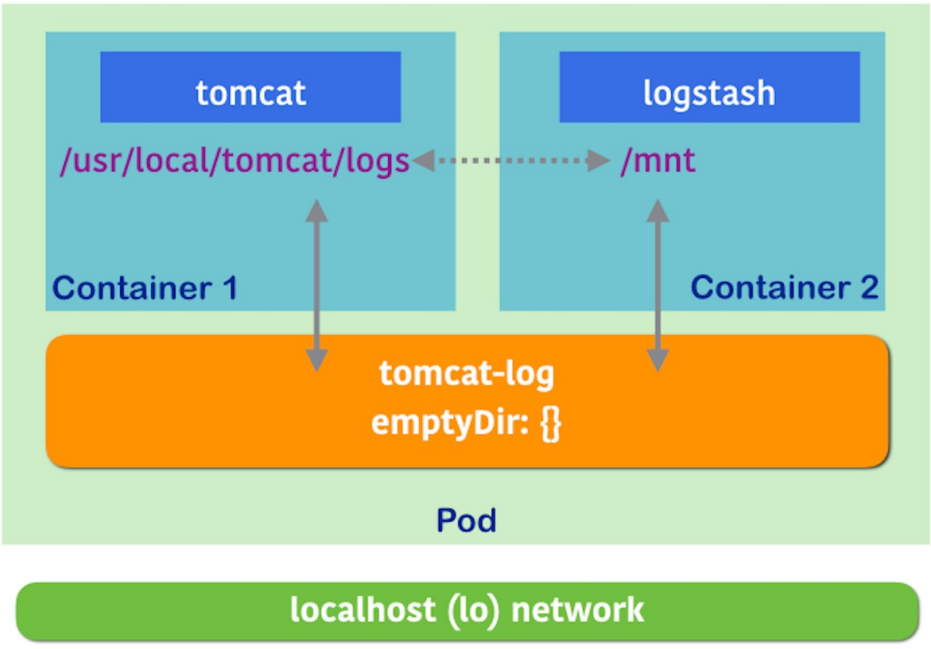
Stateless and stateful applications

Stateless applications don't need to preserve the application or user data on the disk volume. Although stateless applications may write the data to the filesystem while a container exists, it is not important in terms of the application's life cycle.

For example, the `tomcat` container runs some web applications. It also writes an application log under `/usr/local/tomcat/logs/`, but it won't be affected if it loses a `log` file.

But what if you need to persist an application log for analysis or auditing? In this scenario, Tomcat can still be stateless but share the `/usr/local/tomcat/logs` volume with another container such as Logstash (`https://www.elastic.co/products/logstash`). Logstash will then send a log to the chosen analytic store, such as Elasticsearch (`https://www.elastic.co/products/elasticsearch`).

In this case, the `tomcat` container and `logstash` container must be in the same Kubernetes pod and share the `/usr/local/tomcat/logs` volume, as follows:



The preceding diagram shows how Tomcat and Logstash can share the `log` file

using the Kubernetes `emptyDir` volume (https://kubernetes.io/docs/concepts/storage/volumes/#emptydir).

Tomcat and Logstash didn't use the network via localhost, but they did share the filesystem between `/usr/local/tomcat/logs` from the Tomcat container and `/mnt` from the `logstash` container, through Kubernetes' `emptyDir` volume: **$ cat tomcat-logstash.yaml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: tomcat
spec:
replicas: 1
selector:
matchLabels:
run: tomcat
template:
metadata:
labels:
run: tomcat
spec:
containers:
- image: tomcat
name: tomcat
ports:
- containerPort: 8080
env:
- name: UMASK
value: "0022"
volumeMounts:
- mountPath: /usr/local/tomcat/logs
name: tomcat-log
- image: logstash
name: logstash
args: ["-e input { file { path => \"/mnt/localhost_access_log.*\" } } output { stdout { codec => rubydebug } elasticsearch { hosts => [\"http://elasticsearch-svc.default.svc.cluster.local:9200\"] } }"]

volumeMounts:
- mountPath: /mnt
name: tomcat-log
volumes:
- name: tomcat-log
emptyDir: {}

Let's create `tomcat` and `logstash` pod, and then see whether Logstash can see the Tomcat application log under `/mnt`: **//create Pod**
$ kubectl create -f tomcat-logstash.yaml
deployment.apps/tomcat created

//check Pod name
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
tomcat-7d99999565-6pm64 2/2 Running 0 1m

//connect to logstash container to see /mnt directory
$ kubectl exec -it tomcat-7d99999565-6pm64 -c logstash /bin/bash
root@tomcat-7d99999565-6pm64:/# ls /mnt
catalina.2018-09-20.log localhost.2018-09-20.log manager.2018-09-20.log
host-manager.2018-09-20.log localhost_access_log.2018-09-20.txt

In this scenario, Elasticsearch must be stateful in the final destination, meaning that it uses a persistent volume. The Elasticsearch container must preserve the data even if the container is restarted. In addition, you do not need to configure the Elasticsearch container within the same pod as Tomcat/Logstash. Because Elasticsearch should be a centralized log datastore, it can be separated from the Tomcat/Logstash pod and scaled independently.

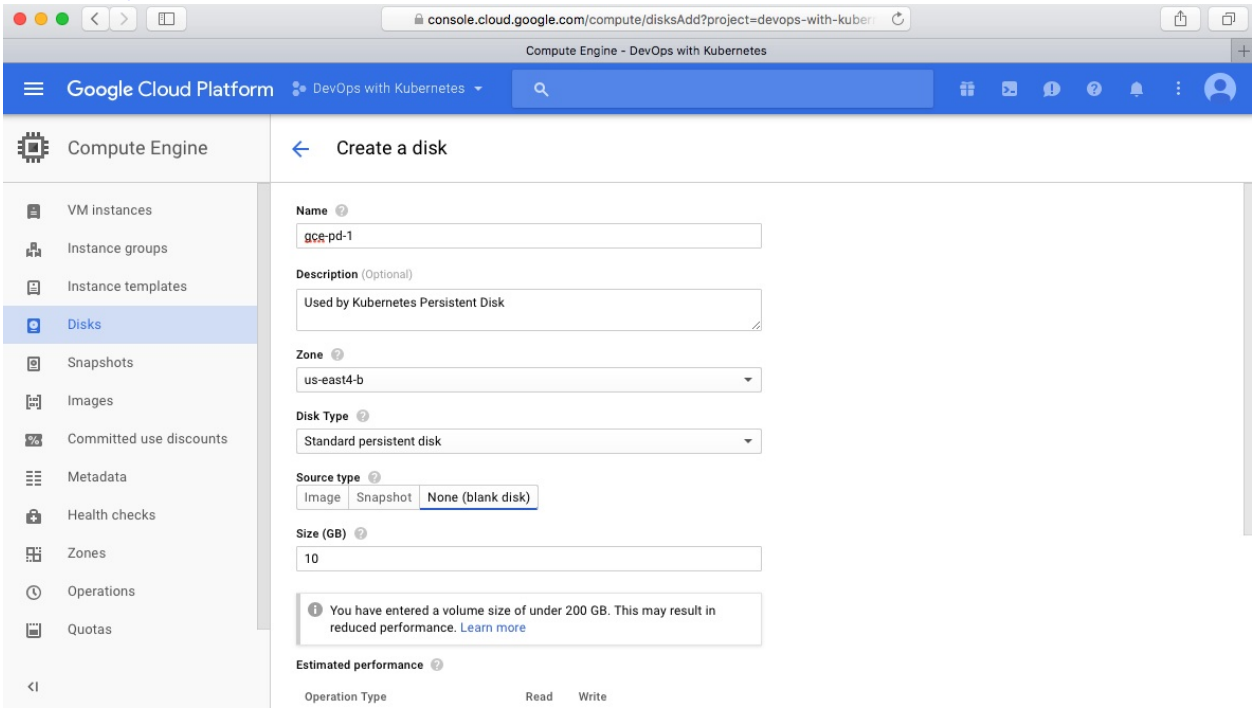
Once you determine that your application needs a persistent volume, there are some different types of volumes and different ways to manage persistent volumes to look at.

Kubernetes' persistent volume and dynamic provisioning

Kubernetes supports a variety of persistent volumes, for example, public cloud storage such as AWS EBS and Google persistent disks. It also supports network (distributed) filesystems such as NFS, GlusterFS, and Ceph. In addition, it can also support a block device such as iSCSI and Fibre Channel. Based on the environment and infrastructure, a Kubernetes administrator can choose the best matching types of persistent volume.

The following example is using a GCP persistent disk as a persistent volume. The first step is to create a GCP persistent disk and name it `gce-pd-1`.

If you use AWS EBS, Google persistent disk, or Azure disk storage, the Kubernetes node must be in the same cloud platform. In addition, Kubernetes has a limit on maximum volumes per node. Please look at the Kubernetes documentation at `https://kubernetes.io/docs/concepts/storage/storage-limits/`.



Then, specify the name `gce-pd-1` in the `Deployment` definition: **$ cat tomcat-pv.yml**
apiVersion: apps/v1

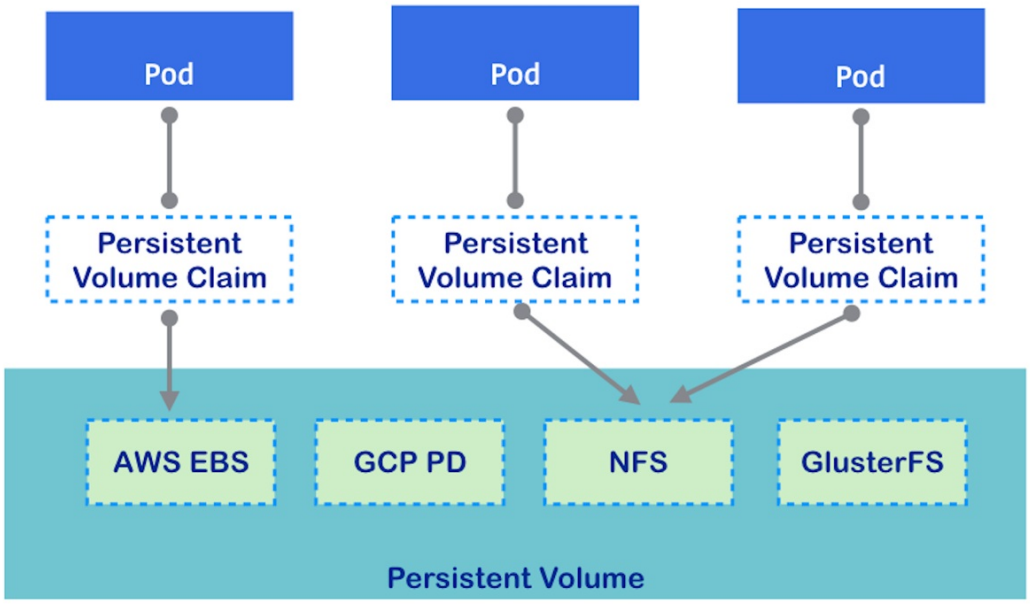
kind: Deployment
metadata:
name: tomcat
spec:
replicas: 1
selector:
matchLabels:
run: tomcat
template:
metadata:
labels:
run: tomcat
spec:
containers:
- image: tomcat
name: tomcat
ports:
- containerPort: 8080
volumeMounts:
- mountPath: /usr/local/tomcat/logs
name: tomcat-log
volumes:
- name: tomcat-log
gcePersistentDisk:
pdName: gce-pd-1
fsType: ext4

This will mount the persistent disk from the GCE persistent disk to `/usr/local/tomcat/logs`, which can persist Tomcat application logs.

Abstracting the volume layer with a persistent volume claim

Directly specifying a persistent volume in a configuration file makes a tight couple with a particular infrastructure. For the preceding example (`tomcat-log` volume), `pdName` is `gce-pd-1` and Volume type is `gcePersistentDisk`. From a container management point of view, the pod definition shouldn't be locked into the specific environment because the infrastructure could be different based on the environment. The ideal pod definition should be flexible or abstract the actual infrastructure and specify only volume name and mount point.

Consequently, Kubernetes provides an abstraction layer that associates the pod with the persistent volume, which is called the **Persistent Volume Claim** (**PVC**). This allows us to decouple from the infrastructure. The Kubernetes administrator just needs to pre-allocate the size of the persistent volume in advance. Then Kubernetes will bind the persistent volume and the PVC as follows:



The following example is a definition of a pod that uses a PVC; let's reuse the previous example (`gce-pd-1`) to register with Kubernetes first: **$ cat pv-gce-pd-1.yml**
apiVersion: "v1"

kind: "PersistentVolume"
metadata:
name: pv-1
spec:
storageClassName: "my-10g-pv-1"
capacity:
storage: "10Gi"
accessModes:
- "ReadWriteOnce"
gcePersistentDisk:
fsType: "ext4"
pdName: "gce-pd-1"

$ kubectl create -f pv-gce-pd-1.yml
persistentvolume/pv-1 created

$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pv-1 10Gi RWO Retain Available my-10g-pv-1 11s

Note that we assign `storageClassName` as `my-10g-pv-1`, as an identifier that the PVC can bind to by specifying the same name.

Next, we create a PVC that associates with the persistent volume (`pv-1`).

*The `storageClassName` parameter lets Kubernetes use static provisioning. This is because some Kubernetes environments, such as **Google Container Engine** (**GKE**), are already set up with dynamic provisioning. If we don't specify `storageClassName`, Kubernetes will allocate a new `PersistentVolume` and then bind to `PersistentVolumeClaim`.*

```
//create PVC specify storageClassName as "my-10g-pv-1"
$ cat pvc-1.yml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-1
spec:
 storageClassName: "my-10g-pv-1"
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 10Gi
```

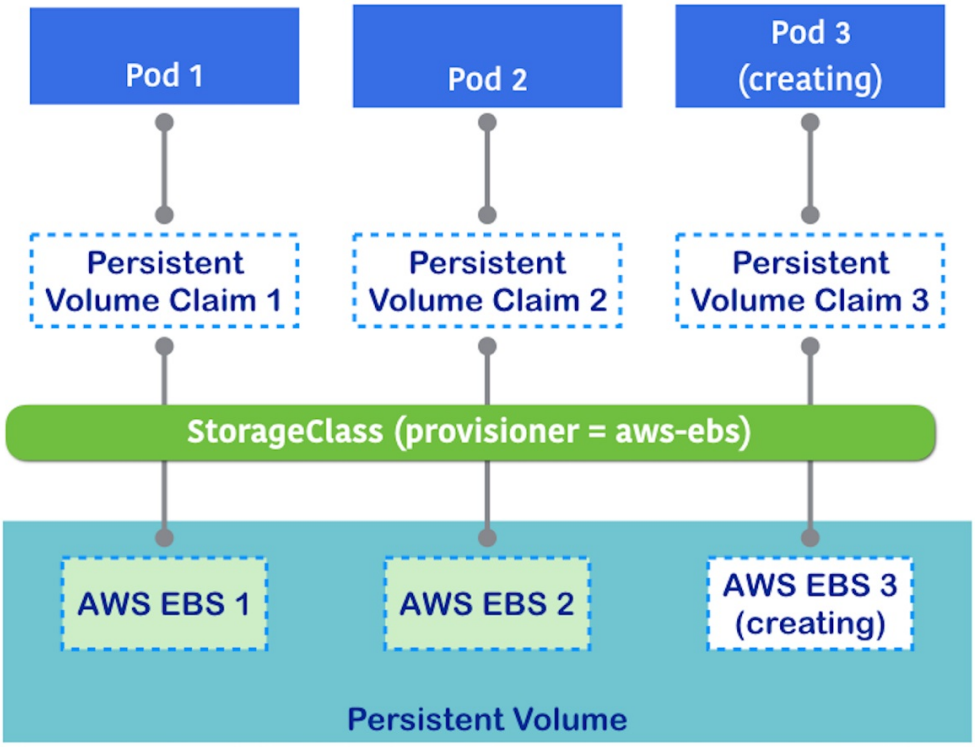
```
$ kubectl create -f pvc-1.yml
persistentvolumeclaim/pvc-1 created


//check PVC status is "Bound"
$ kubectl get pvc
NAME STATUS VOLUME CAPACITY  ACCESS MODES  STORAGECLASS  AGE
pvc-1  Bound  pv-1 10Gi  RWO my-10g-pv-1 7s


//check PV status is also "Bound"
$ kubectl get pv
NAME CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS CLAIM  STORAGECLASS  REASON AGE
pv-1 10Gi  RWO Retain  Bound  default/pvc-1  my-10g-pv-1 2m
```

Now the `tomcat` setting has been decoupled from the GCE persistent volume, and bound to the abstracted volume, `pvc-1`: **$ cat tomcat-pvc.yml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: tomcat
spec:
replicas: 1
selector:
matchLabels:
run: tomcat
template:
metadata:
labels:
run: tomcat
spec:
containers:
- image: tomcat
name: tomcat
ports:
- containerPort: 8080
volumeMounts:
- mountPath: /usr/local/tomcat/logs
name: tomcat-log
volumes:
- name: tomcat-log
persistentVolumeClaim:
claimName: "pvc-1"

Dynamic provisioning and StorageClass

PVC a degree of flexibility for persistent volume management. However, pre-allocating some persistent volume pools might not be cost-efficient, especially in a public cloud.

Kubernetes also assists in this kind of situation by supporting dynamic provisioning for persistent volumes. The Kubernetes administrator defines the provisioner of the persistent volume, which is called `StorageClass`. Then, the PVC asks `StorageClass` to dynamically allocate a persistent volume, and then associates it with the PVC as follows:



In the following example, AWS EBS is used as the `StorageClass`. When creating the PVC, the `StorageClass` dynamically creates an EBS then registers it as **Persistent Volume** (**PV**), and then attaches it to the PVC: **$ cat storageclass-aws.yml**
kind: StorageClass

apiVersion: storage.k8s.io/v1
metadata:
name: aws-sc
provisioner: kubernetes.io/aws-ebs
parameters:
type: gp2

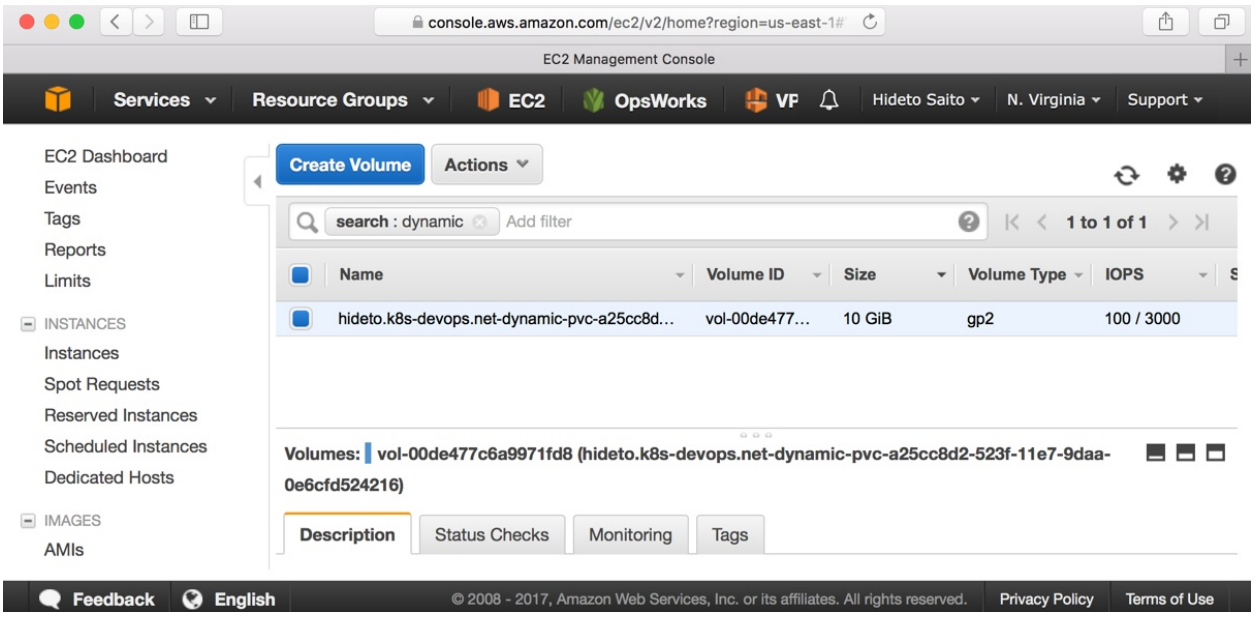
$ kubectl create -f storageclass-aws.yml
storageclass "aws-sc" created

$ kubectl get storageclass
NAME. TYPE
aws-sc. kubernetes.io/aws-ebs

Once `StorageClass` has been successfully created, then, create a PVC without PV, but specify the `StorageClass` name. In this example, this would be `aws-sc`, as follows:

```
$ cat pvc-aws.yml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-aws-1
spec:
 storageClassName: "aws-sc"
 accessModes:
 - ReadWriteOnce
 resources:
 requests:
 storage: 10Gi


$ kubectl create -f pvc-aws.yml
persistentvolumeclaim/pvc-aws-1 created


$ kubectl get pv
NAME  CAPACITY  ACCESS MODES  RECLAIM POLICY  STATUS CLAIM  STORAGECLASS  REASON AGE
pvc-03557eb8-bc8b-11e8-994f-42010a800085  10Gi  RWO Delete  Bound  default/pvc-aws-1  aw
```

The following screenshot shows the EBS after submitting to StorageClass to create a PVC. AWS console shows a new EBS which is created by StorageClass:



Note that managed Kubernetes services such as Amazon EKS (`https://aws.amazon.com/eks/`), Google Kubernetes Engine (`https://cloud.google.com/container-engine/`), and Azure Kubernetes Service (`https://azure.microsoft.com/en-us/services/kubernetes-service/`) create `StorageClass` by default. For example, Google Kubernetes Engine sets up a default storage class as a Google Cloud persistent disk. For more information, please refer to `Chapter 10`, *Kubernetes on AWS*, `Chapter 11`, *Kubernetes on GCP*, and `Chapter 12`, *Kubernetes on Azure*: //**default Storage Class on GKE $ kubectl get sc NAME TYPE standard (default) kubernetes.io/gce-pd**

Problems with ephemeral and persistent volume settings

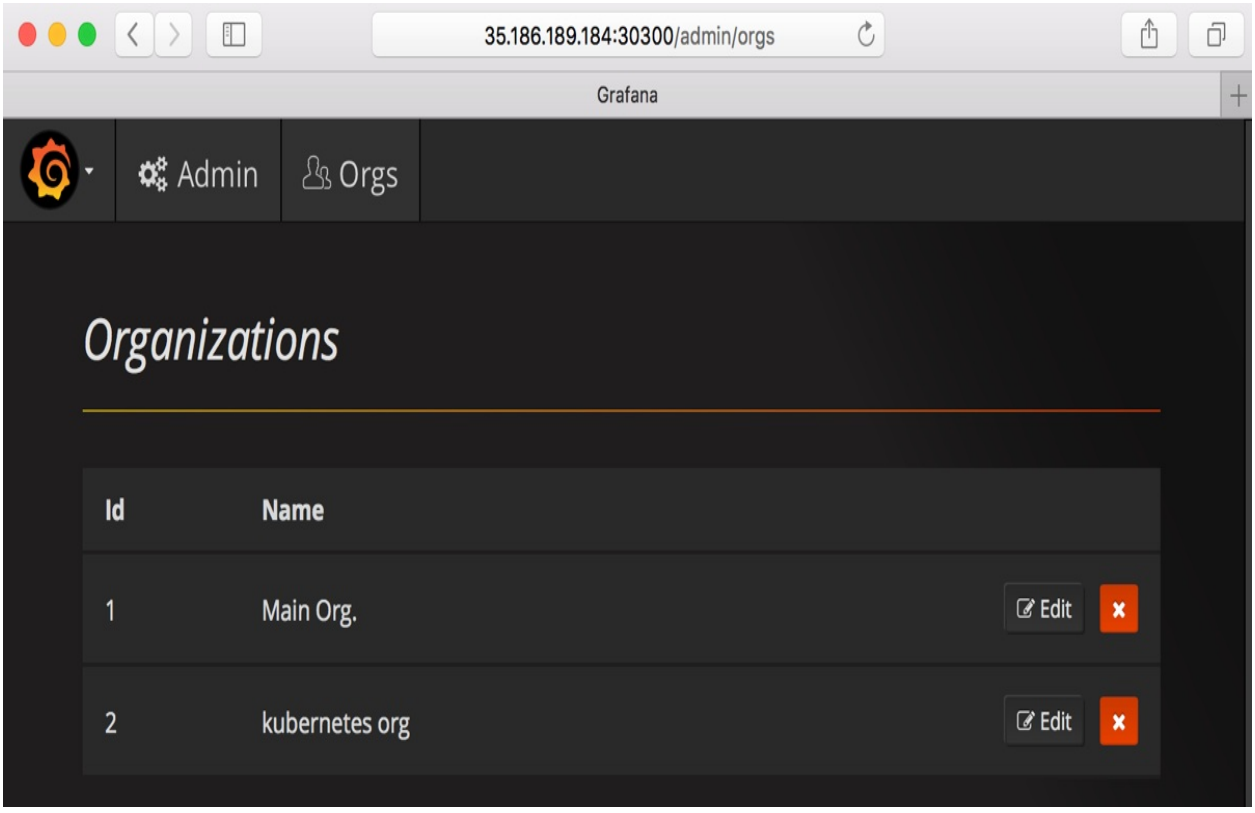
You may determine your application as stateless because the `datastore` function is handled by another pod or system. However, there are some pitfalls in that sometimes; applications actually store important files that you aren't aware of. For example, Grafana (`https://grafana.com/grafana`) connects time series datasources such as Graphite (`https://graphiteapp.org`) and InfluxDB (`https://www.influxdata.com/time-series-database/`), so people may misunderstand that Grafana is a stateless application.

Actually, Grafana itself also uses databases to store user, organization, and dashboard metadata. By default, Grafana uses SQLite3 components and stores the database as `/var/lib/grafana/grafana.db`. Therefore, when a container is restarted, the Grafana settings will all be reset.

The following example demonstrates how Grafana behaves with an ephemeral volume: **$ cat grafana.yml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: grafana
spec:
replicas: 1
selector:
matchLabels:
run: grafana
template:
metadata:
labels:
run: grafana
spec:
containers:
- image: grafana/grafana

name: grafana
ports:
- containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
name: grafana
spec:
ports:
- protocol: TCP
port: 3000
nodePort: 30300
type: NodePort
selector:
run: grafana

Next, navigate to the Grafana web console to create the Grafana `Organizations` named `kubernetes org`, as follows:



Then, look at the `Grafana` directory. Here, there is a database file (`/var/lib/grafana/grafana.db`) with a timestamp that has been updated after creating a Grafana `Organizations`: **$ kubectl get pods**
NAME READY STATUS RESTARTS AGE
grafana-6bf966d7b-7lh89 1/1 Running 0 3m

//access to Grafana container
$ kubectl exec -it grafana-6bf966d7b-7lh89 /bin/bash
grafana@grafana-6bf966d7b-7lh89:/$ ls -l /var/lib/grafana/
total 404
-rw-r--r-- 1 grafana grafana 401408 Sep 20 03:30 grafana.db
drwxrwxrwx 2 grafana grafana 4096 Sep 7 14:59 plugins
drwx------ 4 grafana grafana 4096 Sep 20 03:30 sessions

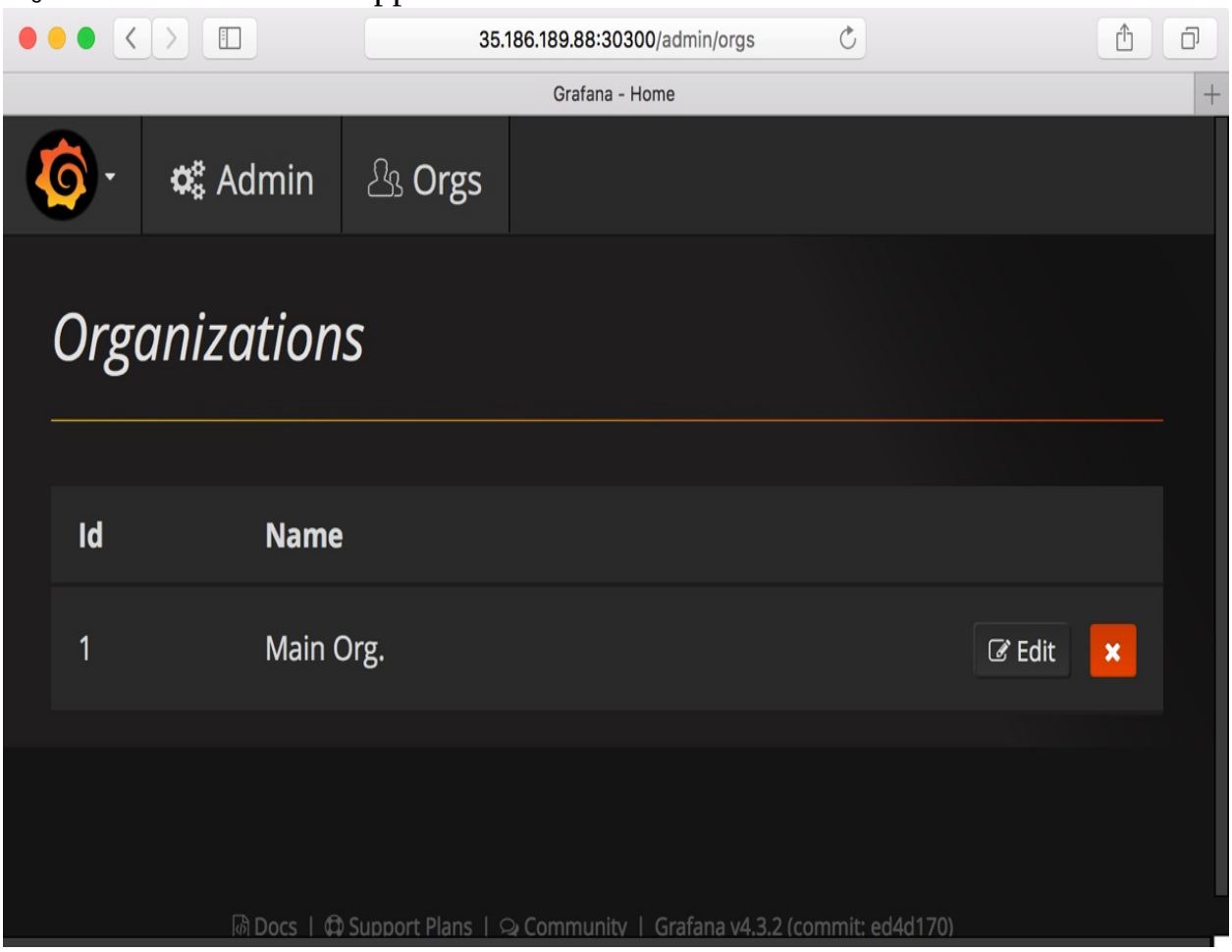
When the pod is deleted, the `Deployment` will start a new pod and check whether a Grafana `Organizations` exists or not: **grafana@grafana-6bf966d7b-7lh89:/$ exit**
//delete grafana pod
$ kubectl delete pod grafana-6bf966d7b-7lh89
pod "grafana-6bf966d7b-7lh89" deleted

//Kubernetes Deployment made a new Pod
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
grafana-6bf966d7b-wpdmk 1/1 Running 0 9s

//contents has been recreated
$ kubectl exec -it grafana-6bf966d7b-wpdmk /bin/bash
grafana@grafana-6bf966d7b-wpdmk:/$ ls -l /var/lib/grafana
total 400
-rw-r--r-- 1 grafana grafana 401408 Sep 20 03:33 grafana.db
drwxrwxrwx 2 grafana grafana 4096 Sep 7 14:59 plugins

It looks like the `sessions` directory has disappeared and `grafana.db` has also been recreated by the Docker image again. If you access the web console, the Grafana

`organization` will also disappear:



How about just attaching a persistent volume to Grafana? You'll soon find that mounting a persistent volume on a pod controlled by a `Deployment` doesn't scale properly as every new pod attempts to mount the same persistent volume. In most cases, only the first pod can mount the persistent volume. Other pods will try to mount the volume, and they will give up and freeze if they can't. This happens if the persistent volume is capable of only RWO (read write once; only one pod can write).

In the following example, Grafana uses a persistent volume to mount `/var/lib/grafana`; however, it can't scale because the Google persistent disk is RWO: **$ cat grafana-pv.yml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: grafana

spec:
replicas: 1
selector:
matchLabels:
run: grafana
template:
metadata:
labels:
run: grafana
spec:
containers:
- image: grafana/grafana:3.1.1
name: grafana
ports:
- containerPort: 3000
volumeMounts:
- mountPath: /var/lib/grafana
name: grafana-data
volumes:
- name: grafana-data
gcePersistentDisk:
pdName: gce-pd-1
fsType: ext4

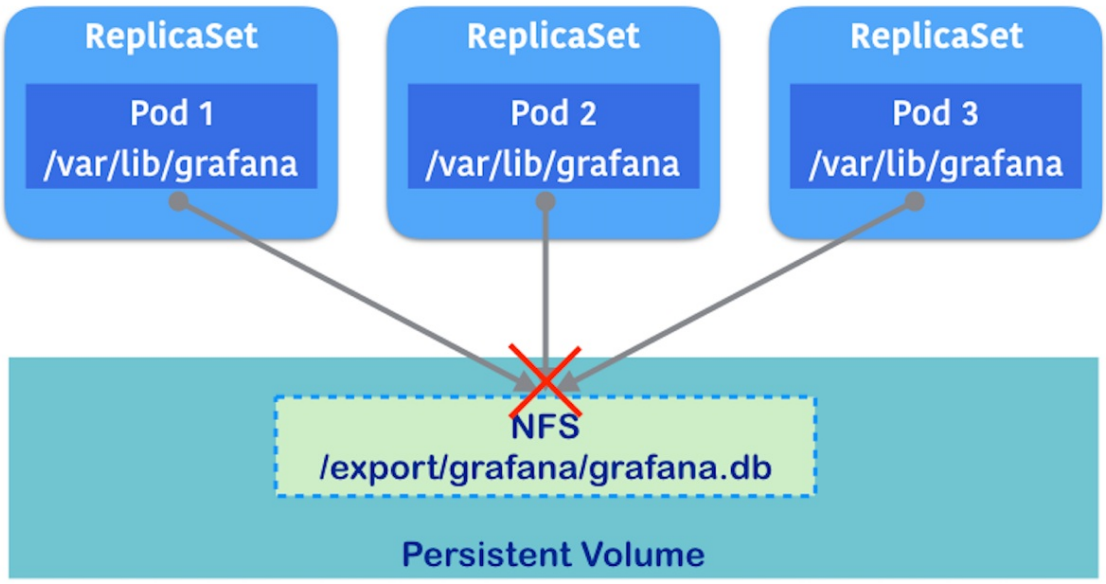
$ kubectl create -f grafana-pv.yml
deployment.apps/grafana created

$ kubectl get pods
NAME READY STATUS RESTARTS AGE
grafana-6cf5467c9d-nw6b7 1/1 Running 0 41s

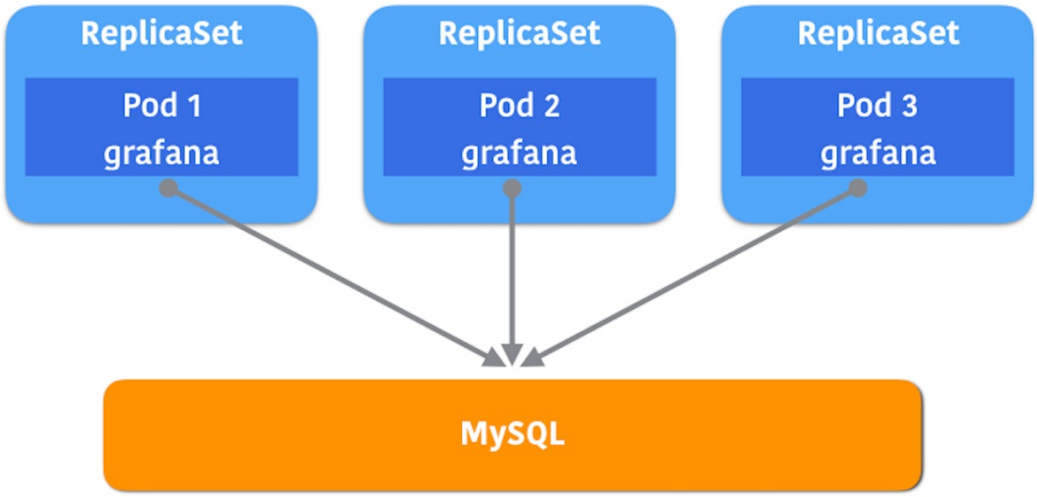
//can't scale up, becaues 3 Pod can't mount the same volume
$ kubectl scale deploy grafana --replicas=3
The Deployment "grafana" is invalid:
spec.template.spec.volumes[0].gcePersistentDisk.readOnly: Invalid value:
false: must be true for replicated pods > 1; GCE PD can only be mounted on

multiple machines if it is read-only

Even if the persistent volume has the RWX capability (read write many; many pods can mount to read and write simultaneously), such as NFS, it won't complain if multiple pods try to bind the same volume. However, we still need to consider whether multiple application instances can use the same folder/file or not. For example, if it replicates Grafana to two or more pods, it will be conflicted, with multiple Grafana instances that try to write to the same `/var/lib/grafana/grafana.db`, and then the data could be corrupted, as shown in the following diagram:



In this scenario, Grafana must use backend databases such as MySQL or PostgreSQL, instead of SQLite3, as follows. It allows multiple Grafana instances to read/write Grafana metadata properly:



Because RDBMS basically supports connecting with multiple application instances via a network, this scenario is perfectly suited to being used by multiple pods. Note that Grafana supports using RDBMS as a backend metadata store; however, not all applications support RDBMS.

For the Grafana configuration that uses MySQL/PostgreSQL, please see the online documentation at

http://docs.grafana.org/installation/configuration/#database.

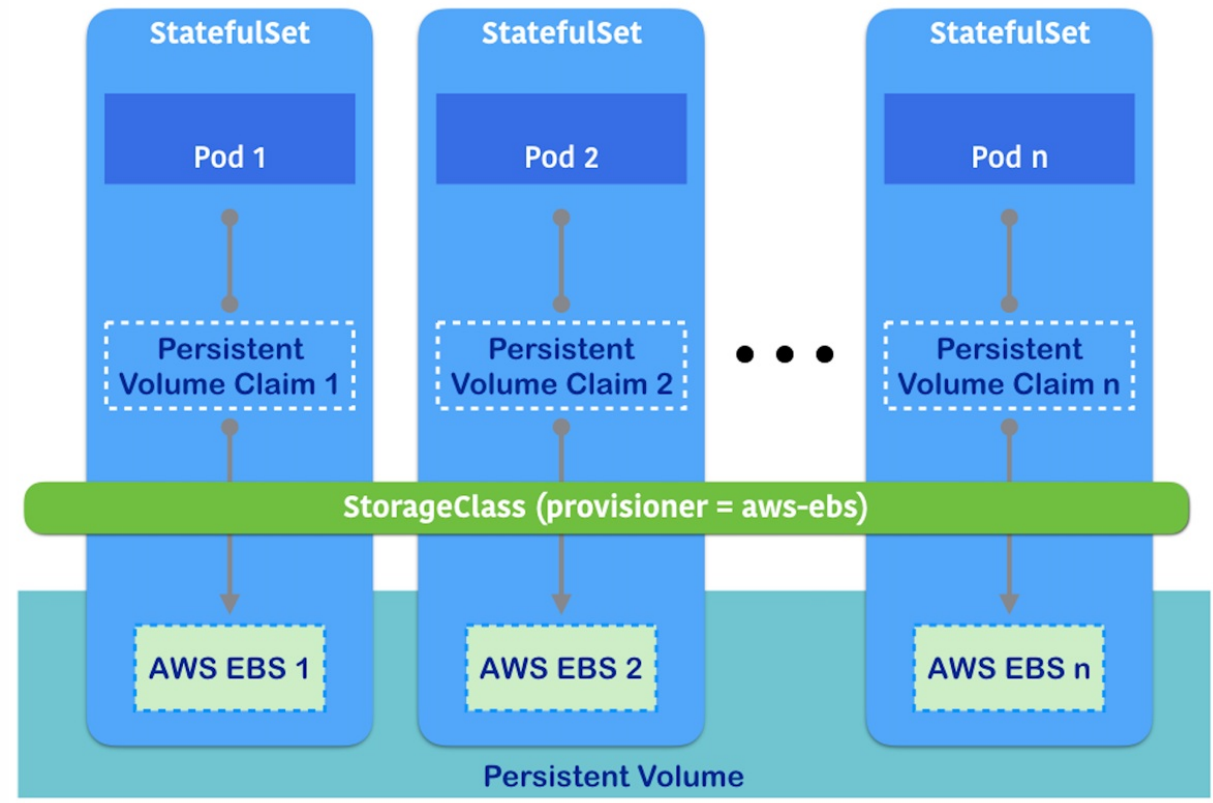
Therefore, the Kubernetes administrator needs to carefully monitor how an application behaves with volumes, and understand that in some use cases, just using a persistent volume may not help because of issues that might arise when scaling pods.

If multiple pods need to access the centralized volume, then consider using the database as previously shown, if applicable. On the other hand, if multiple pods need an individual volume, consider using `StatefulSet`.

Replicating pods with a persistent volume using StatefulSet

`StatefulSet` was introduced in Kubernetes 1.5; this consists of a bond between the pod and the persistent volume. When scaling a pod that increases or decreases, pod and persistent volume are created or deleted together.

In addition, the pod-creation process is serial. For example, when requesting Kubernetes to scale two additional `StatefulSet`s, Kubernetes creates **Persistent Volume Claim 1** and **Pod 1** first, and then creates **Persistent Volume Claim 2** and **Pod 2**, but not simultaneously. This helps the administrator if an application registers to a registry during the application bootstrap:

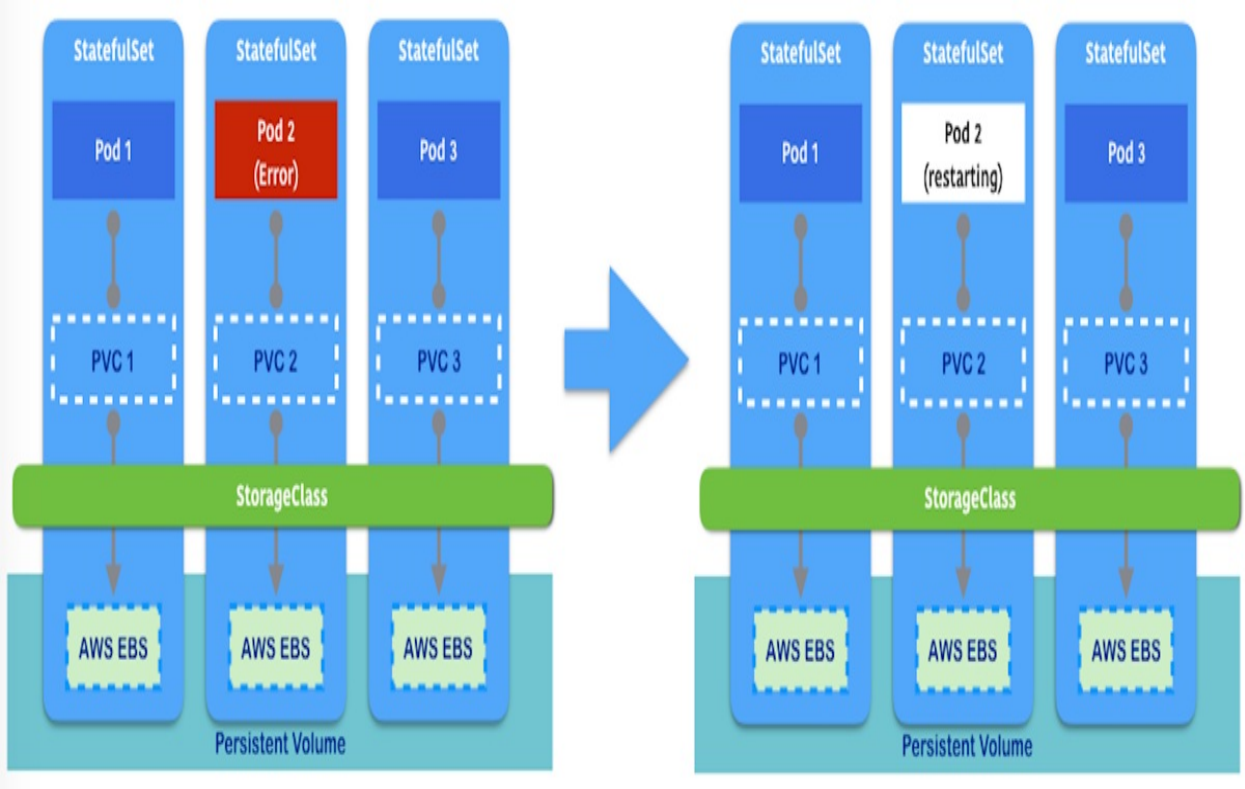


Even if one pod is dead, `StatefulSet` preserves the position of the pod (Kubernetes metadata, such as the pod name) and the persistent volume. Then it attempts to recreate a container that it reassigns to the same pod and mounts the same

persistent volume.

If you run the headless service with `StatefulSet`*, Kubernetes also assigns and preserves the FQDN for the pod as well.*
Headless services will be described in detail in `Chapter 6`*, Kubernetes Network.*

This helps to keep track of and maintain the number of pods/persistent volumes and the application remains online using the Kubernetes scheduler:



`StatefulSet` with a persistent volume requires dynamic provisioning and `StorageClass` because `StatefulSet` can be scalable. Kubernetes needs to know how to provision the persistent volume when adding more pods.

Submitting Jobs to Kubernetes

In general, the application is designed to be long-lived in the same way as the daemon process. A typical long-lived application opens the network port and keeps it running. It is required to keep running the application. If it fails, you will need to restart to recover the state. Therefore, using Kubernetes deployment is the best option for the long-lived application.

On the other hand, some applications are designed to be short-lived, such as a command script. It is expected to exit the application even if it is successful in order to finish the tasks. Therefore, a Kubernetes deployment is not the right choice, because a deployment tries to keep the process running.

No worries; Kubernetes also supports short-lived applications. You can submit a container as a **Job** or **Scheduled Job**, and Kubernetes will dispatch it to an appropriate node and execute your container.

Kubernetes supports several types of Jobs:

- Single Job
- Repeatable Job
- Parallel Job
- Scheduled Job

The last one is also called a **CronJob**. Kubernetes supports these different types of Jobs that are used differently to pods to utilize your resources.

Submitting a single Job to Kubernetes

A Job-like pod is suitable to run for batch programs such as collecting data, querying the database, generating a report, and so on. Although this is referred to as short-lived, it doesn't matter how long is spent on it. This may need to run for a few seconds, or perhaps a few days, in order to complete. It will eventually exit an application, which means it has an end state.

Kubernetes is capable of monitoring a short-lived application as a Job, and in the case of failure, Kubernetes will create a new pod for the Job that tries to accomplish your application to complete.

In order to submit a Job to Kubernetes, you need to write a Job template that specifies the pod configuration. The following example demonstrates how to check the `dpkg` installed in Ubuntu Linux: **$ cat job-dpkg.yaml**
apiVersion: batch/v1
kind: Job
metadata:
name: package-check
spec:
activeDeadlineSeconds: 60
template:
spec:
containers:
- name: package-check
image: ubuntu
command: ["dpkg-query", "-l"]
restartPolicy: Never

Job definition is similar to pod definition, but the important settings are `activeDeadlineSeconds` and `restartPolicy`. The `activeDeadlineSeconds` parameter sets the maximum timescale for the pod to run. If exceeded, the pod will be terminated. The `restartPolicy` parameter defines how Kubernetes behaves in the case of

failure. For example, when the pod is crashed if you specify `Never`, Kubernetes doesn't restart; if you specify `OnFailure`, Kubernetes attempts to resubmit the Job until successfully completed.

Use the `kubectl` command to submit a Job to see how Kubernetes manages the pod: **$ kubectl create -f job-dpkg.yaml**
job.batch/package-check created

Because this Job (the `dpkg-query -l` command) is short-lived, it will `exit()` eventually. Therefore, if the `dpkg-query` command completes gracefully, Kubernetes doesn't attempt to restart, even if no active pod exists. So, when you type `kubectl get pods`, the pod status will be `completed` after finishing: **$ kubectl get pods**
NAME READY STATUS RESTARTS AGE
package-check-7tfkt 0/1 Completed 0 6m

Although no active pod exists, you still have an opportunity to check the application log by typing the `kubectl logs` command as follows: **$ kubectl logs package-check-7tfkt**
Desired=Unknown/Install/Remove/Purge/Hold
| Status=Not/Inst/Conf-files/Unpacked/halF-conf/Half-inst/trig-aWait/Trig-pend
|/ Err?=(none)/Reinst-required (Status,Err: uppercase=bad)
||/ Name Version Architecture Description
+++-==========-==========-==========-==========
ii adduser 3.116ubuntu1 all add and remove users and groups
ii apt 1.6.3ubuntu0.1 amd64 commandline package manager
ii base-files 10.1ubuntu2.2 amd64 Debian base system miscellaneous files
ii base-passwd 3.5.44 amd64 Debian base system master password and group files
...

$ cat repeat-job.yaml
apiVersion: batch/v1
kind: Job
metadata:
 name: package-check-repeat
spec:
activeDeadlineSeconds: 60
 completions: 3
 template:
 spec:
 containers:
 - name: package-check-repeat
 image: ubuntu
 command: ["dpkg-query", "-l"]
 restartPolicy: Never

$ kubectl create -f repeat-job.yaml
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.21.1">job.batch/package-check-repeat created

<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.22.1">$
kubectl get pods
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.23.1">NAME
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.24.1">READY
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.25.1">STATUS
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.26.1">RESTARTS
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.27.1">AGE
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.28.1">package-check-7tfkt<span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.29.1">0/1 <span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.30.1">Completed <span class="Apple-
converted-space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.31.1">0<span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.32.1">52m
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.33.1">package-check-repeat-vl988 <span class="Apple-
converted-space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.34.1">0/1 <span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.35.1">Completed <span class="Apple-
converted-space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.36.1">0<span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.37.1">7s
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.38.1">package-check-repeat-flhz9 <span class="Apple-
converted-space"> <span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.39.1">0/1 <span class="Apple-converted-
space"> <span xmlns="http://www.w3.org/1999/xhtml"

class="koboSpan" id="kobo.40.1">Completed 0 4s
package-check-repeat-xbf8b 0/1 Completed 0 2s

As you can see, three pods are created to complete this Job. This is useful if you need to run your program repeatedly at particular times. However, as you may have noticed from the Age column in the preceding result, these pods ran sequentially one by one. In the preceding result, the ages are 7 seconds, 4 seconds, then 2 seconds. This means that the second Job was started after the first Job was completed, and the third Job was started after the second Job was completed.

If it is the case that a Job runs for a longer period (such as a few days), but if there is no correlation needs between the 1st, 2nd, and 3rd Job, then it does not make sense to run them sequentially. In this case, use a parallel Job.

Submitting a parallel Job

If your batch Job doesn't have a state or dependency between Jobs, you may consider submitting Jobs in parallel. To do so, similar to the `spec.completions` parameter, the Job template has a `spec.parallelism` parameter to specify how many Jobs you want to run in parallel: **$ cat parallel-job.yaml**
apiVersion: batch/v1
kind: Job
metadata:
name: package-check-parallel
spec:
activeDeadlineSeconds: 60
parallelism: 3
template:
spec:
containers:
- name: package-check-parallel
image: ubuntu
command: ["dpkg-query", "-l"]
restartPolicy: Never

//submit a parallel job
$ kubectl create -f parallel-job.yaml
job.batch/package-check-parallel created

//check the result
$ kubectl get pods
NAME READY STATUS RESTARTS AGE
package-check-7tfkt 0/1 Completed 0 1h
package-check-parallel-k8hpz 0/1 Completed 0 4s
package-check-parallel-m272g 0/1 Completed 0 4s
package-check-parallel-mc279 0/1 Completed 0 4s
package-check-repeat-flhz9 0/1 Completed 0 13m

package-check-repeat-vl988 0/1 Completed 0 13m
package-check-repeat-xbf8b 0/1 Completed 0 13m

As you see from the `AGE` column through the `kubectl get pods` command, it indicates that the three pods ran at the same time.

In this setting, Kubernetes can dispatch to an available node to run your application and that easily scales your Jobs. This is useful if you want to run something like a worker application to distribute a bunch of pods to different nodes.

Lastly, if you no longer need to check the Job's results anymore, delete the resource by using the `kubectl delete` command as follows:

```
$ kubectl get jobs
NAME  DESIRED  SUCCESSFUL  AGE
package-check 1  1 1h
package-check-parallel  <none> 3 9m
package-check-repeat  3  3 23m


// delete a job one by one
$ kubectl delete jobs package-check-parallel
job.batch "package-check-parallel" deleted

$ kubectl delete jobs package-check-repeat
job.batch "package-check-repeat" deleted

$ kubectl delete jobs package-check
job.batch "package-check" deleted


//there is no pod
$ kubectl get pods
No resources found.
```

Scheduling running a Job using CronJob

If you are familiar with **UNIX CronJob** or **Java Quartz** (`http://www.quartz-scheduler.org`), Kubernetes CronJob is a very straightforward tool that you can use to define a particular timing to run your Kubernetes Job repeatedly.

The scheduling format is very simple; it specifies the following five items:

- Minutes (0 to 59)
- Hours (0 to 23)
- Days of the month (1 to 31)
- Months (1 to 12)
- Days of the week (0: Sunday to 6: Saturday)

For example, if you only want to run your Job at 9:00 am on November 12th every year to send a birthday greeting to me, the schedule format would be `0 9 12 11 *`

You may also use a slash (`/`) to specify a step value. Running the previous Job example at five-minute intervals would have the following schedule format: */5 * * * *

The following template is using a `CronJob` to run the `package-check` command every five minutes: **$ cat cron-job.yaml**
apiVersion: batch/v1beta1
kind: CronJob
metadata:
name: package-check-schedule
spec:
schedule: "*/5 * * * *"
concurrencyPolicy: "Forbid"
jobTemplate:
spec:
template:

spec:
containers:
- name: package-check-schedule
image: ubuntu
command: ["dpkg-query", "-l"]
restartPolicy: Never

$ kubectl create -f cron-job.yaml
cronjob.batch/package-check-schedule created

You may notice that the template format is slightly different from the Job template here. However, there is one parameter we need to pay attention to: `spec.concurrencyPolicy`. With this, you can specify a behavior if the previous Job is not finished but the next Job's schedule is approaching. This determines how the next Job runs. You can set the following:

- **Allow**: Allow execution of the next Job
- **Forbid**: Skip execution of the next Job
- **Replace**: Delete the current Job, then execute the next Job

If you set `Allow`, there might be the potential risk of accumulating some unfinished Jobs in the Kubernetes cluster. Therefore, during the testing phase, you should set either `Forbid` or `Replace` to monitor Job execution and completion.

After a few moments, the Job will be triggered by your desired timing—in this case, every five minutes. You may then see the Job entry with the `kubectl get jobs` and `kubectl get pods` commands, as follows: **$ kubectl get jobs**
NAME DESIRED SUCCESSFUL AGE
package-check-schedule-1537169100 1 1 8m
package-check-schedule-1537169400 1 1 3m

$ kubectl get pods
NAME READY STATUS RESTARTS AGE
package-check-schedule-1537169100-mnhxw 0/1 Completed 0 8m
package-check-schedule-1537169400-wvbgp 0/1 Completed 0 3m

`CronJob` will remain until you delete it. This means that, every five minutes, `CronJob` will create a new Job entry and related pods will also keep getting created. This

will impact the consumption of Kubernetes resources. Therefore, by default, `CronJob` will keep up to three successful Jobs (with `spec.successfulJobsHistoryLimit`) and one failed Job (with `spec.failedJobsHistoryLimit`). You can change these parameters based on your requirements.

Overall, `CronJob` allows Jobs to automatically run in your application with the desired timing. You can utilize `CronJob` to run report-generation Jobs, daily or weekly batch Jobs, and so on.

Summary

In this chapter, we covered stateful applications that use persistent volumes. Compared to ephemeral volumes, they have some pitfalls when an application restarts or a pod scales. In addition, persistent volume management on Kubernetes has been enhanced to make it easier, as you can see from tools such as `StatefulSet` and dynamic provisioning.

Furthermore, Jobs and CronJobs are special utilities for pods. Compared to deployment/ReplicaSets, this has a desired number of pods running, which is against Job's ideal situation in which the pods should be deleted once they finish their tasks. This kind of short-lived application can be managed by Kubernetes as well.

In `Chapter 5`, *Cluster Administration and Extension*, we will discuss the cluster administration such as authentication, authorization, and admission control. We will also introduce the **Custom Resource Definition** (**CRD**) that how to control Kubernetes object by your own code.

Cluster Administration and Extension

In previous chapters, we familiarized ourselves with basic DevOps skills and Kubernetes objects. This included looking at many areas, such as how to containerize our application and deploy our containerized software into Kubernetes. It is now time to gain a deeper insight into Kubernetes cluster administration.

In this chapter, we'll learn about the following topics:

- Utilizing namespaces to set administrative boundaries
- Using kubeconfig to switch between multiple clusters
- Kubernetes authentication
- Kubernetes authorization
- Dynamic admission control
- Kubernetes **Custom Resources Definition** (**CRD**) and controllers

While minikube is a fairly simple environment, we will use the **Google Kubernetes Engine** (**GKE**) in this chapter. For cluster deployment in GKE, please refer to `Chapter 11`, *Kubernetes on GCP*.

Kubernetes namespaces

We already learned about Kubernetes namespaces in `Chapter 3`, *Getting Started with Kubernetes*, which are used to divide the resources from a cluster into multiple virtual clusters. Namespaces make each group share the same physical cluster with isolation. Each namespace provides the following:

- A scope of names; the object name in each namespace is unique
- Policies to ensure trusted authentication
- The ability to set up resource quotas for resource management

Now, let's learn how to use context to switch between different namespaces.

- context: cluster: gke_devops-with-kubernetes_us-central1-b_cluster user: gke_devops-with-kubernetes_us-central1-b_cluster name: gke_devops-with-kubernetes_us-central1-b_cluster

kubectl config current-context gke_devops-with-kubernetes_us-central1-b_cluster

To list all config information, including contexts, you could use the `kubectl config view` command. To check out what context is currently in use, use the `kubectl config get-contexts` command.

$ kubectl config set-context <context_name> --namespace=<namespace_name> --cluster=<cluster_name> --user=<user_name>

// create a context with my GKE cluster # kubectl config set-context chapter5 --namespace=chapter5 --cluster=gke_devops-with-kubernetes_us-central1-b_cluster --user=gke_devops-with-kubernetes_us-central1-b_cluster Context "chapter5" created.

kubectl config use-context chapter5

Switched to context "chapter5".

After the context is switched, every command we invoke via `kubectl` is under the `chapter5` context. There is no need to explicitly specify the namespace when listing your pods.

Kubeconfig

Kubeconfig is a file that you can use to switch multiple clusters by switching context. We can use `kubectl config view` to view the setting and the `kubectl config current-context` command to check the context you're currently using. The following is an example of a GCP cluster in a `kubeconfig` file: **# kubectl config view apiVersion: v1 clusters:**
- cluster:
certificate-authority-data: DATA+OMITTED
server: https://35.0.0.200
name: gke_devops-with-kubernetes_us-central1-b_cluster
contexts:
- context:
cluster: gke_devops-with-kubernetes_us-central1-b_cluster
user: gke_devops-with-kubernetes_us-central1-b_cluster
name: gke_devops-with-kubernetes_us-central1-b_cluster
current-context: gke_devops-with-kubernetes_us-central1-b_cluster
kind: Config
preferences: {}
users:
- name: gke_devops-with-kubernetes_us-central1-b_cluster
user:
auth-provider:
config:
access-token: XXXXX
cmd-args: config config-helper --format=json
cmd-path: /Users/devops/k8s/bin/gcloud
expiry: 2018-12-16T02:51:21Z
expiry-key: '{.credential.token_expiry}'
token-key: '{.credential.access_token}'
name: gcp

As we learned previously, we can use `kubectl config use-context CONTEXT_NAME` to switch the context. We can also specify `kubeconfig` files according to the `$KUBECONFIG` environment variable to determine which kubeconfig files are used. In this way,

config files could be merged. For example, the following command will merge `kubeconfig-file1` and `kubeconfig-file2`:

```
export KUBECONFIG=$KUBECONFIG: kubeconfig-file1: kubeconfig-file2
```

We could also use `kubectl config --kubeconfig=<config file name> set-cluster <cluster name>` to specify the target cluster in the target `kubeconfig` file.

By default, the `kubeconfig` file is located at `$HOME/.kube/config`. This file will be loaded if none of the preceding settings are set.

// list service account across all namespaces # kubectl get serviceaccount --all-namespaces NAMESPACE NAME SECRETS AGE

default default 1 5d

kube-public default 1 5d

kube-system namespace-controller 1 5d kube-system resourcequota-controller 1 5d kube-system service-account-controller 1 5d kube-system service-controller 1 5d

chapter5 default 1 2h

...

kubectl describe serviceaccount/default Name: default
Namespace: default
Labels: <none>
Annotations: <none>
Image pull secrets: <none>
Mountable secrets: default-token-52qnr
Tokens: default-token-52qnr
Events: <none>

// describe the secret, the name is default-token-52qnr here # kubectl describe secret default-token-52qnr Name: default-token-52qnr

Namespace: chapter5

Annotations: kubernetes.io/service-account.name: default
 kubernetes.io/service-account.uid: 6bc2f108-dae5-11e8-b6f4-42010a8a0244 Type: kubernetes.io/service-account-token Data

====

ca.crt: # the public CA of api server. Base64 encoded.

namespace: # the name space associated with this service account. Base64 encoded token: # bearer token. Base64 encoded

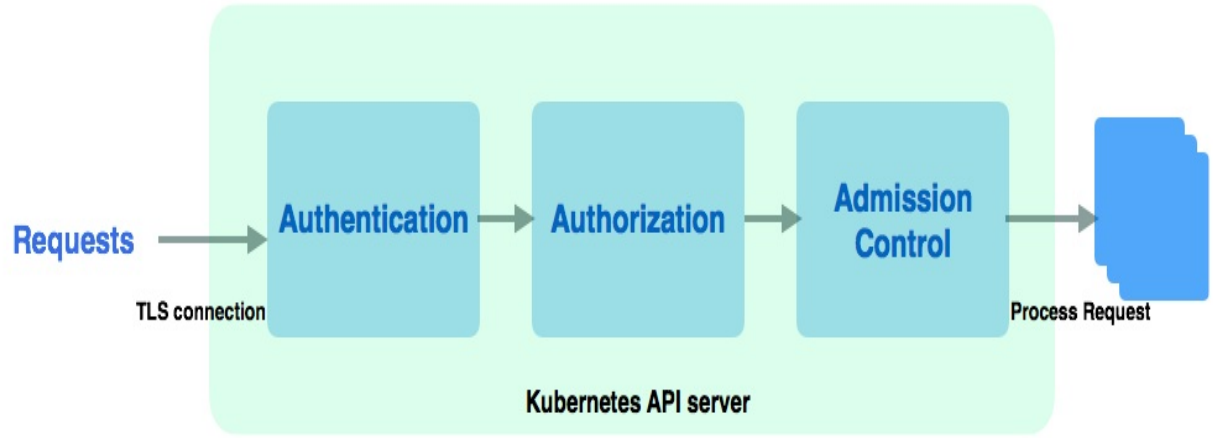
The service account secret will be automatically mounted to the `/var/run/secrets/kubernetes.io/serviceaccount` directory. When the pod accesses the API server, the API server will check the cert and token to do the authentication.

Specifying `automountServiceAccountToken: false` in the service account or pod specification could disable the auto-mount service account secret.

Authentication and authorization

Authentication and authorization are important components in Kubernetes. Authentication verifies users and checks that the user is who they claim to be. Authorization, on the other hand, checks what permission levels users have. Kubernetes supports different authentication and authorization modules.

The following is an illustration that shows how the Kubernetes API server processes access control when it receives a request:



Access control in the Kubernetes API server When the request goes to the API server, first it establishes a TLS connection by validating the clients' certificate with the **Certificate Authority** (**CA**) in the API server. The CA in the API server is usually at `/etc/kubernetes/`, and the clients' certificate is usually at `$HOME/.kube/config`. After the handshake, it moves into the authentication stage. In Kubernetes, authentication modules are chain-based. More than one authentication module can be used here. When receiving the request, Kubernetes will try all the authenticators one by one until it succeeds. If the request fails on all authentication modules, it will be rejected with an HTTP 401 Unauthorized error. If not, one of the authenticators verifies the user's identity and the requests are authenticated. At this point, Kubernetes authorization modules come into play.

Authorization modules verify whether or not the user has sufficient permissions to perform the action that they have requested to perform. Authorization modules are also chain-based. The authorization request needs to pass through every module until it succeeds. If the request fails for all modules, the requester will get a HTTP 403 Forbidden response.

Admission control is a set of configurable plugins in an API server that determine whether a request is admitted or denied. At this stage, if the request doesn't pass through one of the plugins, then it is denied immediately.

Authentication

By default, a service account is token-based. When you create a service account or a namespace with a default service account, Kubernetes creates the token, stores it as a secret that is encoded by Base64, and mounts the secret as a volume into the pod. Then, the processes inside the pod have the ability to talk to the cluster. The user account, on the other hand, represents a normal user who might use `kubectl` to directly manipulate the resource.

Service account token authentication

When we create a service account, a signed bearer token will be automatically created by the Kubernetes service account admission controller plugin. We can use that service account token to authenticate a user.

Let's try creating a service account named `myaccount` in the `chapter5` namespace: // the configuration file of service account object
 # cat service-account.yaml
apiVersion: v1
kind: ServiceAccount
metadata:
name: myaccount
namespace: chapter5

// create myaccount
 # kubectl create -f service-account.yaml
serviceaccount/myaccount created

In `Chapter 9`, *Continuous Delivery*, in the example in which we demonstrated how to deploy `my-app`, we created a namespace named `cd`, and used `get-sa-token.sh` script (`https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/blob/master/chapter9/9-2_service-account-for-ci-tool/utils/push-cd/get-sa-token.sh`) to export the token for us:

```
// export ca.crt and sa.token from myaccount in namespace chapter5
# sh ./chapter9/9-2_service-account-for-ci-tool/utils/push-cd/get-sa-token.sh -n chapter
```

Then, we created a user named `mysa` via the `kubectl config set-credentials <user> --token=$TOKEN` command: // **CI_ENV_K8S_SA_TOKEN=`cat sa.token`**
 # kubectl config set-credentials mysa --token=${K8S_SA_TOKEN}

Next, we set the context to bind with a user and namespace: // **Here we set K8S_CLUSTER=gke_devops-with-kubernetes_us-central1-b_cluster**
 # kubectl config set-context myctxt --cluster=${K8S_CLUSTER} --user=mysa

Finally, we set our `myctxt` context as the default context:

```
// set the context to myctxt
# kubectl config use-context myctxt
```

When we send a request, the token will be verified by the API server, which checks whether the requester is eligible and is what it claims to be. Let's see if we can use this token to list the pods in the default namespace: **# kubectl get po Error from server (Forbidden): pods is forbidden: User "system:serviceaccount:chapter5:myaccount" cannot list pods in the namespace "default"**

Seems like something went wrong! This is because we haven't granted any permissions to this service account yet. We'll learn how to do this using `Role` and `RoleBinding` later in this chapter.

User account authentication

There are several implementations for user account authentication. This includes anything from client certificates, bearer tokens, and static files to OpenID connect tokens. You can choose more than one as authentication chains. Here, we'll demonstrate how client certificates work.

In `Chapter 9`, *Continuous Delivery*, and earlier in this chapter, we learned how to export certificates and tokens for service accounts. Now, let's learn how to do this for a user. Let's assume that we are still inside the `chapter5` namespace, and we want to create a user for our new DevOps member, Linda, who will help us do the deployment for `my-app`:

1. First, we'll generate a private key via OpenSSL (`https://www.openssl.org`):

```
// generate a private key for Linda
# openssl genrsa -out linda.key 2048
```

2. Next, we'll create a certificate sign request (`.csr`) for Linda:

```
// making CN as your username
# openssl req -new -key linda.key -out linda.csr -subj "/CN=linda"
```

3. Now, `linda.key` and `linda.csr` should be located in the current folder. To let the API server validate Linda's certificate, we'll need to find the cluster CA.

In minikube, the root CA is under `~/.minikube/`. For other self-hosted solutions, it's normally under `/etc/kubernetes/`. If you use `kops` to deploy the cluster, the location is under `/srv/kubernetes`, and you will find the path in the `/etc/kubernetes/manifests/kube-apiserver.manifest` file. In GKE, the cluster root CA is non-exportable.

Let's assume we have `ca.crt` and `ca.key` under the current folder; by using these, we could generate a new CA for the user. By using the `-days` parameter, we can define the expired date: **// generate the cert for Linda, this cert is only valid for 30 days. # openssl x509 -req -in linda.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out linda.crt -days 30 Signature ok subject=/CN=linda Getting CA Private Key**

After we have cert signed by our cluster, we could set a user in the cluster, as

follows: **# kubectl config set-credentials linda --client-certificate=linda.crt --client-key=linda.key User "linda" set.**

Remember the concept of context: this is the combination of cluster information, a user for authentication, and a namespace. Now, we'll set a context entry in `kubeconfig`. Remember to replace your cluster name, namespace, and user in the following example: **# kubectl config set-context devops-context --cluster=${K8S_CLUSTER} --namespace=chapter5 --user=linda Context "devops-context" modified.**

Now, Linda should have zero permission: **// test for getting a pod # kubectl --context=devops-context get pods Error from server (Forbidden): User "linda" cannot list pods in the namespace "chapter5". (get pods)**

Linda can now pass the authentication stage as Kubernetes knows she is Linda. However, to give Linda permission to do the deployment, we need to set up the policies in authorization modules.

Authorization

Kubernetes supports several authorization modules. At the time of writing this book, it supports the following:

- ABAC
- RBAC
- Node authorization
- Webhook
- Custom modules

Attribute-Based Access Control (**ABAC**) was the major authorization mode before **Role-Based Access Control** (**RBAC**) was introduced. Node authorization is used by kubelet to make a request to the API server. Kubernetes supports the webhook authorization mode to establish a HTTP callback with an external RESTful service. It'll do a POST whenever it faces an authorization decision. Another common way to do this is by implementing your in-house module by following along with the pre-defined authorizer interface. For more implementation information, refer to `https://kubernetes.io/docs/admin/authorization/#custom-modules`. In this section, we'll walk though how to leverage and utilize RBAC in Kubernetes.

Role-based access control (RBAC)

Since Kubernetes 1.6, RBAC is enabled by default. In RBAC, the admin creates several `Roles` or `ClusterRoles` that define the fine-grained permissions that specify a set of resources and actions (verbs) that roles can access and manipulate. After that, the admin grants the `Role` permission to users through `RoleBinding` or `ClusterRoleBindings`.

If you're running minikube, add `--extra-config=apiserver.Authorization.Mode=RBAC` when using `minikube start`. If you're running a self-hosted cluster on AWS via kops, add `--authorization=rbac` when launching the cluster. Kops launches an API server as a pod; using the `kops edit cluster` command could modify the `spec` of the containers. EKS and GKE support RBAC natively.

Roles and ClusterRoles

A `Role` in Kubernetes is bound within a namespace. A `ClusterRole`, on the other hand, is cluster-wide. The following is an example of `Role`, which can perform all operations, including `get`, `watch`, `list`, `create`, `update`, `delete`, and `patch`, on the `deployments`, `replicasets`, and `pods` resources: **// configuration file for a role named devops-role**
 # cat 5-2_rbac/5-2_role.yaml kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
namespace: chapter5
name: devops-role
rules:
- apiGroups: ["", "extensions", "apps"]
resources:
- "deployments"
- "replicasets"
- "pods"
verbs: ["*"]

// create the role
 # kubectl create -f 5-2_rbac/5-2_role.yaml
role.rbac.authorization.k8s.io/devops-role created

In GKE, the admin doesn't have permission to create a role by default. Instead, you must grant the user access to this with the following command: `kubectl create clusterrolebinding cluster-admin-binding --clusterrole cluster-admin --user ${USER_ACCOUNT}`.

In `apiGroups`, an empty string `[""]` indicates the core API group. The API group is part of the RESTful API call. The core indicates the original API call path, such as `/api/v1`. The newer REST path has the group name and API version in it, such as `/apis/$GROUP_NAME/$VERSION`. To look up API groups you'd like to use, check out API references at `https://kubernetes.io/docs/reference`. Under `resources`, you could add the `resources` you'd like to grant access to, and under `verbs`, you could list a set of actions that this role could perform. Let's get into a more advanced example for `ClusterRoles`, which we used in `Chapter 9`, *Continuous Delivery*: //

configuration file for a cluster role
 # cat 5-2_rbac/5-2_clusterrole.yaml apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
name: cd-role
rules:
- apiGroups: ["extensions", "apps"]
resources:
- deployments
- replicasets
- ingresses
verbs: ["*"]
- apiGroups: [""]
resources:
- namespaces
- events
verbs: ["get", "list", "watch"]
- apiGroups: [""]
resources:
- pods
- services
- secrets
- replicationcontrollers
- persistentvolumeclaims
- jobs
- cronjobs
verbs: ["*"]

// create the cluster role
 # kubectl create -f 5-2_rbac/5-2_clusterrole.yaml
clusterrole.rbac.authorization.k8s.io/cd-role created

`ClusterRole` is cluster-wide. Some resources don't belong to any namespace, such as nodes, and can only be controlled by `ClusterRole`. The namespaces it can access depends on the `namespaces` field that it associates with `ClusterRoleBinding`. In the preceding example, we granted the permission to allow this role read and write `deployments`, `replicasets`, and `ingresses` in both extensions and apps groups. In the

core API group, we only grant access for namespace and events, as well as all permissions for other resources, such as pods and services.

RoleBinding and ClusterRoleBinding

A `RoleBinding` binds a `Role` or `ClusterRole` to a list of users or service accounts. If a `ClusterRole` is bound with a `RoleBinding` instead of a `ClusterRoleBinding`, it will only be granted the permissions within the namespace where `RoleBinding` was specified.

The following is an example of the `RoleBinding` spec:

```
// configuration file for RoleBinding resource
# cat 5-2_rbac/rolebinding_user.yaml
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: devops-role-binding
  namespace: chapter5
subjects:
- kind: User
  name: linda
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: devops-role
  apiGroup: rbac.authorization.k8s.io

// create a RoleBinding for User linda
# kubectl create -f 5-2_rbac/rolebinding_user.yaml
rolebinding.rbac.authorization.k8s.io/devops-role-binding created
```

In this example, we bind a `Role` with a user through `roleRef`. This grants Linda the permission that we defined in the `devops-role`.

On the other hand, a `ClusterRoleBinding` is used to grant permission in all namespaces. Here, we'll leverage the same concept from `Chapter 9`, *Continuous Delivery*. First, we created a service account named `cd-agent`, then created a `ClusterRole` named `cd-role`, and a `ClusterRoleBinding` for `cd-agent` and `cd-role`. We then used `cd-agent` to do the deployment on our behalf: **// configuration file for ClusterRoleBinding**
 # cat 5-2_rbac/cluster_rolebinding_user.yaml apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding metadata: name: cd-agent roleRef: apiGroup: rbac.authorization.k8s.io kind: ClusterRole name: cd-role subjects: - apiGroup: rbac.authorization.k8s.io kind: User name: system:serviceaccount:cd:cd-agent

// create the ClusterRoleBinding
 # kubectl create -f 5-2_rbac/cluster_rolebinding_user.yaml
clusterrolebinding.rbac.authorization.k8s.io/cd-agent created

The `cd-agent` is bound with a `ClusterRole` via `ClusterRoleBinding`, so it can have the permission that's specified in `cd-role` across namespaces. Since a service account is created in a namespace, we'll need to specify its full name, including its `namespace`: **system:serviceaccount:<namespace>:<serviceaccountname>**

`ClusterRoleBinding` also supports `Group` as a subject.

Now, let's try to get pods again via `devops-context`: **# kubectl --context=devops-context get pods No resources found.**

We are no longer getting a forbidden response. What about if Linda wants to list namespaces—is this allowed?

```
# kubectl --context=devops-context get namespaces
Error from server (Forbidden): User "linda" cannot list namespaces at the cluster scope.
```

The answer is no, since Linda has not been granted permission to list namespaces.

--enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,DefaultStorageClass,

The following sections introduce these plugins and why we need them. For the latest information about supported admission control plugins, please visit the official documentation: https://kubernetes.io/docs/admin/admission-controllers.

NamespaceLifecycle

As we learned earlier, when a namespace is deleted, all objects in that namespace will be evicted as well. This plugin ensures that no new object creation requests can be made in a namespace that is terminating or non-existent. It also prevents Kubernetes native namespaces from being deleted.

LimitRanger

This plugin ensures that `LimitRange` can work properly. With `LimitRange`, we can set default requests and limits in a namespace, which will be used when launching a pod without specifying the requests and limits.

ServiceAccount

The service account plugin must be added if you use service account objects. For more information about `ServiceAccount`, revisit the *Service account token authentication* section of this chapter.

PersistentVolumeLabel

`PersistentVolumeLabel` adds labels to newly created PVs, based on the labels provided by the underlying cloud provider. This admission controller has been deprecated since the 1.8 release.

DefaultStorageClass

This plugin ensures that default storage classes work as expected if no `StorageClass` is set in a PVC. Different cloud providers implement their own `DefaultStorageClass` (such as how GKE uses Google Cloud Persistent Disk). Make sure you have this enabled.

ResourceQuota

Just like the `LimitRange`, if you're using the `ResourceQuota` object to administer a different level of QoS, this plugin must be enabled. The `ResourceQuota` should always be put at the end of the admission control plugin list. As we mentioned in the *ResourceQuota* section, `ResourceQuota` is used to limit the resource usage per namespace. Putting the `ResourceQuota` controller at the end of the admission controller list could prevent the request from increasing quota usage prematurely if it eventually gets rejected by controllers that are put after it, if any are.

DefaultTolerationSeconds

The `DefaultTolerationSeconds` plugin is used to set pods without any toleration sets. It will then apply for the default toleration for the `notready:NoExecute` and `unreachable:NoExecute` taints for 300 seconds. If you don't want this behavior to occur in the cluster, disable this plugin. For more information, please refer to the taints and tolerations section of `Chapter 8`, *Resource Management and Scaling*.

podNodeSelectorPluginConfig:
 clusterDefaultNodeSelector: <default-node-selectors-
 labels> namespace1: <namespace-node-selectors-labels-1>

namespace2: <namespace-node-selectors-labels-2>

By doing this, the `node-selector` annotation will be applied to the `namespace`. The pods on that namespace will then run on those matched nodes.

AlwaysPullImages

The pull policy defines the behavior when kubelet pulls images. The default pull policy is `IfNotPresent`, that is, it will pull the image if it is not present locally. If this plugin is enabled, the default pull policy will become `Always`, which is to always pull the latest image. This plugin also brings another benefit if your cluster is shared by different teams. Whenever a pod is scheduled, it'll always pull the latest image, regardless of whether the image exists locally or not. Then, we can ensure that a pod creation request always goes through an authorization check against the image.

DenyEscalatingExec

This plugin prevents any `kubectl exec` or `kubectl attach` commands from making pods escalate to privilege mode. Pods within privilege mode have access to the host namespace, which could become a security risk.

Other admission controller plugins

There are many other admission controller plugins we could use, such as `NodeRestriciton` to limit kubelet's permission, `ImagePolicyWebhook` to establish a webhook to control the access of the images, and `SecurityContextDeny` for controlling the privilege for a pod or a container. Please refer to the official documentation at `https://kubernetes.io/docs/admin/admission-controllers` to learn more about other plugins.

Dynamic admission control

Before Kubernetes 1.7, admission controllers were compiled with Kubernetes API server, so they could only be configured before the API server started. Dynamic admission control aimed to break this limitation. There are two methods to implement custom dynamic admission control: via initializer and admission webhooks. The Initializer webhook can watch an uninitialized workload and check whether it needs to take any action against it.

The Admission webhook intercepts the request and checks the preset rules from its configuration before deciding whether the requests are allowed or not. Both the initializer and admission webhooks can admit and mutate the resource request on certain operations, so we can leverage them to force policies or validate whether the requests fulfill the requirement of your organization. Buggy initializer and admission webhooks might block all the target resources from being created. However, the Admission webhook provides a failure policy, which can address when the webhook server doesn't respond as expected.

At the time of writing this book, the admission webhook has been promoted to beta, but the Initializer is still alpha. In this section, we'll implement a simple Admission webhook controller, which will verify whether the `{"chapter": "5"}` annotation is set to the `podSpec` during pod creation. The request will go through if the annotation is set. If not, the request will fail.

Admission webhook

There are two major components to implementing an Admission webhook controller: a webhook HTTP server to receive the resource life event and a `ValidatingWebhookConfiguration` or `MutatingWebhookConfiguration` resource configuration file. Please refer to https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter5/5-3_admission-webhook/sample-validating-admission-webhook for the source code of our sample Admission webhook.

Let's look at how to write a `ValidatingWebhookConfiguration`. As we can see in the following code, like normal objects, `ValidatingWebhookConfiguration` has an API version, a kind, and metadata with a name and labels. The important session is named `webhooks`. In `webhooks`, one or more rules need to be defined. Here, we are defining a rule that is triggered on any pod's creation request (`operations=CREATE, resources=pods`). The `failurePolicy` is used to determine the action if an error occurs when calling the webhook. The option of `failurePolicy` is either `Fail` or ignore (`Fail` means making the request fail, while ignore means the webhook error will be ignored). The `clientConfig` session defines the endpoint of the webhook server. Here, we are leveraging a Kubernetes service named `sample-webhook-service-svc`. If it's an external server, the URL could be specified directly rather than using a service. The `caBundle` is used to validate the webhook's server certificate. If not specified, by default, it'll use API server's system trust roots.

To export `caBundle` from Kubernetes Service, use the `kubectl get configmap -n kube-system extension-apiserver-authentication -o=jsonpath='{.data.client-ca-file}' | base64 | tr -d '\n'` command and replace the `${CA_BUNDLE}` field as follows:

```
# cat chapter5/5-3_admission-webhook/sample-validating-admission-webhook/validatingwebhc
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: sample-webhook-service
  labels:
    app: sample-webhook-service
webhooks:
  - name: devops.kubernetes.com
    rules:
      - apiGroups:
          - ""
        apiVersions:
          - v1
        operations:
```

```
        - CREATE
      resources:
        - pods
  failurePolicy: Fail
  clientConfig:
    service:
      name: sample-webhook-service-svc
      namespace: default
      path: "/"
    caBundle: ${CA_BUNDLE}
```

To make a webhook HTTP server, we create a simple express web application in Node.js (https://expressjs.com/). The main logic of the application is to receive a pod creation event and send an `admissionResponse` (https://github.com/kubernetes/api/blob/master/admission/v1beta1/types.go#L81) back. Here, we'll return an `admissionResponse` with the `allowed` field, which indicates whether a request is allowed or denied:

```
function webhook(req, res) {
  var admissionRequest = req.body;
  var object = admissionRequest.request.object;

  var allowed = false;
  if (!isEmpty(object.metadata.annotations) && !isEmpty(object.metadata.annotations.chap
    allowed = true;
  }

  var admissionResponse = {
    allowed: allowed
  };

  for (var container of object.spec.containers) {
    console.log(container.securityContext);
    var image = container.image;
        var admissionReview = {
          response: admissionResponse
        };
        console.log("Response: " + JSON.stringify(admissionReview));
        res.setHeader('Content-Type', 'application/json');
        res.send(JSON.stringify(admissionReview));
        res.status(200).end();
    };
};
```

In the preceding function, we checked whether `object.metadata.annotations.chapter` is annotated in the pod and whether the chapter is equal to `5`. If it is, the webhook server will pass the request. The webhook and API servers need to establish a mutual trust. To do this, we'll generate a certificate for the webhook server by requesting a certificate signing through the `certificates.k8s.io` API. The popular service mesh implementation, Istio (https://istio.io/), has a useful tool for us (https://raw.githubusercontent.com/istio/istio/41203341818c4dada2ea5385cfedc7859c01e957/install/kubernetes/webhook-create-signed-cert.sh): **# wget https://raw.githubusercontent.com/istio/istio/41203341818c4dada2ea5385cfe create-signed-cert.sh**

// create the cert
 # sh webhook-create-signed-cert.sh --service sample-webhook-service-svc --secret sample-webhook-service-certs --namespace default

`server-key.pem` and `server-cert.pem` will be generated via the script under the default `temp` folder in your operating system. Copying them can put them under the `src/keys` folder inside our sample webhook HTTP server. Now, it's time to build the application with docker via `docker build -t $registry/$repository:$tag .` and push the target docker image to the registry (here, we used `devopswithkubernetes/sample-webhook-service:latest`). After doing this, we can launch the webserver: **# cat chapter5/5-3_admission-webhook/sample-validating-admission-webhook/deployment.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: sample-webhook-service
labels:
app: sample-webhook-service
spec:
replicas: 1
selector:
matchLabels:
app: sample-webhook-service
template:
metadata:
labels:
app: sample-webhook-service
spec:
containers:
- name: sample-webhook-service
image: devopswithkubernetes/sample-webhook-service:latest
imagePullPolicy: Always
---
apiVersion: v1
kind: Service
metadata:
name: sample-webhook-service-svc
```

labels:
app: sample-webhook-service
spec:
ports:
- port: 443
targetPort: 443
selector:
app: sample-webhook-service

 # kubectl create -f chapter5/5-3_admission-webhook/sample-validating-admission-webhook/deployment.yaml
deployment.apps/sample-webhook-service created
service/sample-webhook-service-svc created

After checking that the pod is up and running, we can create a `ValidatingWebhookConfiguration` resource: **# kubectl create -f chapter5/5-3_admission-webhook/sample-validating-admission-webhook/validatingwebhookconfiguration.yaml**
validatingwebhookconfiguration.admissionregistration.k8s.io/sample-webhook-service
created

 # kubectl get validatingwebhookconfiguration
NAME CREATED AT
sample-webhook-service 2018-12-27T21:04:50Z

Let's try deploying two `nginx` pods without any annotations: **# cat chapter5/5-3_admission-webhook/sample-validating-admission-webhook/test-sample.yaml**
apiVersion: apps/v1
kind: Deployment
metadata:
name: nginx
labels:
app: nginx
spec:
replicas: 2

selector:
matchLabels:
app: nginx
template:
metadata:
labels:
app: nginx
#annotations:
 # chapter: "5"
spec:
containers:
- name: nginx
image: nginx

 # kubectl create -f chapter5/5-3_admission-webhook/sample-validating-admission-webhook/test-sample.yaml
Hdeployment.apps/nginx created

This is supposed to create two pods; however, we did not see any `nginx` pods being created. We can only see our webhook service pod: **# kubectl get po**
NAME READY STATUS RESTARTS AGE
sample-webhook-service-789d87b8b7-m58wq 1/1 Running 0 7h

If we check the corresponding `ReplicaSet`, and use `kubectl describe rs $RS_NAME` to check the events, we will get the following result: **# kubectl get rs**
NAME DESIRED CURRENT READY AGE
nginx-78f5d695bd 2 0 0 1m
sample-webhook-service-789d87b8b7 1 1 1 7h

 # kubectl describe rs nginx-78f5d695bd
Name: nginx-78f5d695bd
Namespace: default
Selector: app=nginx,pod-template-hash=3491825168
Labels: app=nginx
pod-template-hash=3491825168
Annotations: deployment.kubernetes.io/desired-replicas: 2
deployment.kubernetes.io/max-replicas: 3

deployment.kubernetes.io/revision: 1
Controlled By: Deployment/nginx
Replicas: 0 current / 2 desired
Pods Status: 0 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
Labels: app=nginx
pod-template-hash=3491825168
Containers:
nginx:
Image: nginx
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 28s (x15 over 110s) replicaset-controller Error creating: Internal error occurred: admission webhook "devops.kubernetes.com" denied the request without explanation

From this, we can see that the admission webhook denies the request. Delete and recreate the deployment by using the `uncomment` annotation in the preceding code block:

```
// uncomment this annotation.
     #annotations:
     #  chapter: "5"
```

If we do this, we should be able to see that the `nginx` pods are created accordingly: **# kubectl get po**
NAME READY STATUS RESTARTS AGE
nginx-978c784c5-v8xk9 0/1 ContainerCreating 0 2s
nginx-978c784c5-wrmdb 1/1 Running 0 2s

sample-webhook-service-789d87b8b7-m58wq 1/1 Running 0 7h

The request passed through the authentication, authorization, and admission controls, including our webhook service. The pod objects were created and scheduled accordingly.

Please remember to clean up after testing the dynamic admission controllers. It might block pod creation in future experiments.

Custom resources

Custom resources, which were first introduced in Kubernetes 1.7, were designed as an extension point to let users create custom API objects and act as native Kubernetes objects. This was done so that users could extend Kubernetes to support the custom objects for their application or specific use cases. Custom resources can be dynamically registered and unregistered. There are two ways to create custom resources: by using a CRD or aggregated API. CRDs are much easier, while an aggregated API requires additional coding in Go. In this section, we'll learn how to write a CRD from scratch.

Custom resources definition

Creating a **Custom Resources Definition** (**CRD**) object includes two steps: CRD registration and object creation.

Let's create a CRD configuration first: **# cat chapter5/5-4_crd/5-4-1_crd.yaml**

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
name: books.devops.kubernetes.com
spec:
group: devops.kubernetes.com
version: v1alpha1
scope: Namespaced
names:
plural: books
singular: book
kind: Book
shortNames:
- bk
validation:
openAPIV3Schema:
required: ["spec"]
properties:
spec:
required: ["name", "edition"]
properties:
name:
type: string
minimum: 50
edition:
type: string
minimum: 10
chapter:
type: integer
```

minimum: 1
maximum: 2

With `CustomResourceDefinition`, we can define our own spec for the custom object. First, we'll have to decide on the name of the CRD. The naming convention for a CRD must be `spec.names.plural+"."+spec.group`. Next, we'll define the group, version, scope, and names. The scope is either `Namespaced` or `Cluster` (non-namespaced). After Kubernetes 1.13, we can add a validation section to validate the custom objects via the OpenAPI v3 schema (`https://github.com/OAI/OpenAPI-Specification/blob/master/versions/3.0.0.md#schemaObject`). We can define the required fields, as well as the spec and validation condition for each field. In the preceding example, we created a custom object named `books.devops.kubernetes.com` that has three properties: `name`, `edition`, and `chapter`. `name` and `edition` are required during object creation. Let's create the CRD via the `kubectl` command. We can also list all the CRDs via the `kubectl get crd` command: **# kubectl create -f chapter5/5-4_crd/5-4-1_crd.yaml**
customresourcedefinition.apiextensions.k8s.io/books.devops.kubernetes.com created

 # kubectl get crd
NAME CREATED AT
backendconfigs.cloud.google.com 2018-12-22T20:48:00Z
books.devops.kubernetes.com 2018-12-28T16:14:34Z
scalingpolicies.scalingpolicy.kope.io 2018-12-22T20:48:30Z

Next, we'll create an object accordingly. In `spec`, `name`, and `edition` are required. The `apiVersion` will be the `<group>/<version>` we defined in the preceding CRD configuration: **# cat chapter5/5-4_crd/5-4-2_objectcreation.yaml**
apiVersion: devops.kubernetes.com/v1alpha1
kind: Book
metadata:
name: book-object
spec:
name: DevOps-with-Kubernetes
edition: second

 # kubectl create -f chapter5/5-4_crd/5-4-2_objectcreation.yaml

book.devops.kubernetes.com/book-object created

If we set the `edition` to `second`, an error will be thrown, as follows:

```
spec.edition in body must be of type string: "integer"
spec.edition in body should be greater than or equal to 10
```

Now, we should be able to get and describe it, just like a normal API object: **# kubectl get books**
NAME AGE
book-object 3s

 # kubectl describe books book-object
Name: book-object
Namespace: default
Labels: <none>
Annotations: <none>
API Version: devops.kubernetes.com/v1alpha1
Kind: Book
Metadata:
/apis/devops.kubernetes.com/v1alpha1/namespaces/default/books/book-object
UID: c6912ab5-0abd-11e9-be06-42010a8a0078
Spec:
Edition: second
Name: DevOps-with-Kubernetes
Events: <none>

After registering a CRD, a custom controller might be needed to handle custom object operations. The custom controller requires additional programming effort. There are also multiple tools available in the community that can help us create a skeleton controller, such as the following:

- Controller: `https://github.com/kubernetes/sample-controller`
- Kubebuilder: `https://github.com/kubernetes-sigs/kubebuilder`
- Operator: `https://coreos.com/operators/`

With the sample controller (provided by Kubernetes), a set of `ResourceEventHandlerFuncs` will be added into `EventHandler` for handling object life

cycle events, such as `AddFunc` for `UpdateFunc` and `DeleteFunc`.

Both **Kubebuilder** and **Operator** can simplify the preceding steps. Kubebuilder provides support for building APIs via CRDs, controllers, and admission webhooks. Operator, which was introduced by CoreOS, is an application-specific controller that's implemented with CRD. There are existing operators being implemented by the community, and they can be found at `https://github.com/operator-framework/awesome-operators`. We'll introduce how to leverage the operator SDK in the Operator framework to build a simple controller with the same book CRD.

First, we'll have to install the operator SDK (`https://github.com/operator-framework/operator-sdk`). We are using v.0.3.0 in the following example: **// Check the prerequisites at https://github.com/operator-framework/operator-sdk#prerequisites**
 # mkdir -p $GOPATH/src/github.com/operator-framework # cd $GOPATH/src/github.com/operator-framework # git clone https://github.com/operator-framework/operator-sdk # cd operator-sdk # git checkout master # make dep # make install

// check version
 # operator-sdk --version
 # operator-sdk version v0.3.0+git

Let's create a new operator named `devops-operator` via the following command: **// operator-sdk new <operator_name>**
 # operator-sdk new devops-operator
INFO[0076] Run git init done
INFO[0076] Project creation complete.

After the operator is initialized, we can start adding components to it. Let's add `api` to create an API object and `controller` to handle object operations: **// operator-sdk add api --api-version <group>/<version> --kind <kind>**
 # operator-sdk add api --api-version devops.kubernetes.com/v1alpha1 --kind Book
INFO[0000] Generating api version devops.kubernetes.com/v1alpha1 for kind Book.
INFO[0000] Create pkg/apis/devops/v1alpha1/book_types.go

INFO[0000] Create pkg/apis/addtoscheme_devops_v1alpha1.go
INFO[0000] Create pkg/apis/devops/v1alpha1/register.go
INFO[0000] Create pkg/apis/devops/v1alpha1/doc.go
INFO[0000] Create deploy/crds/devops_v1alpha1_book_cr.yaml
INFO[0000] Create deploy/crds/devops_v1alpha1_book_crd.yaml
INFO[0008] Running code-generation for Custom Resource group versions: [devops:[v1alpha1],]
INFO[0009] Code-generation complete.
INFO[0009] Api generation complete.

 # operator-sdk add controller --api-version devops.kubernetes.com/v1alpha1 --kind Book
INFO[0000] Generating controller version devops.kubernetes.com/v1alpha1 for kind Book.
INFO[0000] Create pkg/controller/book/book_controller.go
INFO[0000] Create pkg/controller/add_book.go
INFO[0000] Controller generation complete.

There are multiple files we need to modify. The first one is API spec. In the previous CRD example, we added three custom properties in the `book` resource: `name`, `edition`, and `chapter`. We'll need to add that into spec here, too. This can be found under `pkg/apis/devops/v1alpha1/book_types.go`: **// in pkg/apis/devops/v1alpha1/book_types.go**

```
type BookSpec struct {
// INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
// Important: Run "operator-sdk generate k8s" to regenerate code after modifying this file
Name string `json:"name"`
Edition string `json:"edition"`
Chapter int32 `json:"chapter"`
}
```

Run `operator-sdk generate k8s` after modifying the file, as shown in the preceding code. Next, we will add some custom logic to the controller logic. This is located in the `Reconcile` function in `pkg/controller/book/book_controller.go`.

In the existing example that was created by the framework, the controller will

receive a `podSpec`. This is exactly what we need, and we'll just get the `name` and `edition` from the `Spec` and print it out in `busybox stdout`: // in pkg/controller/book/book_controller.go
func newPodForCR(cr *devopsv1alpha1.Book) *corev1.Pod {
labels := map[string]string{
"app": cr.Name,
}
name := cr.Spec.Name
edition := cr.Spec.Edition
return &corev1.Pod{
ObjectMeta: metav1.ObjectMeta{
Name: cr.Name + "-pod",
Namespace: cr.Namespace,
Labels: labels,
},
Spec: corev1.PodSpec{
Containers: []corev1.Container{
{
Name: "busybox",
Image: "busybox",
Command: []string{"echo", "Please support", name, edition, "Edition :-) "},
Stdin: true,
},
},
},
}
}

Then, we can run `operator-sdk build devopswithkubernetes/sample-operator` to build the docker image and push it to a registry. Here, we'll just push it to our public docker hub, `docker push devopswithkubernetes/sample-operator`.

The operator is done! After that, we can start looking into how to deploy it. The deployment scripts are automatically created in the `deploy` folder. The file we need to change is `operator.yaml`, which specifies the operator container image. Find the `image: REPLACE_IMAGE` line in `podSpec` and update that so that it points into

your registry (here, we'll point it to `devopswithkubernetes/sample-operator`). Now, it's good to deploy: **# kubectl create -f deploy/service_account.yaml # kubectl create -f deploy/role.yaml # kubectl create -f deploy/role_binding.yaml # kubectl create -f deploy/crds/app_v1alpha1_appservice_crd.yaml # kubectl create -f deploy/operator.yaml**

Now, you should be able to see an operator pod when listing the pods: **# kubectl get po**
NAME READY STATUS RESTARTS AGE
devops-operator-58476dbcdd-s5m5v 1/1 Running 0 41m

Now, let's create a `Book` resource. You could modify `deploy/crds/devops_v1alpha1_book_cr.yaml` in the current folder or reuse `5-4_crd/5-4-2_objectcreation.yaml` from our repo: **# kubectl create -f deploy/crds/devops_v1alpha1_book_cr.yaml**
book.devops.kubernetes.com/book-object created

Then, we should be able to see that another pod was created by the CRD, and we can also check its logs: **# kubectl get po**
NAME READY STATUS RESTARTS AGE
book-object-pod 0/1 Completed 0 2s
devops-operator-58476dbcdd-s5m5v 1/1 Running 0 45m

 # kubectl logs book-object-pod
Please support DevOps-with-Kubernetes second Edition :-)

Hurray! Everything looks fine. Here, we have demonstrated a very simple example. Of course, we could leverage this concept and evolve more sophisticated logic and handlers.

Application CRD:
A containerized application might contain multiple Kubernetes resources, such as deployments, services, `ConfigMaps`, secrets, as well as custom CRDs. An application CRD has been implemented at `https://github.com/kubernetes-sigs/application`, providing a bridge to make application metadata describable. It also has application level health checks so that users don't need to list all the resources after deployment and check whether the application has been deployed properly. Instead, they list the application CRD and check its status.

Summary

In this chapter, we learned about what `namespace` and `context` are, including how they work, and a how to switch between a physical cluster and virtual cluster by setting the context. We then learned about an important object—service account, which provides the ability to identify processes that are running within a pod. Then, we familiarized ourselves with how to control access flow in Kubernetes. We learned what the difference is between authentication and authorization, and how these work in Kubernetes. We also learned how to leverage RBAC to have fine-grained permission for users. In addition, we looked at a couple of admission controller plugins and dynamic admission controls, which are the last goalkeepers in the access control flow. Finally, we learned about what the CRD is and implemented it and its controller via the operator SDK (`https://github.com/operator-framework/operator-sdk`) in the operator framework.

In `Chapter 6`, *Kubernetes Network*, we'll move on and learn more about cluster networking.

Kubernetes Network

In `Chapter 3`, *Getting Started with Kubernetes*, we learned how to deploy containers with different resources and also looked at how to use volumes to persist data, dynamic provisioning, different storage classes, and advanced administration in Kubernetes. In this chapter, we'll learn how Kubernetes routes traffic to make all of this possible. Networking always plays an important role in the software world. We'll learn about Kubernetes networking step by step, looking at the communication between containers on a single host, multiple hosts, and inside a cluster.

The following are the topics we'll cover in this chapter:

- Kubernetes networking
- Docker networking
- Ingress
- Network policy
- Service mesh

Kubernetes networking

There are plenty of options when it comes to implementing networking in Kubernetes. Kubernetes itself doesn't care about how you implement it, but you must meet its three fundamental requirements:

- All containers should be accessible to each other without NAT, regardless of which nodes they are on
- All nodes should communicate with all containers
- The IP container should see itself in the same way as others see it

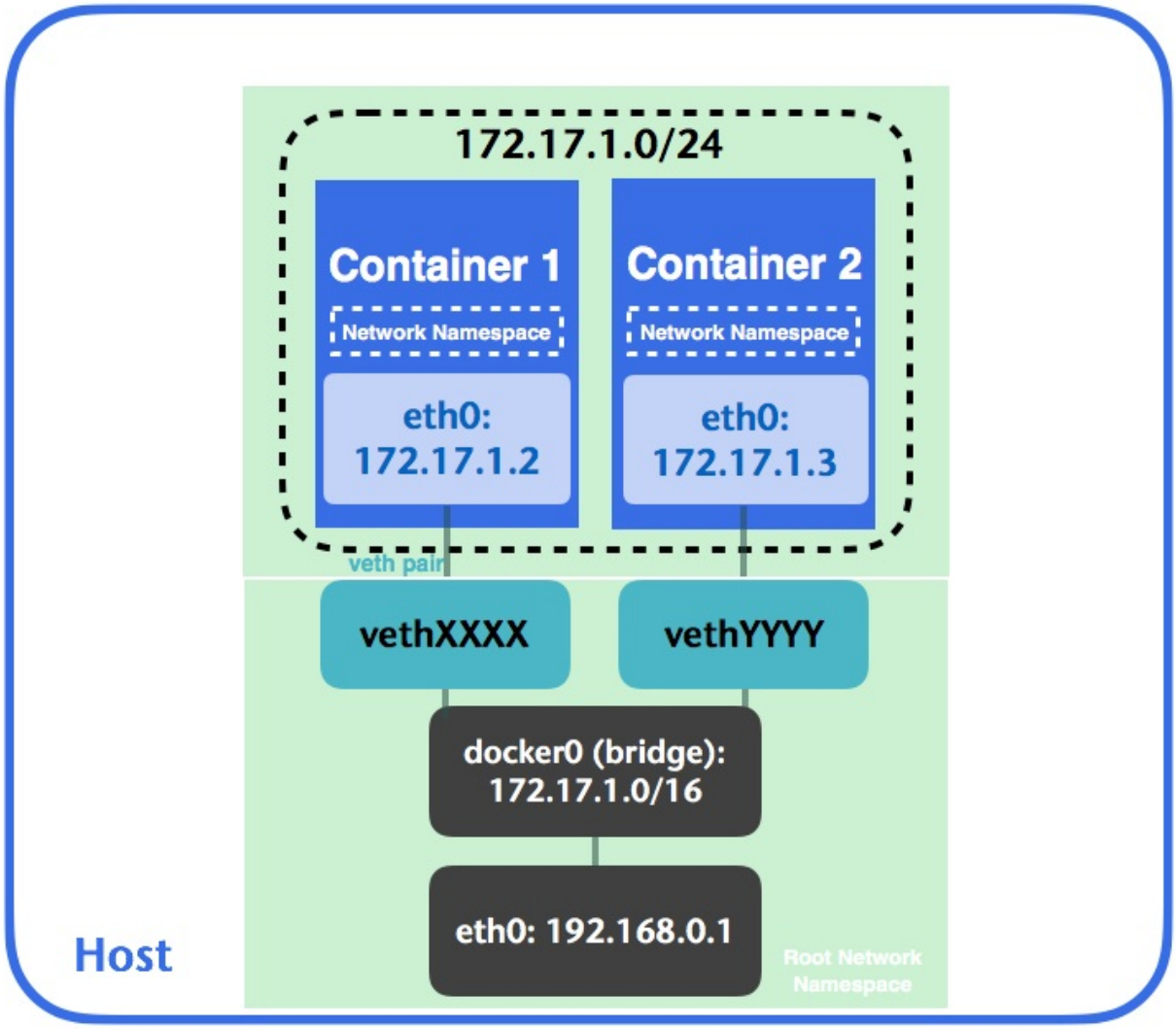
Before getting any further into this, we'll first examine how default container networking works.

Docker networking

Let's now review how docker networking works before getting into Kubernetes networking. For container networking, there are different modes: bridge, none, overlay, macvlan, and host. We've learned about the major modes in `Chapter 2`, *DevOps with Containers.* Bridge is the default networking model. Docker creates and attaches a virtual Ethernet device (also known as `veth`) and assigns a network namespace to each container.

*The **network namespace** is a feature in Linux that is logically another copy of a network stack. It has its own routing tables, ARP tables, and network devices. This is a fundamental concept of container networking.*

`Veth` always comes in pairs; one is in the network namespace and the other is in the bridge. When the traffic comes into the host network, it will be routed into the bridge. The packet will be dispatched to its `veth`, and will go into the namespace inside the container, as shown in the following diagram:



Let's take a closer look at this. In the following example, we'll use a minikube node as the docker host. First, we'll have to use `minikube ssh` to ssh into the node because we're not using Kubernetes yet. After we get into the minikube node, we'll launch a container to interact with us:

```
// launch a busybox container with `top` command, also, expose container port 8080 to ho
# docker run -d -p 8000:8080 --name=busybox busybox top
737e4d87ba86633f39b4e541f15cd077d688a1c8bfb83156d38566fc5c81f469
```

Let's see the implementation of outbound traffic within a container. `docker exec <container_name or container_id>` can run a command in a running container. Let's use `ip link list` to list all the interfaces:

```
// show all the network interfaces in busybox container
// docker exec <container_name> <command>
# docker exec busybox ip link list
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: sit0@NONE: <NOARP> mtu 1480 qdisc noop qlen 1
   link/sit 0.0.0.0 brd 0.0.0.0
53: eth0@if54: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN>
    mtu 1500 qdisc noqueue
   link/ether 02:42:ac:11:00:07 brd ff:ff:ff:ff:ff:ff
```

We can see that we have three interfaces inside the `busybox` container. One has an ID of `53` with the name `eth0@if54`. The number after `if` is the other interface ID in the pair. In this case, the pair ID is `54`. If we run the same command on the host, we can see that the `veth` in the host is pointing to `eth0` inside the container:

```
// show all the network interfaces from the host
# ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
   state UNKNOWN mode DEFAULT group default qlen 1
   link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
   pfifo_fast state UP mode DEFAULT group default qlen
   1000
   link/ether 08:00:27:ca:fd:37 brd ff:ff:ff:ff:ff:ff
...
54: vethfeec36a@if53: <BROADCAST,MULTICAST,UP,LOWER_UP>
    mtu 1500 qdisc noqueue master docker0 state UP mode
    DEFAULT group default
   link/ether ce:25:25:9e:6c:07 brd ff:ff:ff:ff:ff:ff link-netnsid 5
```

We have a `veth` on the host named `vethfeec36a@if53`. This pairs with `eth0@if54` in the container network namespace. `veth` 54 is attached to the `docker0` bridge, and eventually accesses the internet via `eth0`. If we take a look at the `iptables` rules, we can find a masquerading rule (also known as SNAT) on the host that docker creates for outbound traffic, which will make internet access available for containers:

```
// list iptables nat rules. Showing only POSTROUTING rules which allows packets to be al
# sudo iptables -t nat -nL POSTROUTING
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
...
MASQUERADE  all  --  172.17.0.0/16        0.0.0.0/0
...
```

On the other hand, as regards the inbound traffic, docker creates a custom filter chain on prerouting and dynamically creates forwarding rules in the `DOCKER` filter chain. If we expose a container port, `8080`, and map it to a host port, `8000`, we can see that we're listening to port `8000` on any IP address (`0.0.0.0/0`), which will then be routed to container port `8080`:

```
// list iptables nat rules
# sudo iptables -t nat -nL
```

```
Chain PREROUTING (policy ACCEPT)
target     prot opt source               destination
...
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0            ADDRTYPE match dst-type LO
...
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0           !127.0.0.0/8          ADDRTYPE match dst-type LO
...
Chain DOCKER (2 references)
target     prot opt source               destination
RETURN     all  --  0.0.0.0/0            0.0.0.0/0
...
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:8000 to:172.17.0.7
...
```

Now that we know how a packet goes in/out of containers, let's look at how containers in a pod communicate with each other.

User-defined custom bridges:
As well as the default bridge network, docker also supports user-defined bridges. Users can create the custom bridge on the fly. This provides better network isolation, supports DNS resolution through embedded DNS server, and can be attached and detached from the container at runtime. For more information, please refer to the following documentation: https://docs.docker.com/network/bridge/#manage-a-user-defined-bridge.

#cat 6-1-1_pod.yaml

apiVersion: v1

kind: Pod

metadata:

  name: example

spec:

  containers:

  - name: web

  image: nginx

  - name: centos

  image: centos

  command: ["/bin/sh", "-c", "while : ;do curl http://localhost:80/; sleep 10; done"]

// create the Pod

#kubectl create -f 6-1-1_pod.yaml

pod/example created

kubectl describe pods example

Name: example

Node: minikube/192.168.99.100

...

Containers:

web:

Container ID: docker://
d9bd923572ab186870284535044e7f3132d5cac11ecb18576078b9c7bae86c73

Image: nginx

...

centos:

Container ID: docker:
//f4c019d289d4b958cd17ecbe9fe22a5ce5952cb380c8ca4f9299e10bf5e94a0f

Image: centos

...

docker ps

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

f4c019d289d4 36540f359ca3 "/bin/sh -c 'while : " 2 minutes ago Up 2 minutes k8s_centos_example_default_9843fc27-677b-11e7-9a8c-080027cafd37_1

d9bd923572ab e4e6d42c70b3 "nginx -g 'daemon off" 2 minutes ago Up 2 minutes k8s_web_example_default_9843fc27-677b-11e7-9a8c-080027cafd37_1

4ddd3221cc47 gcr.io/google_containers/pause-amd64:3.0 "/pause" 2 minutes ago Up 2 minutes

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.42.1"># docker inspect d9bd923572ab | grep NetworkMode

"NetworkMode":
"container:4ddd3221cc4792207ce0a2b3bac5d758a5c7ae321634436fa3e6dd627a

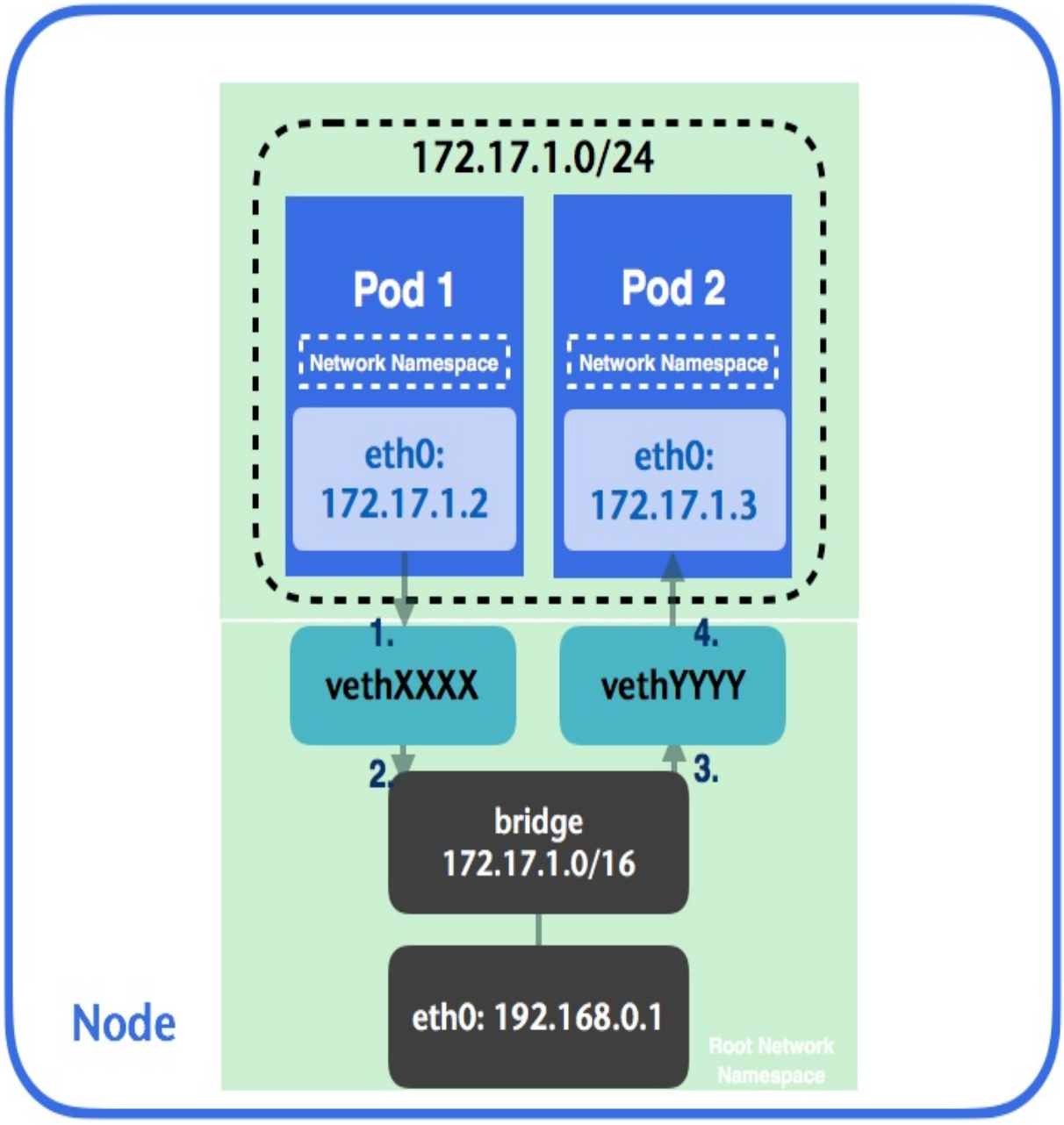
The `4ddd3221cc47` container is the so-called network container in this case. This holds the network namespace to let the `web` and `centos` containers join. Containers in the same network namespace share the same IP address and network configuration. This is the default implementation in Kubernetes for achieving container-to-container communications, which is mapped to the first requirement.

Pod-to-pod communications

Pod IP addresses are accessible from other pods, no matter which nodes they're on. This fits the second requirement. We'll describe the pods' communication within the same node and across nodes in the upcoming section.

Pod communication within the same node

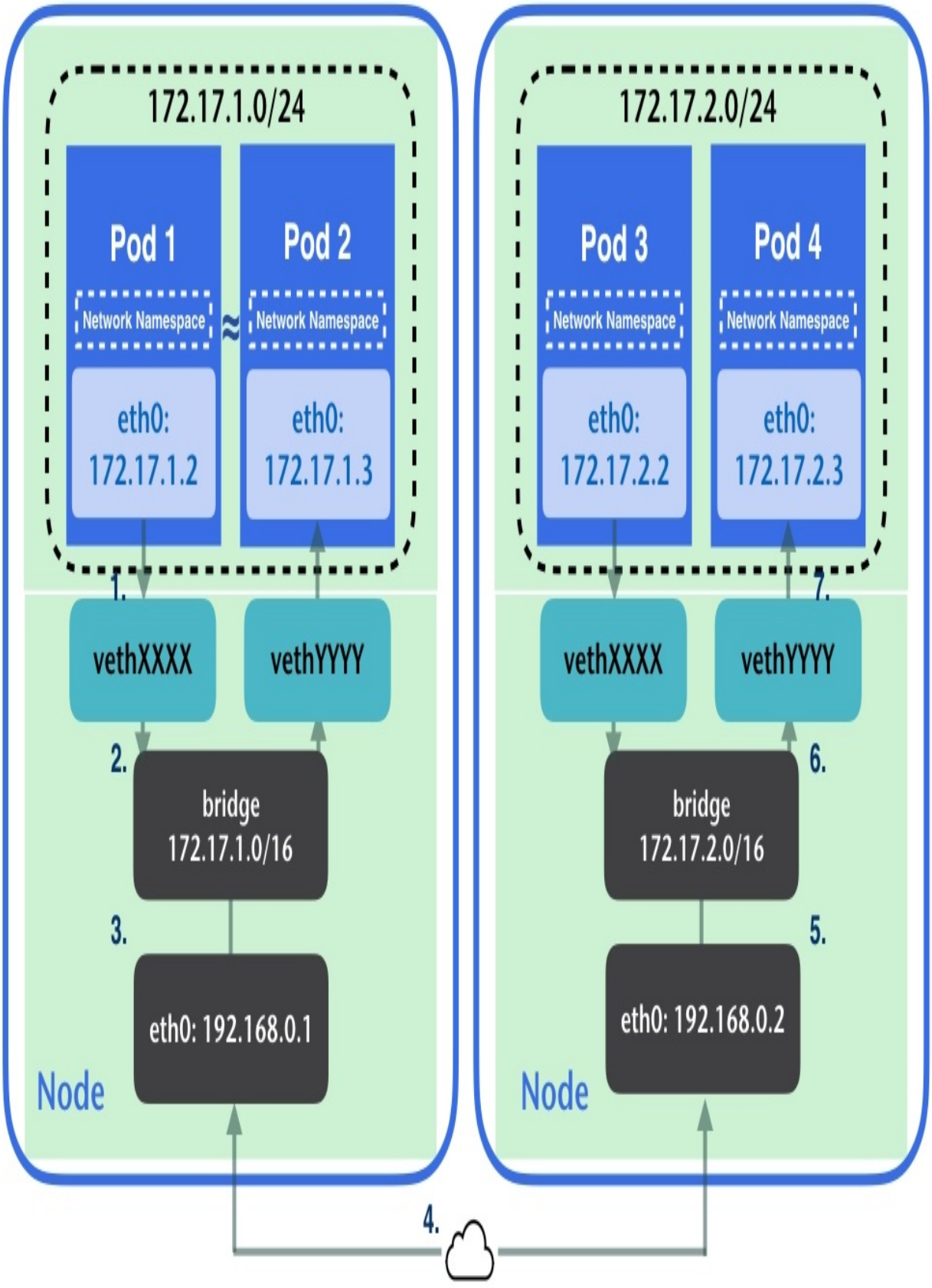
Pod-to-pod communication within the same node goes through the bridge by default. Let's say we have two pods that have their own network namespaces. When pod 1 wants to talk to pod 2, the packet passes through pod 1's namespace to the corresponding `veth` pair, **`vethXXXX`**, and eventually goes to the bridge. The bridge then broadcasts the destination IP to help the packet find its way. `vethYYYY` responds with the broadcasts. The packet then arrives at pod 2:



Now that we know how the packet travels in a single node, we will move on and talk about how traffic gets routed when the pods are in different nodes.

Pod communication across nodes

According to the second requirement, all nodes must communicate with all containers. Kubernetes delegates implementation to the **container network interface** (**CNI**). Users can choose different implementations, by L2, L3, or overlay. Overlay networking, also known as packet encapsulation, is one of the most common solutions. This wraps a message before leaving the source, delivers it, and unwraps the message at its destination. This leads to a situation in which the overlay increases the network latency and complexity. As long as all the containers can access each other across nodes, you're free to use any technology, such as L2 adjacency or the L3 gateway. For more information about CNI, refer to its spec (`https://github.com/containernetworking/cni/blob/master/SPEC.md`):


172.17.1.0/24
Pod 1
Pod 2
Network Namespace
Network Namespace
eth0:
172.17.1.2
eth0:
172.17.1.3
1.
vethXXXX
vethYYYY
2.
bridge
172.17.1.0/16
3.
eth0: 192.168.0.1
Node
172.17.2.0/24
Pod 3
Pod 4
Network Namespace
Network Namespace
eth0:
172.17.2.2
eth0:
172.17.2.3
7.
vethXXXX
vethYYYY
6.
bridge
172.17.2.0/16
5.
eth0: 192.168.0.2
Node
4.

Let's say we have a packet moving from pod 1 to pod 4. The packet leaves the container interface and reaches the `veth` pair, and then passes through the bridge and the node's network interface. Network implementation comes into play in step 4. As long as the packet can be routed to the target node, you are free to use any options. In the following example, we'll launch minikube with the `--network-plugin=cni` option. With CNI enabled, the parameters will be passed through kubelet in the node. Kubelet has a default network plugin, but you can probe any supported plugin when it starts up. Before starting `minikube`, you can use `minikube stop` first if it has been started or `minikube delete` to delete the whole cluster thoroughly before doing anything else. Although `minikube` is a single node environment that might not completely represent the production scenario we'll encounter, this just gives you a basic idea of how all of this works. We will learn about the deployment of networking options in the real world in `Chapter 9`, *Continuous Delivery*, and `Chapter 10`, *Kubernetes on AWS*: // **start minikube with cni option # minikube start --network-plugin=cni ... Loading cached images from config file.**
Everything looks great. Please enjoy minikube!

When we specify the `network-plugin` option, `minikube` will use the directory specified in `--network-plugin-dir` for plugins on startup. In the CNI plugin, the default plugin directory is `/opt/cni/net.d`. After the cluster comes up, we can log into the node and look at the network interface configuration inside the minikube via `minikube ssh`: **# minikube ssh $ ifconfig ... mybridge Link encap:Ethernet HWaddr 0A:58:0A:01:00:01 inet addr:10.1.0.1 Bcast:0.0.0.0 Mask:255.255.0.0 ...**

We will find that there is one new bridge in the node, and if we create the example pod again by using `5-1-1_pod.yml`, we will find that the IP address of the pod becomes `10.1.0.x`, which is attaching to `mybridge` instead of `docker0`: **# kubectl create -f 6-1-1_pod.yaml pod/example created # kubectl describe po example Name: example Namespace: default Node: minikube/10.0.2.15 Start Time: Sun, 23 Jul 2017 14:24:24 -0400 Labels: <none> Annotations: <none> Status: Running IP: 10.1.0.4**

This is because we have specified that we'll use CNI as the network plugin, and `docker0` will not be used (also known as the **container network model**, or **libnetwork**). The CNI creates a virtual interface, attaches it to the underlay

network, sets the IP address, and eventually routes and maps it to the pods' namespace. Let's take a look at the configuration that's located at `/etc/cni/net.d/k8s.conf` in minikube:

```
# cat /etc/cni/net.d/k8s.conf
{
  "name": "rkt.kubernetes.io",
  "type": "bridge",
  "bridge": "mybridge",
  "mtu": 1460,
  "addIf": "true",
  "isGateway": true,
  "ipMasq": true,
  "ipam": {
    "type": "host-local",
    "subnet": "10.1.0.0/16",
    "gateway": "10.1.0.1",
    "routes": [
      {
       "dst": "0.0.0.0/0"
      }
    ]
  }
}
```

In this example, we are using the bridge CNI plugin to reuse the L2 bridge for pod containers. If the packet is from `10.1.0.0/16`, and its destination is to anywhere, it'll go through this gateway. Just like the diagram we saw earlier, we could have another node with CNI enabled with the `10.1.2.0/16` subnet so that ARP packets can go out to the physical interface on the node where the target pod is located. This then achieves pod-to-pod communication across nodes.

Let's check the rules in `iptables`: **// check the rules in iptables # sudo iptables -t nat -nL ... Chain POSTROUTING (policy ACCEPT) target prot opt source destination KUBE-POSTROUTING all -- 0.0.0.0/0 0.0.0.0/0 /* kubernetes postrouting rules */ MASQUERADE all -- 172.17.0.0/16 0.0.0.0/0 CNI-25df152800e33f7b16fc085a all -- 10.1.0.0/16 0.0.0.0/0 /* name: "rkt.kubernetes.io" id: "328287949eb4d4483a3a8035d65cc326417ae7384270844e59c2f4e963d87e18" */ CNI-f1931fed74271104c4d10006 all -- 10.1.0.0/16 0.0.0.0/0 /* name: "rkt.kubernetes.io" id: "08c562ff4d67496fdae1c08facb2766ca30533552b8bd0682630f203b18f8c0a" */**

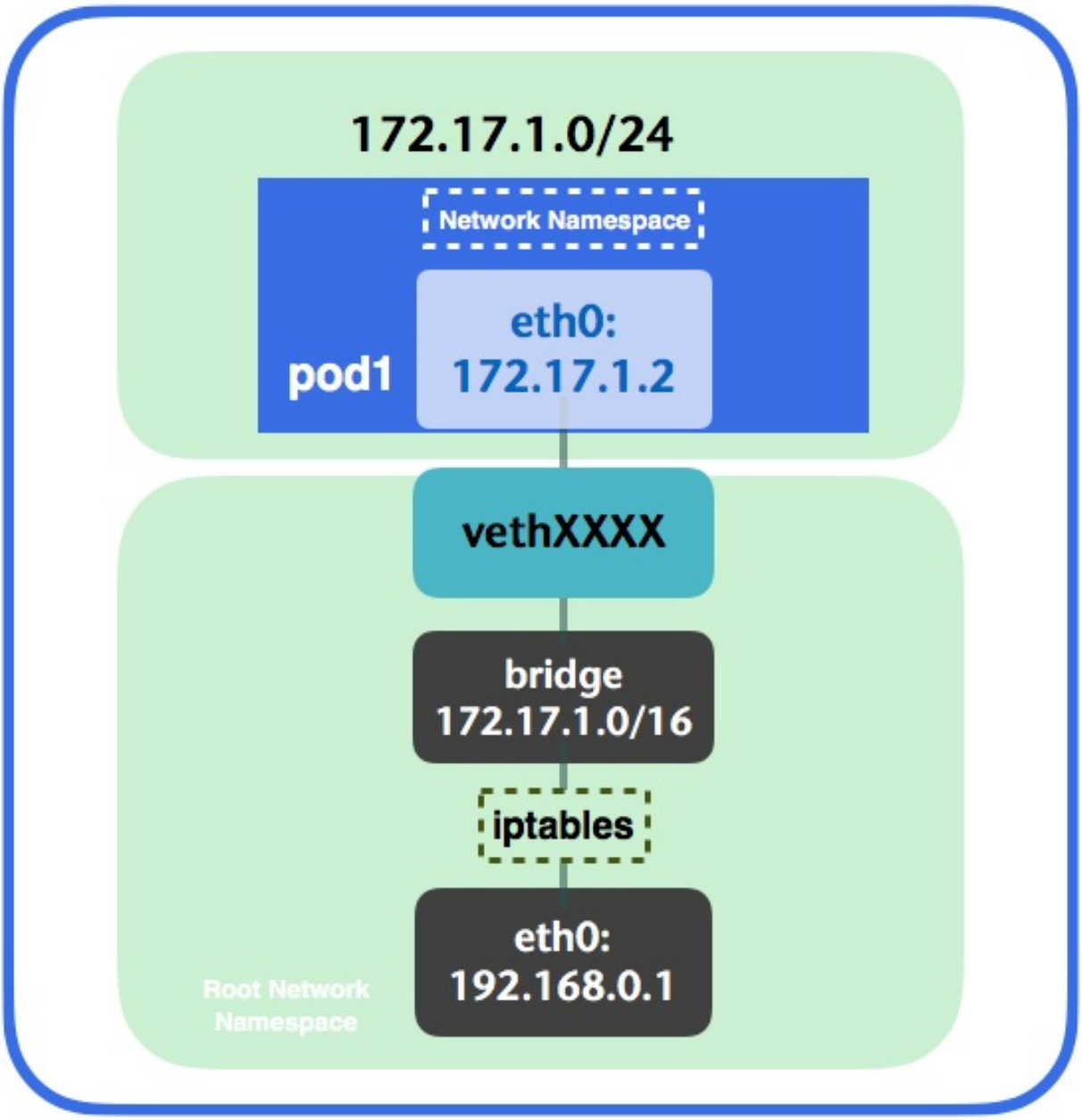
All the related rules have been switched to `10.1.0.0/16` CIDR.

Pod-to-service communications

Kubernetes is dynamic. Pods are created and deleted all the time. The Kubernetes service is an abstraction to define a set of pods by label selectors. We normally use the service to access pods instead of specifying a pod explicitly. When we create a service, an `endpoint` object will be created, which describes a set of pod IPs that the label selector in that service has selected.

In some cases, the `endpoint` object will not be created upon creation of the service. For example, services without selectors will not create a corresponding `endpoint` object. For more information, refer to the Service without selectors section in `Chapter 3`, Getting Started with Kubernetes.

So, how does traffic get from pod to the pod behind the service? By default, Kubernetes uses `iptables` to perform this magic, and does so by using `kube-proxy`. This is explained in the following diagram:



Let's reuse the `3-2-3_rc1.yaml` and `3-2-3_nodeport.yaml` examples from `Chapter 3`, *Getting Started with Kubernetes*, to observe the default behavior:

```
// create two pods with nginx and one service to observe default networking. Users are f
# kubectl create -f chapter3/3-2-3_Service/3-2-3_rs1.yaml
replicaset.apps/nginx-1.12 created
# kubectl create -f chapter3/3-2-3_Service/3-2-3_nodeport.yaml
service/nginx-nodeport created
```

Let's observe the `iptable` rules and see how this works. As you can see in the

following code, our service IP is `10.0.0.167`. The two pods' IP addresses underneath are `10.1.0.4` and `10.1.0.5`: **# kubectl describe svc nginx-nodeport Name: nginx-nodeport Namespace: default Selector: project=chapter3,service=web Type: NodePort IP: 10.0.0.167 Port: <unset> 80/TCP NodePort: <unset> 32261/TCP Endpoints: 10.1.0.4:80,10.1.0.5:80 ...**

Let's get into the minikube node by using `minikube ssh` and check its `iptables` rules: **# sudo iptables -t nat -nL ... Chain KUBE-SERVICES (2 references) target prot opt source destination KUBE-SVC-37ROJ3MK6RKFMQ2B tcp -- 0.0.0.0/0 10.0.0.167 /* default/nginx-nodeport: cluster IP */ tcp dpt:80 KUBE-NODEPORTS all -- 0.0.0.0/0 0.0.0.0/0 /* kubernetes service nodeports; NOTE: this must be the last rule in this chain */ ADDRTYPE match dst-type LOCAL Chain KUBE-SVC-37ROJ3MK6RKFMQ2B (2 references) target prot opt source destination KUBE-SEP-SVVBOHTYP7PAP3J5 all -- 0.0.0.0/0 0.0.0.0/0 /* default/nginx-nodeport: */ statistic mode random probability 0.50000000000 KUBE-SEP-AYS7I6ZPYFC6YNNF all -- 0.0.0.0/0 0.0.0.0/0 /* default/nginx-nodeport: */ Chain KUBE-SEP-SVVBOHTYP7PAP3J5 (1 references) target prot opt source destination KUBE-MARK-MASQ all -- 10.1.0.4 0.0.0.0/0 /* default/nginx-nodeport: */ DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 /* default/nginx-nodeport: */ tcp to:10.1.0.4:80 Chain KUBE-SEP-AYS7I6ZPYFC6YNNF (1 references) target prot opt source destination KUBE-MARK-MASQ all -- 10.1.0.5 0.0.0.0/0 /* default/nginx-nodeport: */ DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 /* default/nginx-nodeport: */ tcp to:10.1.0.5:80 ...**

The key point here is that the service exposes the cluster IP to outside traffic from `KUBE-SVC-37ROJ3MK6RKFMQ2B`, which links to two custom chains, `KUBE-SEP-SVVBOHTYP7PAP3J5` and `KUBE-SEP-AYS7I6ZPYFC6YNNF`, with a statistic mode random probability of 0.5. This means that `iptables` will generate a random number and tune it based on the probability distribution of 0.5 to the destination. These two custom chains have the `DNAT` target set to the corresponding pod IP. The `DNAT` target is responsible for changing the packets' destination IP address. By default, conntrack is enabled to track the destination and source of connection when the traffic comes in. All of this results in a routing behavior. When the traffic comes to the service, `iptables` will randomly pick one of the pods to route and modify the destination IP from the service IP to the real pod IP. It then gets the response and un-DNAT on the reply packets and sends them back to the requester.

IPVS-based kube-proxy:
In Kubernetes 1.11, the IPVS-based `kube-proxy` *feature graduated to GA. This could deal with the scaling problem of* `iptables` *to tens of thousands of services. The* ***IP Virtual Server (IPVS)*** *is part of the Linux kernel, which can direct TCP or UDP requests to real servers.* `ipvs` *proxier is a good fit if your application contains a huge number of services. However, it will fall back on* `iptables` *in some specific cases. For more information, please refer to* `https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/`.

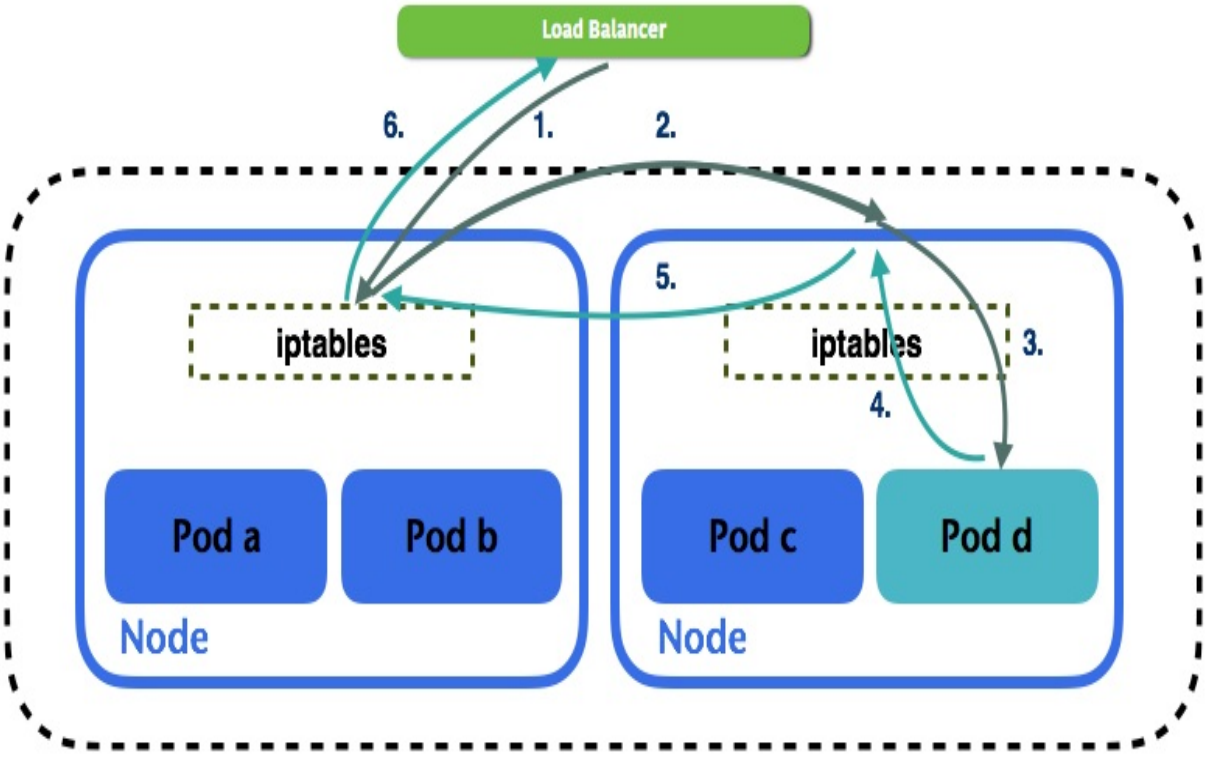
External-to-service communications

The ability to serve external traffic to Kubernetes is critical. Kubernetes provides two API objects to achieve this:

- **Service**: External network LoadBalancer or NodePort (L4)
- **Ingress**: HTTP(S) LoadBalancer (L7)

We'll learn more about ingress in the next section. For now, we'll focus on the L4 service. Based on what we've learned about pod-to-pod communication across nodes, the packet goes in and out between the service and pod. The following diagram is an illustration of this process. Let's say we have two services: service A has three pods (pod a, pod b, and pod c) and service B gets only one pod (pod d). When the traffic comes in from the LoadBalancer, the packet will be dispatched to one of the nodes. Most of the LoadBalancer cloud itself is not aware of pods or containers; it only knows about the node. If the node passes the health check, then it will be the candidate for the destination.

Let's assume that we want to access service B; this currently only has one pod running on one node. However, LoadBalancer sends the packet to another node that doesn't have any of our desired pods running. In this case, the traffic route will look like this:



The packet routing journey will be as follows:

1. LoadBalancer will choose one of the nodes to forward to the packet. In GCE, it selects the instance based on a hash of the source IP and port, destination IP and port, and protocol. In AWS, load balancing is based on a round-robin algorithm.
2. Here, the routing destination will be changed to pod d (DNAT) and will forward the packet to the other node, similar to pod-to-pod communication across nodes.
3. Then comes service-to-pod communication. The packet arrives at pod d and pod d returns the response.
4. Pod-to-service communication is manipulated by `iptables` as well.
5. The packet will be forwarded to the original node.
6. The source and destination will be un-DNAT to the LoadBalancer and client, and will be sent all the way back to the requester.

From Kubernetes 1.7, there is a new attribute in this service called `externalTrafficPolicy`. Here, you can set its value to local, and then, after the traffic goes into a node, Kubernetes will route the pods on that node if possible, as follows:

```
kubectl patch $service_name nodeport -p '{"spec":{"externalTrafficPolicy":"Local"}}'
```

// start over our minikube local
minikube delete && minikube start

// enable ingress in minikube
minikube addons enable ingress ingress was successfully enabled

// check current setting for addons in minikube # minikube addons list

- registry: disabled

- registry-creds: disabled

- addon-manager: enabled

- dashboard: enabled

- default-storageclass: enabled - kube-dns: enabled

- heapster: disabled

- ingress: enabled

cat chapter6/6-2-1_nginx.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 2
<span

xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.50.1">
template:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.51.1">
metadata:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.52.1">
labels:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.53.1">
project: chapter6
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.54.1">
service: nginx
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.55.1">
spec:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.56.1">
containers:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.57.1"> -
name: nginx
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.58.1">
image: nginx
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.59.1">
ports:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.60.1"> -
containerPort: 80
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.61.1">---

<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.62.1">kind: Service
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.63.1">apiVersion: v1
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.64.1">metadata:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.65.1">
name: nginx
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.66.1">spec:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.67.1">
type: NodePort
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.68.1">
selector:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.69.1">
project: chapter6
<span

xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.70.1">
service: nginx
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.71.1">
ports:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.72.1"> -
protocol: TCP
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.73.1">
port: 80
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.74.1">
targetPort: 80
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.75.1">//
create nginx RS and service
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.76.1">#
kubectl create -f chapter6/6-2-1_nginx.yaml
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.77.1">deployment.apps/nginx created
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.78.1">service/nginx created

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.82.1">// another backend named echoserver
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.83.1">#
cat chapter6/6-2-1_echoserver.yaml apiVersion: apps/v1

kind: Deployment

metadata:

name: echoserver spec:

replicas: 1

template: metadata: name: echoserver labels: project: chapter6

service: echoserver spec:

containers: - name: echoserver image: gcr.io/google_containers/echoserver:1.4

ports: - containerPort: 8080

---

kind: Service

apiVersion: v1

metadata:

name: echoserver spec:

type: NodePort selector: project: chapter6

service: echoserver ports: - protocol: TCP

port: 8080

targetPort: 8080

// create RS and SVC by above configuration file # kubectl create -f chapter6/6-2-1_echoserver.yaml deployment.apps/echoserver created
service/echoserver created

cat chapter6/6-2-1_ingress.yaml apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: ingress-example
 annotations:
 nginx.ingress.kubernetes.io/rewrite-target: /
spec:
<span

xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.103.1">
rules:
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.104.1"> - host: devops.k8s
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.105.1">
http:
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.106.1"> paths:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.107.1"> -
path: /welcome
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.108.1"> backend:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.109.1">
serviceName: nginx
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.110.1"> servicePort: 80
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.111.1"> -
path: /echoserver
<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.112.1"> backend:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.113.1">
serviceName: echoserver
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.114.1">
servicePort: 8080

// create ingress

kubectl create -f chapter6/6-2-1_ingress.yaml ingress.extensions/ingress-example created

// normally host file located in /etc/hosts in linux # sudo sh -c "echo `minikube ip` devops.k8s >> /etc/hosts"

curl http://devops.k8s/welcome ...

<title>Welcome to nginx!</title> ...

// check echoserver # curl http://devops.k8s/echoserver CLIENT VALUES:

client_address=172.17.0.4

command=GET

real path=/

query=nil

request_version=1.1

request_uri=http://devops.k8s:8080/

The pod ingress controller dispatches the traffic based on the URL's path. The routing path is similar to external-to-service communication. The packet hops between nodes and pods. Kubernetes is pluggable; lots of third-party implementation is going on. We have only scratched the surface here, even though `iptables` is just a default and common implementation. Networking evolves a lot with every single release. At the time of writing, Kubernetes had just released version 1.13.

Network policy

The network policy works as a software firewall to the pods. By default, every pod can communicate with each other without any boundaries. The network policy is one of the isolations you could apply to these pods. This defines who can access which pods in which port by namespace selector and pod selector. The network policy in a namespace is additive, and once a pod enables the network policy, it denies any other ingress (also known as deny all).

Currently, there are multiple network providers that support the network policy, such as Calico (`https://www.projectcalico.org/calico-network-policy-comes-to-kubernetes/`), Romana (`https://github.com/romana/romana`), Weave Net (`https://www.weave.works/docs/net/latest/kube-addon/#npc`), Contiv (`http://contiv.github.io/documents/networking/policies.html`), and Trireme (`https://github.com/aporeto-inc/trireme-kubernetes`). Users are free to choose between any of these. For the purpose of simplicity, though, we're going to use Calico with minikube. To do that, we'll have to launch minikube with the `--network-plugin=cni` option. The network policy is still pretty new in Kubernetes at this point. We're running Kubernetes version v.1.7.0 with the v.1.0.7 minikube ISO to deploy Calico by self-hosted solution (`http://docs.projectcalico.org/v1.5/getting-started/kubernetes/installation/hosted/`). Calico can be installed with etcd datastore or the Kubernetes API datastore. For convenience, we'll demonstrate how to install Calico with the Kubernetes API datastore here. Since rbac is enabled in minikube, we'll have to configure the roles and bindings for Calico: **# kubectl apply -f **
https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/rbac-kdd.yaml
clusterrole.rbac.authorization.k8s.io/calico-node configured
clusterrolebinding.rbac.authorization.k8s.io/calico-node configured

Now, let's deploy Calico:

```
# kubectl apply -f https://docs.projectcalico.org/v3.3/getting-started/kubernetes/instal
configmap/calico-config created
service/calico-typha created
deployment.apps/calico-typha created
poddisruptionbudget.policy/calico-typha created
daemonset.extensions/calico-node created
serviceaccount/calico-node created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico....
```

```
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org crea
```

After doing this, we can list the Calico pods and see whether it's launched successfully: **# kubectl get pods --namespace=kube-system**
NAME READY STATUS RESTARTS AGE
calico-node-ctxq8 2/2 Running 0 14m

Let's reuse `6-2-1_nginx.yaml` for our example: **# kubectl create -f chapter6/6-2-1_nginx.yaml replicaset "nginx" created service "nginx" created // list the services # kubectl get svc**
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.96.0.1 <none> 443/TCP 36m
nginx NodePort 10.96.51.143 <none> 80:31452/TCP 5s

We will find that our `nginx` service has an IP address of `10.96.51.143`. Let's launch a simple bash and use `wget` to see whether we can access our `nginx`: **# kubectl run busybox -i -t --image=busybox /bin/sh If you don't see a command prompt, try pressing enter. / # wget --spider 10.96.51.143**
Connecting to 10.96.51.143 (10.96.51.143:80)

The `--spider` parameter is used to check whether the URL exists. In this case, `busybox` can access `nginx` successfully. Next, let's apply a `NetworkPolicy` to our `nginx` pods: **// declare a network policy # cat chapter6/6-3-1_networkpolicy.yaml**
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
name: nginx-networkpolicy
spec:
podSelector:
matchLabels:
service: nginx
ingress:
- from:
- podSelector:
matchLabels:
project: chapter6

We can see some important syntax here. The `podSelector` is used to select pods that

should match the labels of the target pod. Another one is `ingress[].from[].podSelector`, which is used to define who can access these pods. In this case, all the pods with `project=chapter6` labels are eligible to access the pods with `server=nginx` labels. If we go back to our busybox pod, we're unable to contact `nginx` any more because, right now, the `nginx` pod has `NetworkPolicy` on it.

By default, it is deny all, so busybox won't be able to talk to `nginx`: **// in busybox pod, or you could use `kubectl attach <pod_name> -c busybox -i -t` to re-attach to the pod # wget --spider --timeout=1 10.96.51.143 Connecting to 10.96.51.143 (10.96.51.143:80) wget: download timed out**

We can use `kubectl edit deployment busybox` to add the `project=chapter6` label in busybox pods.

After that, we can contact the `nginx` pod again: **// inside busybox pod / # wget --spider 10.96.51.143 Connecting to 10.96.51.143 (10.96.51.143:80)**

With the help of the preceding example, we now have an idea of how to apply a network policy. We could also apply some default polices to deny all, or allow all, by tweaking the selector to select nobody or everybody. For example, the deny all behavior can be achieved as follows: **# cat chapter6/6-3-1_np_denyall.yaml apiVersion: networking.k8s.io/v1**
kind: NetworkPolicy
metadata:
name: default-deny
spec:
podSelector: {}
policyTypes:
- Ingress

This way, all pods that don't match labels will deny all other traffic. Alternatively, we could create a `NetworkPolicy` whose ingress is listed everywhere. By doing this, the pods running in this namespace can be accessed by anyone: **# cat chapter6/6-3-1_np_allowall.yaml apiVersion: networking.k8s.io/v1**
kind: NetworkPolicy
metadata:
name: allow-all
spec:

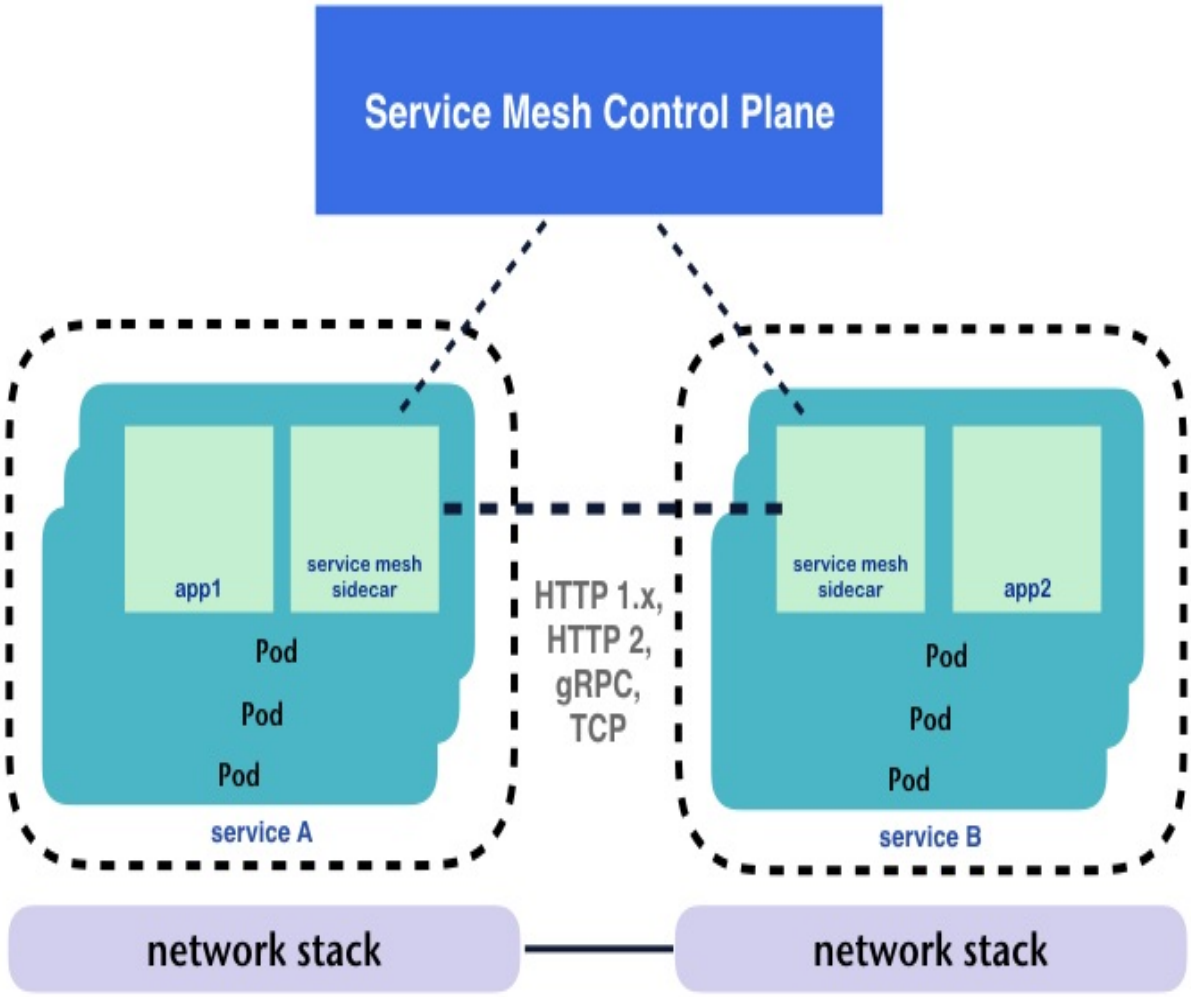
podSelector: {}
ingress:
- {}

Service mesh

A service mesh is an infrastructure layer for handling service-to-service communication. Especially in the microservice world, the application at hand might contain hundreds of thousands of services. The network topology can be very complicated here. A service mesh can provide the following:

- Traffic management (such as A/B testing and canary deployment)
- Security (such as TLS and key management)
- Observability (such as providing traffic visibility. This is easy to integrate with monitoring systems such as Prometheus (https://prometheus.io/), tracing systems such as Jaeger (https://www.jaegertracing.io) or Zipkin (https://github.com/openzipkin/zipkin), and logging systems)

There are two major service mesh implementations on the market—Istio (https://istio.io) and Linkerd (https://linkerd.io). Both of these deploy network proxy containers alongside the application container (the so-called sidecar container) and provide Kubernetes support. The following diagram is a simplified common architecture of the service mesh:



A service mesh normally contains a control plane, which is the brain of the mesh. This can manage and enforce the policies for route traffic, as well as collect telemetry data that can be integrated with other systems. It also carries out identity and credential management for services or end users. The service mesh sidecar container, which acts as a network proxy, lives side by side with the application container. The communication between services is passed through the sidecar container, which means that it can control the traffic by user-defined policies, secure the traffic via TLS encryption, do the load balancing and retries, control the ingress/egress, collect the metrics, and so on.

In the following section, we'll use Istio as the example, but you're free to use any implementation in your organization. First, let's get the latest version of Istio. At the time of writing, the latest version is 1.0.5:

```
// get the latest istio
# curl -L https://git.io/getLatestIstio | sh -
Downloading istio-1.0.5 from https://github.com/istio/istio/releases/download/1.0.5/isti

// get into the folder
# cd istio-1.0.5/
```

Next, let's create a **Custom Resource Definition** (**CRD**) for Istio: **# kubectl apply -f install/kubernetes/helm/istio/templates/crds.yaml**
customresourcedefinition.apiextensions.k8s.io/virtualservices.networking.isti
created
customresourcedefinition.apiextensions.k8s.io/destinationrules.networking.is
created
customresourcedefinition.apiextensions.k8s.io/serviceentries.networking.istio
created
customresourcedefinition.apiextensions.k8s.io/gateways.networking.istio.io
created
...

In the following example, we're installing Istio with default mutual TLS authentication. The resource definition is under the `install/kubernetes/istio-demo-auth.yaml` file. If you'd like to deploy it without TLS authentication, you can use `install/kubernetes/istio-demo.yaml` instead:

```
# kubectl apply -f install/kubernetes/istio-demo-auth.yaml
namespace/istio-system created
configmap/istio-galley-configuration created
...
```

```
kubernetes.config.istio.io/attributes created
destinationrule.networking.istio.io/istio-policy created
destinationrule.networking.istio.io/istio-telemetry created
```

After deployment, let's check that the services and pods have all been deployed successfully into the `istio-system` namespace: **// check services are launched successfully**
 # kubectl get svc -n istio-system
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
grafana ClusterIP 10.98.182.66 <none> 3000/TCP 13s
istio-citadel ClusterIP 10.105.65.6 <none> 8060/TCP,9093/TCP 13s
istio-egressgateway ClusterIP 10.105.178.212 <none> 80/TCP,443/TCP 13s
istio-galley ClusterIP 10.103.123.213 <none> 443/TCP,9093/TCP 13s
istio-ingressgateway LoadBalancer 10.107.243.112 <pending>
80:31380/TCP,443:31390/TCP,31400:31400/TCP,15011:32320/TCP,8060:317

13s
istio-pilot ClusterIP 10.104.123.60 <none>
15010/TCP,15011/TCP,8080/TCP,9093/TCP 13s
istio-policy ClusterIP 10.111.227.237 <none>
9091/TCP,15004/TCP,9093/TCP 13s
istio-sidecar-injector ClusterIP 10.107.43.206 <none> 443/TCP 13s
istio-telemetry ClusterIP 10.103.118.119 <none>
9091/TCP,15004/TCP,9093/TCP,42422/TCP 13s
jaeger-agent ClusterIP None <none> 5775/UDP,6831/UDP,6832/UDP 11s
jaeger-collector ClusterIP 10.110.234.134 <none> 14267/TCP,14268/TCP 11s
jaeger-query ClusterIP 10.103.19.74 <none> 16686/TCP 12s
prometheus ClusterIP 10.96.62.77 <none> 9090/TCP 13s
servicegraph ClusterIP 10.100.191.216 <none> 8088/TCP 13s
tracing ClusterIP 10.107.99.50 <none> 80/TCP 11s
zipkin ClusterIP 10.98.206.168 <none> 9411/TCP 11s

After waiting for a few minutes, check that the pods are all in `Running` and `Completed` states, as follows: **# kubectl get pods -n istio-system**
NAME READY STATUS RESTARTS AGE
grafana-7ffdd5fb74-hzwcn 1/1 Running 0 5m1s
istio-citadel-55cdfdd57c-zzs2s 1/1 Running 0 5m1s
istio-cleanup-secrets-qhbvk 0/1 Completed 0 5m3s
istio-egressgateway-687499c95f-fbbwq 1/1 Running 0 5m1s
istio-galley-76bbb946c8-9mw2g 1/1 Running 0 5m1s
istio-grafana-post-install-8xxps 0/1 Completed 0 5m3s
istio-ingressgateway-54f5457d68-n7xsj 1/1 Running 0 5m1s
istio-pilot-7bf5674b9f-jnnvx 2/2 Running 0 5m1s
istio-policy-75dfcf6f6d-nwvdn 2/2 Running 0 5m1s
istio-security-post-install-stv2c 0/1 Completed 0 5m3s
istio-sidecar-injector-9c6698858-gr86p 1/1 Running 0 5m1s
istio-telemetry-67f94c555b-4mt4l 2/2 Running 0 5m1s
istio-tracing-6445d6dbbf-8r5r4 1/1 Running 0 5m1s
prometheus-65d6f6b6c-qrp6f 1/1 Running 0 5m1s
servicegraph-5c6f47859-qzlml 1/1 Running 2 5m1s

Since we have `istio-sidecar-injector` deployed, we can simply use `kubectl label namespace default istio-injection=enabled` to enable the sidecar container injection for

every pod in the `default` namespace. `istio-sidecar-injector` acts as a mutating admission controller, which will inject the sidecar container to the pod if the namespace is labelled with `istio-injection=enabled`. Next, we can launch a sample application from the `samples` folder. Helloworld demonstrates the use of canary deployment (`https://en.wikipedia.org/wiki/Deployment_environment`), which will distribute the traffic to the helloworld-v1 and helloworld-v2 services: **// launch sample application**
 # kubectl run nginx --image=nginx
deployment.apps/nginx created

// list pods
 # kubectl get po
NAME READY STATUS RESTARTS AGE
nginx-64f497f8fd-b7d4k 2/2 Running 0 3s

If we inspect one of the pods, we'll find that the `istio-proxy` container was injected into it: **# kubectl describe po nginx-64f497f8fd-b7d4k**
Name: nginx-64f497f8fd-b7d4k
Namespace: default
Labels: pod-template-hash=2090539498
run=nginx
Annotations: kubernetes.io/limit-ranger: LimitRanger plugin set: cpu request for container nginx
sidecar.istio.io/status:
{"version":"50128f63e7b050c58e1cdce95b577358054109ad2aff4bc4995158c0
["istio-init"],"containers":["istio-proxy"]...
Status: Running
Init Containers:
istio-init:
Container ID:
docker://3ec33c4cbc66682f9a6846ae6f310808da3a2a600b3d107a0d361b5deb
Image: docker.io/istio/proxy_init:1.0.5
...
Containers:
nginx:
Container ID:
docker://42ab7df7366c1838489be0c7264a91235d8e5d79510f3d0f078726165e5

Image: nginx
...
istio-proxy:
Container ID:
docker://7bdf7b82ce3678174dea12fafd2c7f0726bfffc562ed3505a69991b06cf3
Image: docker.io/istio/proxyv2:1.0.5
Image ID: docker-
pullable://istio/proxyv2@sha256:8b7d549100638a3697886e549c149fb588800
Port: 15090/TCP
Host Port: 0/TCP
Args:
proxy
sidecar
--configPath
/etc/istio/proxy
--binaryPath
/usr/local/bin/envoy
--serviceCluster
istio-proxy
--drainDuration
45s
--parentShutdownDuration
1m0s
--discoveryAddress
istio-pilot.istio-system:15005
--discoveryRefreshDelay
1s
--zipkinAddress
zipkin.istio-system:9411
--connectTimeout
10s
--proxyAdminPort
15000
--controlPlaneAuthPolicy
MUTUAL_TLS

Taking a closer look, we can see that the `istio-proxy` container was launched with

the configuration of its control plane address, tracing system address, and connection configuration. Istio has now been verified. There are lots of to do with Istio traffic management, which is beyond of the scope of this book. Istio has a variety of detailed samples for us to try, which can be found in the `istio-1.0.5/samples` folder that we just downloaded.

Summary

In this chapter, we learned how containers communicate with each other. We also introduced how pod-to-pod communication works. A service is an abstraction that routes traffic to any of the pods underneath it if the label selectors match. We also learned how a service works with a pod using `iptables`. We also familiarized ourselves with how packet routes from external services to a pod using DNAT and un-DAT packets. In addition to this, we looked at new API objects such as ingress, which allows us to use the URL path to route to different services in the backend. In the end, another `NetworkPolicy` object was introduced. This provides a second layer of security, and acts as a software firewall rule. With the network policy, we can make certain pods communicate with certain other pods. For example, only data retrieval services can talk to the database container. In the last section, we got a glimpse at Istio, one of the popular implementations of service mesh. All of these things make Kubernetes more flexible, secure, robust, and powerful.

Before this chapter, we covered the basic concepts of Kubernetes. In `Chapter 7`, *Monitoring and Logging*, we'll get a clearer understanding of what is happening inside your cluster by monitoring cluster metrics and analyzing applications and system logs for Kubernetes. Monitoring and logging tools are essential for every DevOps, which also plays an extremely important role in dynamic clusters such as Kubernetes. Consequently, we'll get an insight into the activities of the cluster, such as scheduling, deployment, scaling, and service discovery. `Chapter 7`, *Monitoring and Logging*, will help you to better understand the act of operating Kubernetes in the real world.

Monitoring and Logging

Monitoring and logging are crucial parts of a site's reliability. So far, we've learned how to use various controllers to take care of our application. We have also looked at how to utilize services together with Ingress to serve our web applications, both internally and externally. In this chapter, we'll gain more visibility over our applications by looking at the following topics:

- Getting a status snapshot of a container
- Monitoring in Kubernetes
- Converging metrics from Kubernetes with Prometheus
- Various concepts to do with logging in Kubernetes
- Logging with Fluentd and Elasticsearch
- Gaining insights into traffic between services using Istio

Inspecting a container

Whenever our application behaves abnormally, we need to figure out what has happened with our system. We can do this by checking logs, resource usage, a watchdog, or even getting into the running host directly to dig out problems. In Kubernetes, we have `kubectl get` and `kubectl describe`, which can query controller states about our deployments. This helps us determine whether an application has crashed or whether it is working as desired.

If we want to know what is going on using the output of our application, we also have `kubectl logs`, which redirects a container's `stdout` and `stderr` to our Terminal. For CPU and memory usage stats, there's also a `top`-like command we can employ, which is `kubectl top`. `kubectl top node` gives an overview of the resource usage of nodes, while `kubectl top pod <POD_NAME>` displays per-pod usage: **$ kubectl top node**
NAME CPU(cores) CPU% MEMORY(bytes) MEMORY%
node-1 31m 3% 340Mi 57%
node-2 24m 2% 303Mi 51%
$ kubectl top pod mypod-6cc9b4bc67-j6zds
NAME CPU(cores) MEMORY(bytes)
mypod-6cc9b4bc67-j6zds 0m 0Mi

To use `kubectl top`, you'll need the metrics-server or Heapster (if you're using Kubernetes prior to 1.13) deployed in your cluster. We'll discuss this later in the chapter.

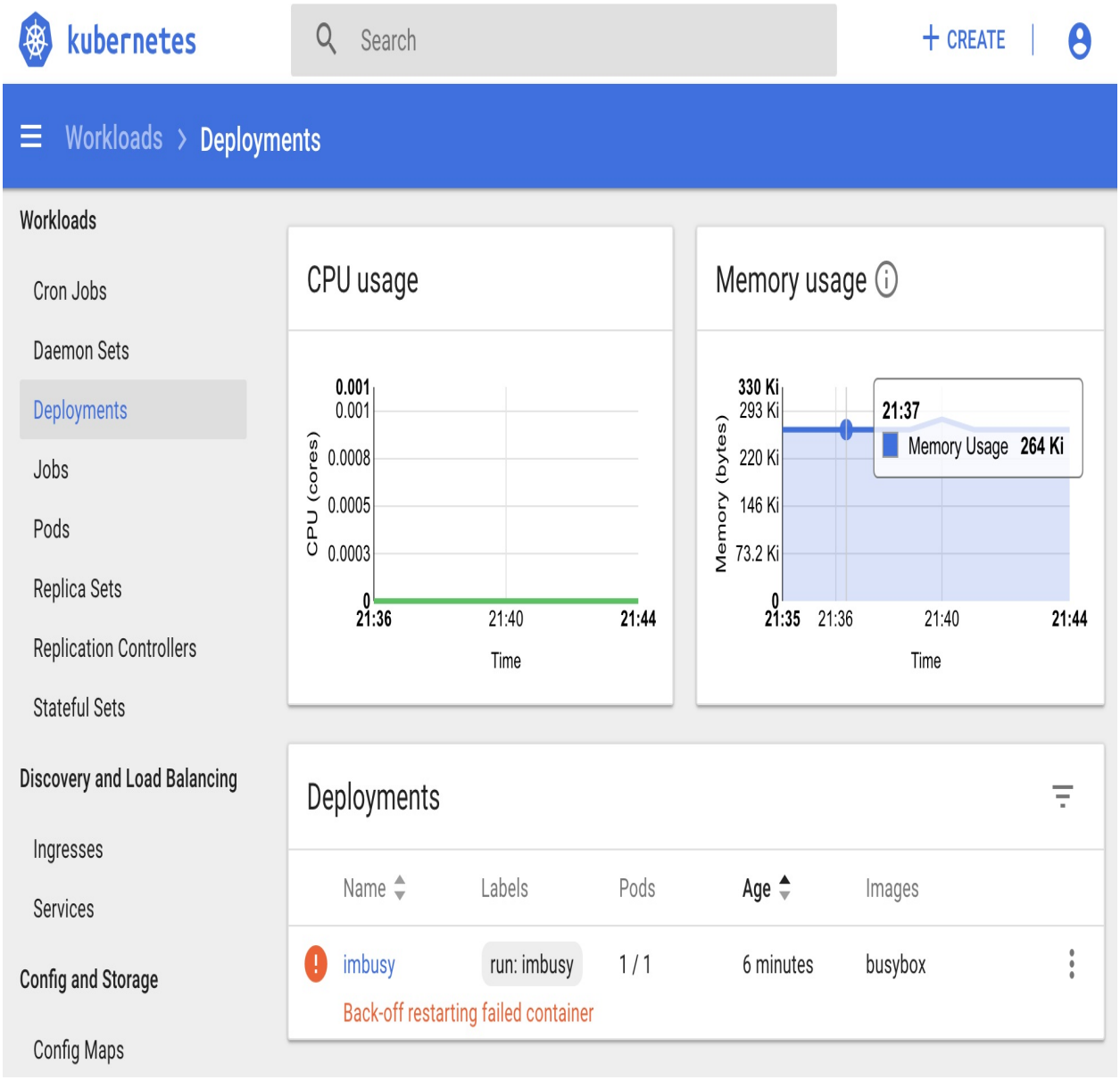
What if we leave something such as logs inside a container and they are not sent out anywhere? We know that there's a `docker exec` execute command inside a running container, but it's unlikely that we will have access to nodes every time. Fortunately, `kubectl` allows us to do the same thing with the `kubectl exec` command. Its usage is similar to `docker exec`. For example, we can run a shell inside the container in a pod as follows: **$ kubectl exec -it mypod-6cc9b4bc67-j6zds /bin/sh**
/ #
/ # hostname
mypod-6cc9b4bc67-j6zds

This is pretty much the same as logging onto a host via SSH. It enables us to troubleshoot with tools we are familiar with, as we've done previously without containers.

If the container is built by `FROM scratch`, the `kubectl exec` trick may not work well because the core utilities such as the shell (`sh` or `bash`) might not be present inside the container. Before ephemeral containers, there was no official support for this problem. If we happen to have a `tar` binary inside our running container, we can use `kubectl cp` to copy some binaries into the container in order to carry out troubleshooting. If we're lucky and we have privileged access to the node the container runs on, we can utilize `docker cp`, which doesn't require a `tar` binary inside the container, to move the utilities we need into the container.

The Kubernetes dashboard

In addition to the command-line utility, there is a dashboard that aggregates almost all the information we just discussed and displays the data in a decent web UI:



This is actually a general purpose graphical user interface of a Kubernetes cluster as it also allows us to create, edit, and delete resources. Deploying it is

quite easy; all we need to do is apply a template:

```
$ kubectl create -f \ https://raw.githubusercontent.com/kubernetes/dashboard/v1.10.0/src
```

Many managed Kubernetes services, such as **Google Kubernetes Engine** (**GKE**), provide an option to pre-deploy a dashboard in the cluster so that we don't need to install it ourselves. To determine whether the dashboard exists in our cluster or not, use `kubectl cluster-info`. If it's installed, we'll see the message `kubernetes-dashboard is running at ...` as shown in the following:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.64.32:8443
CoreDNS is running at https://192.168.64.32:8443/api/v1/namespaces/kube-system/services/
kubernetes-dashboard is running at https://192.168.64.32:8443/api/v1/namespaces/kube-sys


To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

The service for the dashboard deployed with the preceding default template or provisioned by cloud providers is usually `ClusterIP`. We've learned a bunch of ways to access a service inside a cluster, but here let's just use the simplest built-in proxy, `kubectl proxy`, to establish the connection between our Terminal and our Kubernetes API server. Once the proxy is up, we are then able to access the dashboard at `http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/`. Port `8001` is the default port of the `kubectl proxy` command.

*The dashboard deployed with the previous template wouldn't be one of the services listed in the output of `kubectl cluster-info` as it's not managed by the **addon manager**. The addon manager ensures that the objects it manages are active, and it's enabled in most managed Kubernetes services in order to protect the cluster components. Take a look at the following repository for more information: https://github.com/kubernetes/kubernetes/tree/master/cluster/addons/addon-manager.*

The methods to authenticate to the dashboard vary between cluster setups. For example, the token that allows `kubectl` to access a GKE cluster can also be used to log in to the dashboard. It can either be found in `kubeconfig`, or obtained via the one-liner shown in the following (supposing the current context is the one in use):

```
$ kubectl config view --minify -o \
 jsonpath={.users[].user.auth-provider.config.access-token}
```

If we skip the sign in, the service account for the dashboard would be used

instead. For other access options, check the wiki page of the dashboard's project to choose one that suits your cluster setup: https://github.com/kubernetes/dashboard/wiki/Access-control#authentication.

As with `kubectl top`, to display the CPU and memory stats, you'll need a metric server deployed in your cluster.

Monitoring in Kubernetes

We now know how to examine our applications in Kubernetes. However, we are not yet confident enough to answer more complex questions, such as how healthy our application is, what changes have been made to the CPU usage from the new patch, when our databases will run out of capacity, and why our site rejects any requests. We therefore need a monitoring system to collect metrics from various sources, store and analyze the data received, and then respond to exceptions. In a classical setup of a monitoring system, we would gather metrics from at least three different sources to measure our service's availability, as well as its quality.

Monitoring applications

The data we are concerned with relates to the internal states of our running application. Collecting this data gives us more information about what's going on inside our service. The data may be to do with the goal the application is designed to achieve, or the runtime data intrinsic to the application. Obtaining this data often requires us to manually instruct our program to expose the internal data to the monitoring pipeline because only we, the service owner, know what data is meaningful, and also because it's often hard to get information such as the size of records in the memory for a cache service externally.

The ways in which applications interact with the monitoring system differ significantly. For example, if we need data about the statistics of a MySQL database, we could set an agent that periodically queries the information and performance schema for the raw data, such as numbers of SQL queries accumulated at the time, and transform them to the format for our monitoring system. In a Golang application, as another example, we might expose the runtime information via the `expvar` package and its interface and then find another way to ship the information to our monitoring backend. To alleviate the potential difficulty of these steps, the **OpenMetrics** (`https://openmetrics.io/`) project endeavours to provide a standardized format for exchanging telemetry between different applications and monitoring systems.

In addition to time series metrics, we may also want to use profiling tools in conjunction with tracing tools to assert the performance of our program. This is especially important nowadays, as an application might be composed of dozens of services in a distributed way. Without utilizing tracing tools such as **OpenTracing** (`http://opentracing.io`), identifying the reasons behind performance declines can be extremely difficult.

Monitoring infrastructure

The term infrastructure may be too broad here, but if we simply consider where our application runs and how it interacts with other components and users, it is obvious what we should monitor: the application hosts and the connecting network.

As collecting tasks at the host is a common practice for system monitoring, it is usually performed by agents provided by the monitoring framework. The agent extracts and sends out comprehensive metrics about a host, such as loads, disks, connections, or other process statistics that help us determine the health of a host.

For the network, these can be merely the web server software and the network interface on the same host, plus perhaps a load balancer, or even within a platform such as Istio. Although the way to collect telemetry data about the previously mentioned components depends on their actual setup, in general, the metrics we'd like to measure would be traffic, latency, and errors.

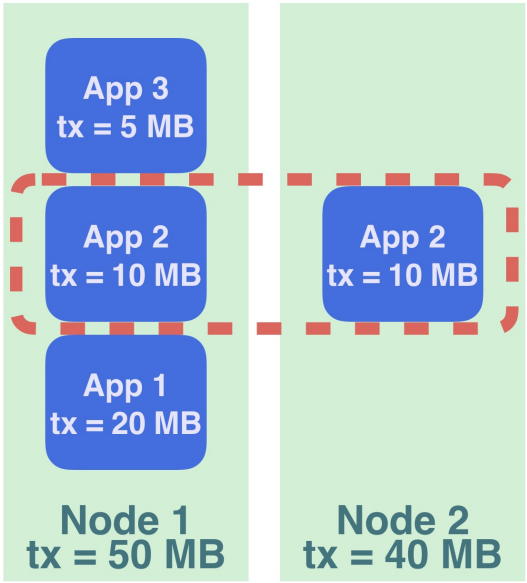
Monitoring external dependencies

Aside from the aforementioned two components, we also need to check the statuses of dependent components, such as the utilization of external storage, or the consumption rate of a queue. For instance, let's say we have an application that subscribes to a queue as an input and executes tasks from that queue. In this case, we'd also need to consider metrics such as the queue length and the consumption rate. If the consumption rate is low and the queue length keeps growing, our application may have trouble.

These principles also apply to containers on Kubernetes, as running a container on a host is almost identical to running a process. However, because of the subtle distinction between the way in which containers on Kubernetes and on traditional hosts utilize resources, we still need to adjust our monitoring strategy accordingly. For instance, containers of an application on Kubernetes would be spread across multiple hosts and would not always be on the same hosts. It would be difficult to produce a consistent recording of one application if we are still adopting a host-centric monitoring approach. Therefore, rather than observing resource usage at the host only, we should add a container layer to our monitoring stack. Moreover, since Kubernetes is the infrastructure for our applications, it is important to take this into account as well.

Monitoring containers

As a container is basically a thin wrapper around our program and dependent runtime libraries, the metrics collected at the container level would be similar to the metrics we get at the container host, particularly with regard to the use of system resources. Although collecting these metrics from both the containers and their hosts might seem redundant, it actually allows us to solve problems related to monitoring moving containers. The idea is quite simple: what we need to do is attach logical information to metrics, such as pod labels or their controller names. In this way, metrics coming from containers across distinct hosts can be grouped meaningfully. Consider the following diagram. Let's say we want to know how many bytes were transmitted (**tx**) on **App 2**. We could add up the **tx** metrics that have the **App 2** label, which would give us a total of **20 MB**:



tx of App 2 on all nodes:
sum(tx{app="App 2"}) = 20 MB

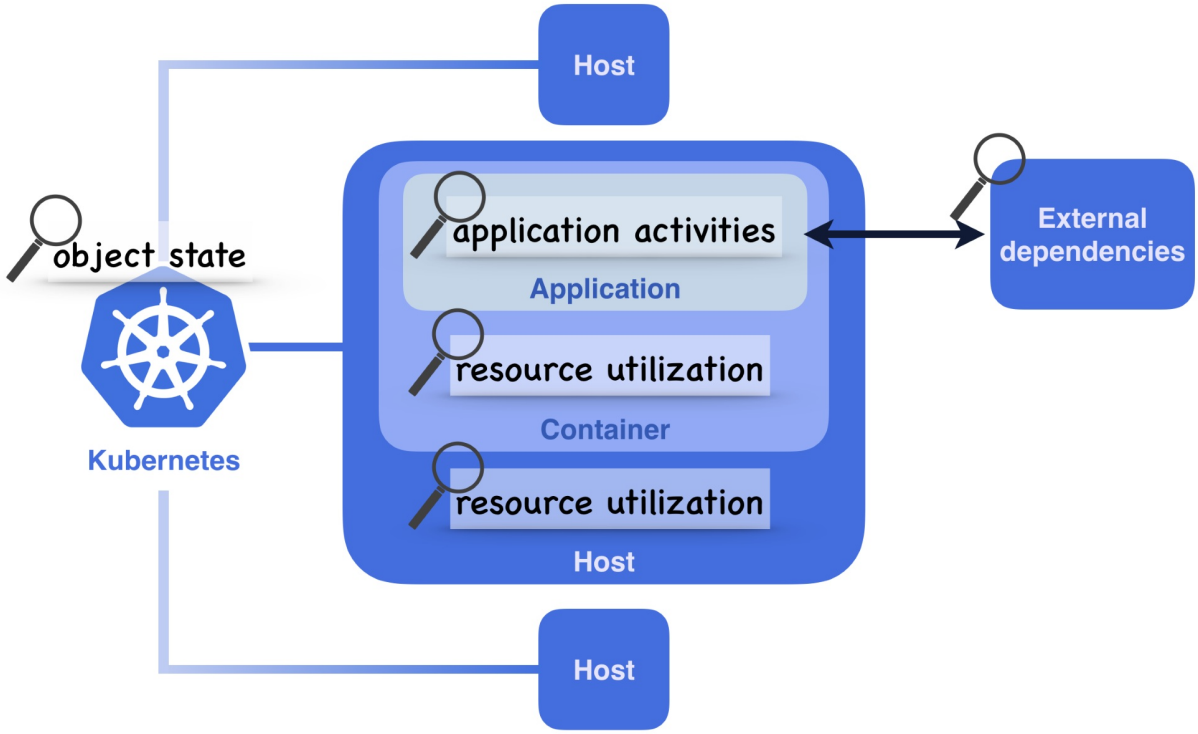
Another difference is that metrics related to CPU throttling are reported at the container level only. If performance issues are encountered in a certain application but the CPU resource on the host is spare, we can check if it's

throttled with the associated metrics.

Monitoring Kubernetes

Kubernetes is responsible for managing, scheduling, and orchestrating our applications. Once an application has crashed, Kubernetes is one of the first places we would like to look at. In particular, when a crash happens after rolling out a new deployment, the state of the associated objects would be reflected instantly on Kubernetes.

To sum up, the components that should be monitored are illustrated in the following diagram:



Getting monitoring essentials for Kubernetes

Since monitoring is an important part of operating a service, the existing monitoring system in our infrastructure might already provide solutions for collecting metrics from common sources like well-known open source software and the operating system. As for applications run on Kubernetes, let's have a look at what Kubernetes and its ecosystem offer.

To collect metrics of containers managed by Kubernetes, we don't have to install any special controller on the Kubernetes master node, nor any metrics collector inside our containers. This is basically done by kubelet, which gathers various telemetries from a node, and exposes them in the following API endpoints (as of Kubernetes 1.13):

- `/metrics/cadvisor`: This API endpoint is used for cAdvisor container metrics that are in Prometheus format
- `/spec/`: This API endpoint exports machine specifications
- `/stats/`: This API endpoint also exports cAdvisor container metrics but in JSON format
- `/stats/summary`: This endpoint contains various data aggregated by kubelet. It's also known as the Summary API

The metrics under the bare path `/metrics/` relate to kubelet's internal statistics.

The Prometheus format (`https://prometheus.io/docs/instrumenting/exposition_formats/`) is the predecessor of the OpenMetrics format, so it is also known as OpenMetrics v0.0.4 after OpenMetrics was published. If our monitoring system supports this kind of format, we can configure it to pull metrics from kubelet's Prometheus endpoint (`/metrics/cadvisor`).

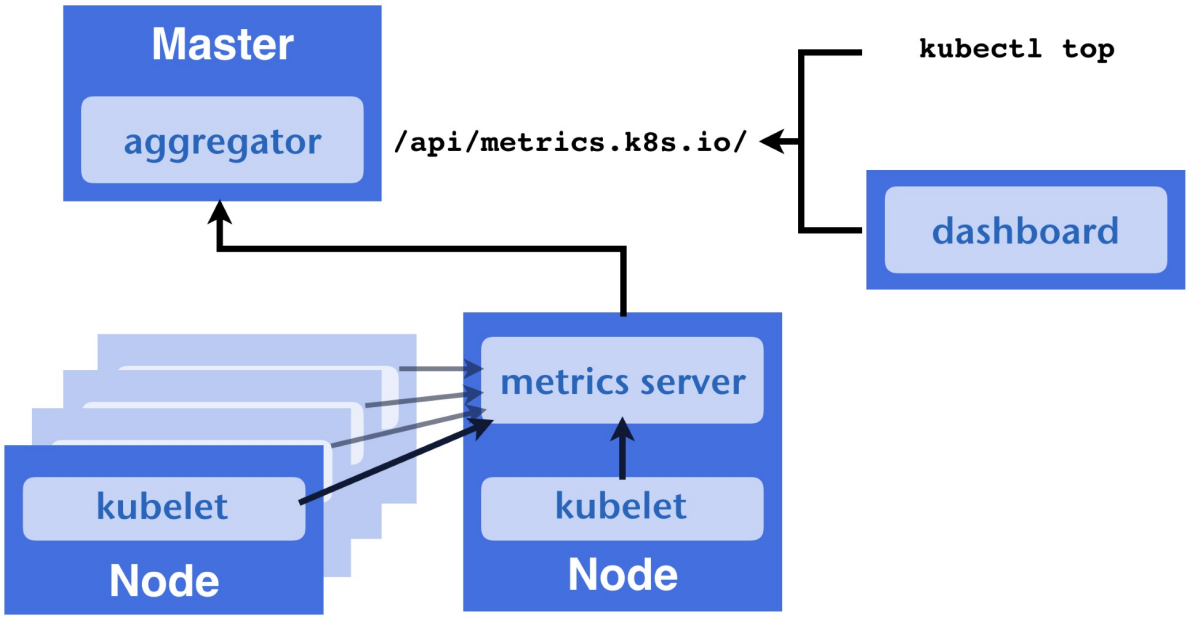
To access those endpoints, kubelet has two TCP ports, `10250` and `10255`. Port `10250` is the safer one and the one that it is recommended to use in production as it's an HTTPS endpoint and protected by Kubernetes' authentication and authorization

system. `10255` is in plain HTTP, which should be used restrictively.

cAdvisor (`https://github.com/google/cadvisor`) is a widely used container-level metrics collector. Put simply, cAdvisor aggregates the resource usage and performance statistics of every container running on a machine. Its code is currently sold inside kubelet, so we don't need to deploy it separately. However, since it focuses on certain container runtimes and Linux containers only, which may not suit future Kubernetes releases for different container runtimes, there won't be an integrated cAdvisor in future releases of Kubernetes. In addition to this, not all cAdvisor metrics are currently published by kubelet. Therefore, if we need that data, we'll need to deploy cAdvisor by ourselves. Notice that the deployment of cAdvisor is one per host instead of one per container, which is more reasonable for containerized applications, and we can use DaemonSet to deploy it.

Another important component in the monitoring pipeline is the metrics server (`https://github.com/kubernetes-incubator/metrics-server`). This aggregates monitoring statistics from the summary API by kubelet on each node and acts as an abstraction layer between Kubernetes' other components and the real metrics sources. To be more specific, the metrics server implements the resource metrics API under the aggregation layer, so other intra-cluster components can get the data from a unified API path (`/api/metrics.k8s.io`). In this instance, `kubectl top` and kube-dashboard get data from the resource metrics API.

The following diagram illustrates how the metrics server interacts with other components in a cluster:



If you're using an older version of Kubernetes, the role of the metrics server will be played by Heapster (`https://github.com/kubernetes/heapster`).

Most installations of Kubernetes deploy the metrics server by default. If we need to do this manually, we can download the manifest of the metrics server and apply them:

```
$ git clone https://github.com/kubernetes-incubator/metrics-server.git
$ kubectl apply -f metrics-server/deploy/1.8+/
```

While kubelet metrics are focused on system metrics, we also want to see the logical states of objects displayed on our monitoring dashboard. `kube-state-metrics` (https://github.com/kubernetes/kube-state-metrics) is the piece that completes our monitoring stack. It watches Kubernetes masters and transforms the object statuses we see from `kubectl get` or `kubectl describe` into metrics in the Prometheus format. We are therefore able to scrape the states into metrics storage and then be alerted on events such as unexplainable restart counts. Download the templates to install as follows:

```
$ git clone https://github.com/kubernetes/kube-state-metrics.git
$ kubectl apply -f kube-state-metrics/kubernetes
```

Afterward, we can view the state metrics from the `kube-state-metrics` service inside our cluster:

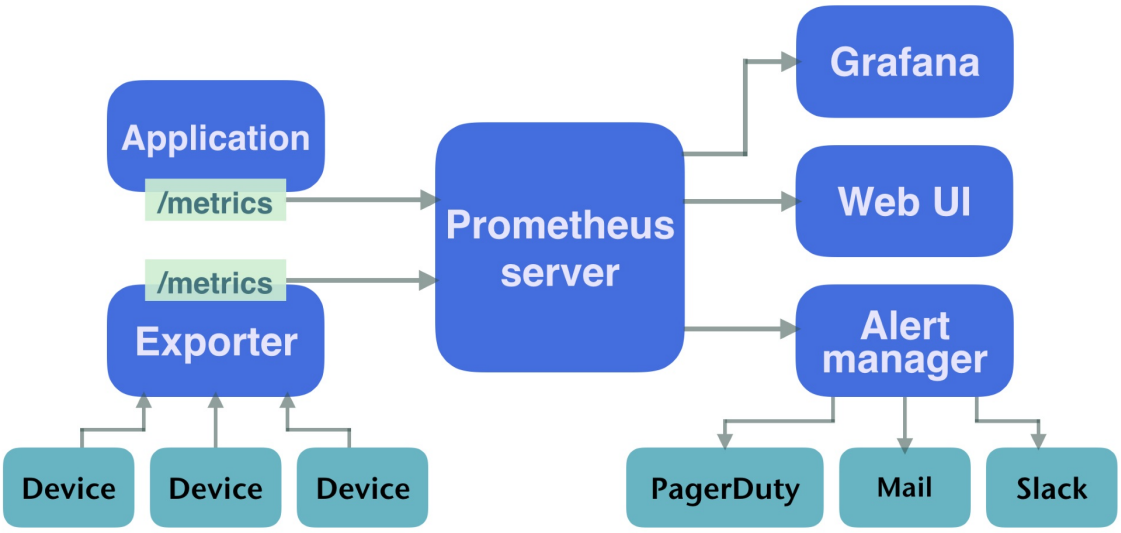
```
http://kube-state-metrics.kube-system:8080/metrics
```

Hands-on monitoring

So far, we've learned about a wide range of principles that are required to create an impervious monitoring system in Kubernetes, which allows us to build a robust service. It's time to implement one. Because the vast majority of Kubernetes components expose their instrumented metrics on a conventional path in Prometheus format, we are free to use any monitoring tool with which we are acquainted, as long as the tool understands the format. In this section, we'll set up an example with Prometheus. Its popularity in the Kubernetes ecosystem is not only due to its power, but also for its backing by the **Cloud Native Computing Foundation** (`https://www.cncf.io/`), which also sponsors the Kubernetes project.

Getting to know Prometheus

The Prometheus framework is made up of several components, as illustrated in the following diagram:



As with all other monitoring frameworks, Prometheus relies on agents scraping statistics from the components of our system. Those agents are the exporters shown to the left of the diagram. Besides this, Prometheus adopts the pull model of metric collection, which is to say that it does not receive metrics passively, but actively pulls data from the metrics' endpoints on the exporters. If an application exposes a metric's endpoint, Prometheus is able to scrape that data as well. The default storage backend is an embedded TSDB, and can be switched to other remote storage types such as InfluxDB or Graphite. Prometheus is also responsible for triggering alerts according to preconfigured rules in **Alertmanager**, which handles alarm tasks. It groups alarms received and dispatches them to tools that actually send messages, such as email, **Slack** (`https://slack.com/`), **PagerDuty** (`https://www.pagerduty.com/`), and so on. In addition to alerts, we also want to visualize the collected metrics to get a quick overview of our system, which is where Grafana comes in handy.

Aside from collecting data, alerting is one of the most important concepts to do with monitoring. However, alerting is more relevant to business concerns, which is out of the scope of this chapter. Therefore, in this section, we'll focus on

metric collection with Prometheus and won't look any closer at Alertmanager.

Deploying Prometheus

The templates we've prepared for this chapter can be found at the following link: https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter7.

Under `7-1_prometheus` are the manifests of components to be used for this section, including a Prometheus deployment, exporters, and related resources. These will be deployed at a dedicated namespace, `monitoring`, except those components required to work in `kube-system` namespaces. Please review them carefully. For now, let's create our resources in the following order: **$ kubectl apply -f monitoring-ns.yml**
$ kubectl apply -f prometheus/configs/prom-config-default.yml
$ kubectl apply -f prometheus

The resource usage, such as storage and memory, at the provided manifest for Prometheus is confined to a relatively low level. If you'd like to use them in a more realistic way, you can adjust your parameters according to your actual requirements. After the Prometheus server is up, we can connect to its web UI at port `9090` with `kubectl port-forward`. We can also use NodePort or Ingress to connect to the UI if we modify its service (`prometheus/prom-svc.yml`) accordingly. The first page we will see when entering the UI is the Prometheus expression browser, where we build queries and visualize metrics. Under the default settings, Prometheus will collect metrics by itself. All valid scraping targets can be found at the `/targets` path. To speak to Prometheus, we have to gain some understanding of its language: **PromQL**.

*To run Prometheus in production, there is also a **Prometheus Operator** (https://github.com/coreos/prometheus-operator), which aims to simplify the monitoring task in Kubernetes by CoreOS.*

sum(container_memory_usage_bytes{namespace="kube-system", pod_name=~"kube-dns-([^-]+)-.*"}) / 1048576

More detailed documentation can be found at the official Prometheus site (https://prometheus.io/docs/querying/basics/). This will help you to unleash the power of Prometheus.

Discovering targets in Kubernetes

Since Prometheus only pulls metrics from endpoints it knows, we have to explicitly tell it where we'd like to collect data from. Under the `/config` path is a page that lists the current configured targets to pull. By default, there would be one job that runs against Prometheus itself, and this can be found in the conventional scraping path, `/metrics`. If we are connecting to the endpoint, we would see a very long text page, as shown in the following:

```
$ kubectl exec -n monitoring <prometheus_pod_name> -- \
    wget -qO - localhost:9090/metrics

# HELP go_gc_duration_seconds A summary of the GC invocation durations.
# TYPE go_gc_duration_seconds summary
go_gc_duration_seconds{quantile="0"} 1.5657e-05
go_gc_duration_seconds{quantile="0.25"} 1.9756e-05
go_gc_duration_seconds{quantile="0.5"} 2.4567e-05
go_gc_duration_seconds{quantile="0.75"} 2.8386e-05
...
```

This is the Prometheus metrics format we've mentioned several times before. Next time we see a page like this, we will know that it's a metrics endpoint. The job to scrape Prometheus is a static target in the default configuration file. However, due to the fact that containers in Kubernetes are created and destroyed dynamically, it is really difficult to find out the exact address of a container, let alone set it in Prometheus. In some cases, we may utilize the service DNS as a static metrics target, but this still cannot solve all cases. For instance, if we'd like to know how many requests are coming to each pod behind a service individually, setting a job to scrape the service might get a result from random pods instead of from all of them. Fortunately, Prometheus helps us overcome this problem with its ability to discover services inside Kubernetes.

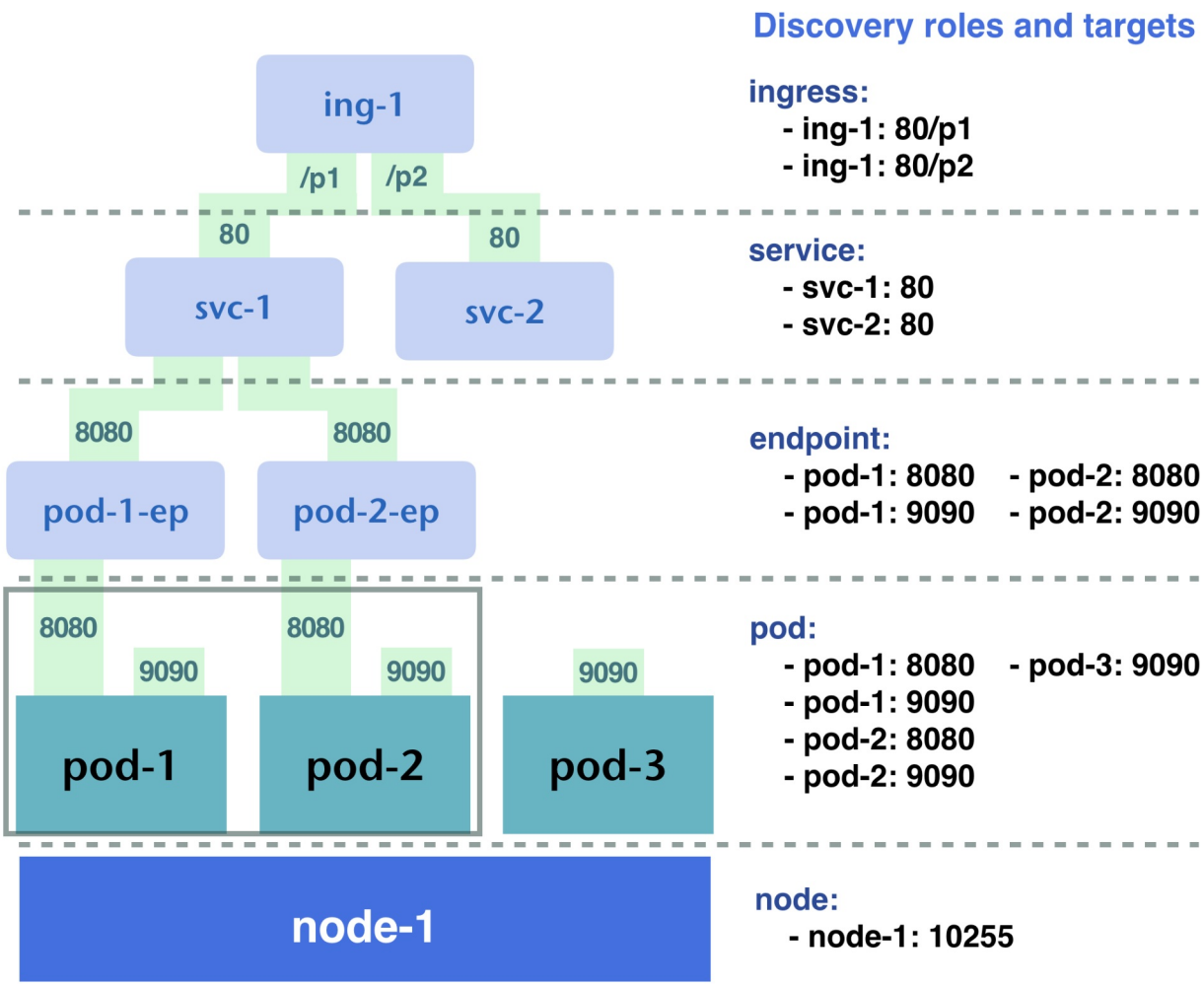
To be more specific, Prometheus is able to query Kubernetes about the information of running services. It can then add them to or delete them from the target configuration accordingly. Five discovery mechanisms are currently supported:

- The **node** discovery mode creates one target per node. The target port would be kubelet's HTTPS port (`10250`) by default.
- The **service** discovery mode creates a target for every `Service` object. All

defined target ports in a service would become a scraping target.

- The **pod** discovery mode works in a similar way to the service discovery role; it creates a target per pod and it exposes all the defined container ports for each pod. If there is no port defined in a pod's template, it would still create a scraping target with its address only.
- The **endpoints** mode discovers the `Endpoint` objects created by a service. For example, if a service is backed by three pods with two ports each, we'll have six scraping targets. In addition, for a pod, not only ports that expose to a service, but also other declared container ports would be discovered.
- The **ingress** mode creates one target per Ingress path. As an Ingress object can route requests to more than one service, and each service might have own metrics set, this mode allows us to configure all those targets at once.

The following diagram illustrates four discovery mechanisms. The left-hand ones are the resources in Kubernetes, and those on the right are the targets created in Prometheus:



Generally speaking, not all exposed ports are served as a metrics endpoint, so we certainly don't want Prometheus to grab everything it discovers in our cluster, but instead to only collect marked resources. To achieve this in Prometheus, a conventional method is to utilize annotations on resource manifests to distinguish which targets are to be grabbed, and then we can filter out those non-annotated targets using the `relabel` module in the Prometheus configuration. Consider this example configuration:

```
...

kubernetes_sd_configs:
- role: pod
  relabel_configs:
  - source_labels: [__meta_kubernetes_pod_annotation_mycom_io_scrape]
    action: keep
    regex: true
...
```

This tells Prometheus to keep only targets with the `__meta_kubernetes_pod_annotation_{name}` label and the value `true`. The label is fetched from the annotation field on the pod's specification, as shown in the following snippet:

```
...
apiVersion: apps/v1
kind: Deployment
spec:
  template:
    metadata:
      annotations:
        mycom.io/scrape: "true"
...
```

Note that Prometheus would translate every character that is not in the range `[a-zA-Z0-9_]` to `_`, so we can also write the previous annotation as `mycom-io-scrape: "true"`.

By combining those annotations and the label filtering rule, we can precisely control the targets that need to be collected. Some commonly-used annotations in Prometheus are listed as follows:

- `prometheus.io/scrape: "true"`
- `prometheus.io/path: "/metrics"`
- `prometheus.io/port: "9090"`
- `prometheus.io/scheme: "https"`
- `prometheus.io/probe: "true"`

Those annotations can be seen at `Deployment` objects (for their pods) and `Service` objects. The following template snippet shows a common use case:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    prometheus.io/scrape: "true"
    prometheus.io/path: "/monitoring"
    prometheus.io/scheme: "http"
    prometheus.io/port: "9090"
```

By applying the following configuration, Prometheus will translate the discovered target in endpoints mode into `http://<pod_ip_of_the_service>:9090/monitoring`:

```
- job_name: 'kubernetes-endpoints'
```

```
kubernetes_sd_configs:
- role: endpoints
relabel_configs:
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scrape]
  action: keep
  regex: true
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_path]
  action: replace
  target_label: __metrics_path__
  regex: (.+)
- source_labels: [__address__,__meta_kubernetes_service_annotation_prometheus_io_port]
  action: replace
  regex: ([^:]+)(?::\d+)?;(\d+)
  replacement: $1:$2
  target_label: __address__
- source_labels: [__meta_kubernetes_service_annotation_prometheus_io_scheme]
  action: replace
  target_label: __scheme__
  regex: (https?)
```

We can use the `prometheus.io/probe` annotation in Prometheus to denote whether a service should be added to the probing target or not. The probing task would be executed by the Blackbox exporter (`https://github.com/prometheus/blackbox_exporter`).

The purpose of probing is to determine the quality of connectivity between a probe and the target service. The availability of the target service would also be evaluated, as a probe could act as a customer. Because of this, where we put the probes is also a thing that should be taken into consideration if we want the probing to be meaningful.

Occasionally, we might want the metrics from any single pod under a service, not from all pods of a service. Since most endpoint objects are not created manually, the endpoint discovery mode uses the annotations inherited from a service. This means that if we annotate a service, the annotation will be visible in both the service discovery and endpoint discovery modes simultaneously, which prevents us from distinguishing whether the targets should be scraped per endpoint or per service. To solve this problem, we could use `prometheus.io/scrape: "true"` to denote endpoints that are to be scraped, and use another annotation like `prometheus.io/scrape_service_only: "true"` to tell Prometheus to create exactly one target for this service.

The `prom-config-k8s.yml` template under our example repository contains some basic configurations to discover Kubernetes resources for Prometheus. Apply it as follows:

```
$ kubectl apply -f prometheus/configs/prom-config-k8s.yml
```

Because the resource in the template is a ConfigMap, which stores data in the `etcd` consensus storage, it takes a few seconds to become consistent. Afterward,

we can reload Prometheus by sending a `SIGHUP` to the process:

```
$ kubectl exec -n monitoring <prometheus_pod_name> -- kill -1 1
```

The provided template is based on this example from Prometheus' official repository. You can find out further uses at the following link, which also includes the target discovery for the Blackbox exporter: https://github.com/prometheus/prometheus/blob/master/documentation/examples/prometheus-kubernetes.yml. We have also passed over the details of how the actions in the configuration actually work; to find out more, consult the official documentation: https://prometheus.io/docs/prometheus/latest/configuration/configuration/#configuration.

Gathering data from Kubernetes

The steps for implementing the monitoring layers discussed previously in Prometheus are now quite clear:

1. Install the exporters
2. Annotate them with appropriate tags
3. Collect them on auto-discovered endpoints

The host layer monitoring in Prometheus is done by the node exporter (`https://github.com/prometheus/node_exporter`). Its Kubernetes template can be found under the examples for this chapter, and it contains one DaemonSet with a scrape annotation. Install it as follows: **$ kubectl apply -f exporters/prom-node-exporter.yml**

Its corresponding target in Prometheus will be discovered and created by the pod discovery role if using the example configuration.

The container layer collector should be kubelet. Consequently, discovering it with the node mode is the only thing we need to do.

Kubernetes monitoring is done by `kube-state-metrics`, which was also introduced previously. It also comes with Prometheus annotations, which means we don't need to do anything else to configure it.

At this point, we've already set up a strong monitoring stack based on Prometheus. With respect to the application and the external resource monitoring, there are extensive exporters in the Prometheus ecosystem to support the monitoring of various components inside our system. For instance, if we need statistics on our MySQL database, we could just install MySQL Server Exporter (`https://github.com/prometheus/mysqld_exporter`), which offers comprehensive and useful metrics.

In addition to the metrics that we have already described, there are some other useful metrics from Kubernetes components that play an important role:

- **Kubernetes API server**: The API server exposes its stats at `/metrics`, and this target is enabled by default.
- `kube-controller-manager`: This component exposes metrics on port `10252`, but it's invisible on some managed Kubernetes services such as GKE. If you're on a self-hosted cluster, applying `kubernetes/self/kube-controller-manager-metrics-svc.yml` creates endpoints for Prometheus.
- `kube-scheduler`: This uses port `10251`, and it's also not visible on clusters by GKE. `kubernetes/self/kube-scheduler-metrics-svc.yml` is the template for creating a target to Prometheus.

- `kube-dns`: DNS in Kubernetes is managed by CoreDNS, which exposes its stats at port `9153`. The corresponding template is `kubernetes/self/ core-dns-metrics-svc.yml`.
- `etcd`: The `etcd` cluster also has a Prometheus metrics endpoint on port `2379`. If your `etcd` cluster is self-hosted and managed by Kubernetes, you can use `kubernetes/self/etcd-server.yml` as a reference.
- **Nginx ingress controller**: The nginx controller publishes metrics at port `10254`, and will give you rich information about the state of nginx, as well as the duration, size, method, and status code of traffic routed by nginx. A full guide can be found here: https://github.com/kubernetes/ingress-nginx/blob/master/docs/user-guide/monitoring.md.

The DNS in Kubernetes is served by `skydns` and it also has a metrics path exposed on the container. The typical setup in a `kube-dns` pod using `skydns` has two containers, `dnsmasq` and `sky-dns`, and their metrics ports are `10054` and `10055` respectively. The corresponding template is `kubernetes/self/ skydns-metrics-svc.yml` if we need it.

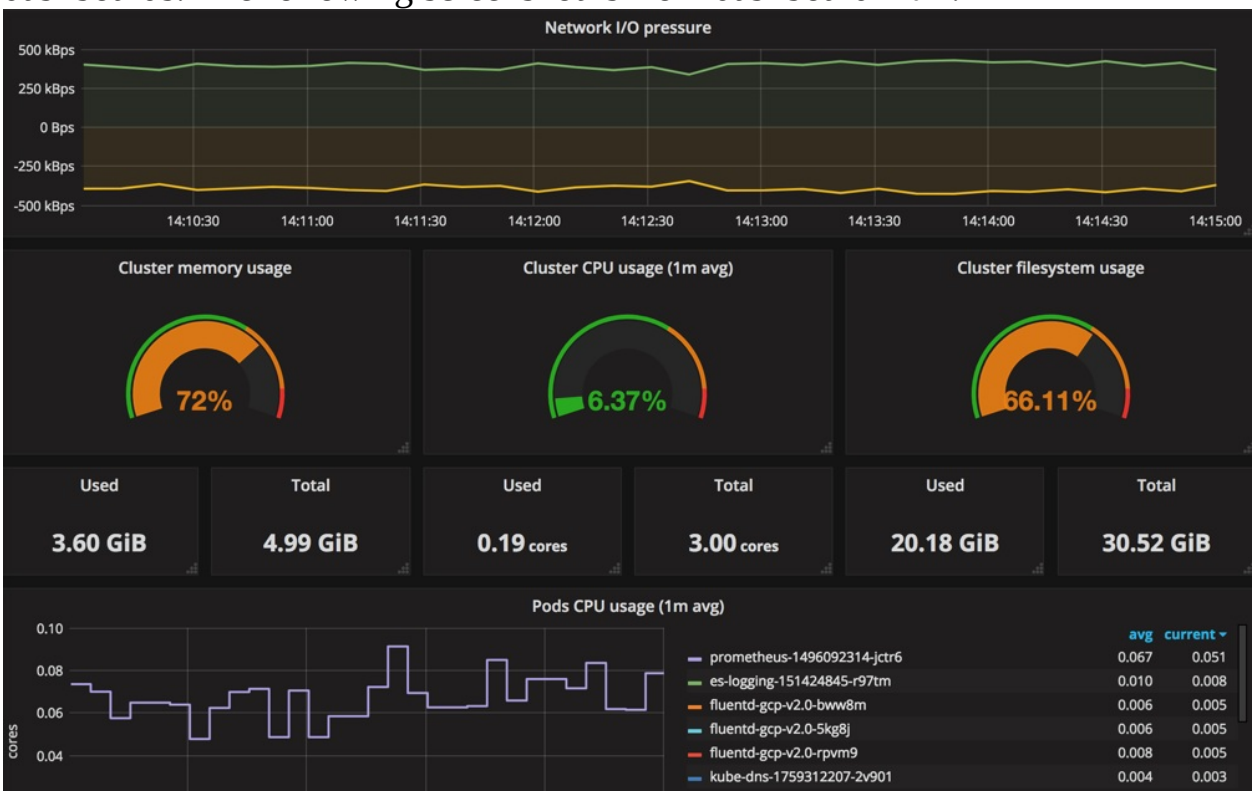
Visualizing metrics with Grafana

The expression browser has a built-in graph panel that enables us to see the metrics, but it's not designed to serve as a visualization dashboard for daily routines. Grafana is the best option for Prometheus. We discussed how to set up Grafana in `Chapter 4`, *Managing Stateful Workloads*, and we also provided templates in the repository for this chapter.

To see Prometheus metrics in Grafana, we first have to add a data source. The following configurations are required to connect to our Prometheus server:

- **Type**: `Prometheus`
- **URL**: `http://prometheus-svc.monitoring:9090`

Once it's connected, we can import a dashboard. On Grafana's sharing page (`https://grafana.com/dashboards?dataSource=prometheus`), we can find rich off-the-shelf dashboards. The following screenshot is from dashboard `#1621`:



Because the graphs are drawn by data from Prometheus, we are capable of

plotting any data we want, as long as we master PromQL.

The content of a dashboard might vary significantly as every application focuses on different things. It is not a good idea, however, to put everything into one huge dashboard. The USE method (http://www.brendangregg.com/usemethod.html) and the four golden signals (https://landing.google.com/sre/book/chapters/monitoring-distributed-systems.html) provide a good start for building a monitoring dashboard.

Logging events

Monitoring with a quantitative time series of the system status enables us to quickly identify which components in our system have failed, but it still isn't capable of diagnosing the root cause of a problem. What we need is a logging system that gathers, persists, and searches logs, by means of correlating events with the anomalies detected. Surely, in addition to troubleshooting and postmortem analysis of system failures, there are also various business use cases that need a logging system.

In general, there are two main components in a logging system: the logging agent and the logging backend. The former is an abstract layer of a program. It gathers, transforms, and dispatches logs to the logging backend. A logging backend warehouses all logs received. As with monitoring, the most challenging part of building a logging system for Kubernetes is determining how to gather logs from containers to a centralized logging backend. Typically, there are three ways to send out the logs of a program:

- Dumping everything to `stdout`/`stderr`.
- Writing log files to the filesystem.
- Sending logs to a logging agent or logging to the backend directly. Programs in Kubernetes are also able to emit logs in the same manner, so long as we understand how log streams flow in Kubernetes.

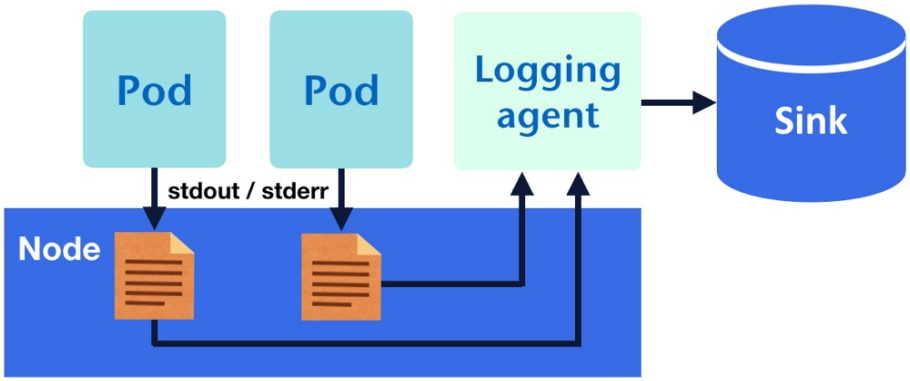
Patterns of aggregating logs

For programs that log to a logging agent or a backend directly, whether they are inside Kubernetes or not doesn't actually matter, because they technically don't send out logs through Kubernetes. In other cases, we'd use the following two patterns for logging.

Collecting logs with a logging agent per node

We know that messages we retrieved via `kubectl logs` are streams redirected from the `stdout`/`stderr` of a container, but it's obviously not a good idea to collect logs with `kubectl logs`. In fact, `kubectl logs` gets logs from kubelet, and kubelet aggregates logs from the container runtime underneath the host path, `/var/log/containers/`. The naming pattern of logs is `{pod_name}_{namespace}_{container_name}_{container_id}.log`.

Therefore, what we need to do to converge the standard streams of running containers is to set up logging agents on every node and configure them to tail and forward log files under the path, as shown in the following diagram:



In practice, we'd also configure the logging agent to tail the logs of the system and the Kubernetes components under `/var/log` on masters and nodes, such as the following:

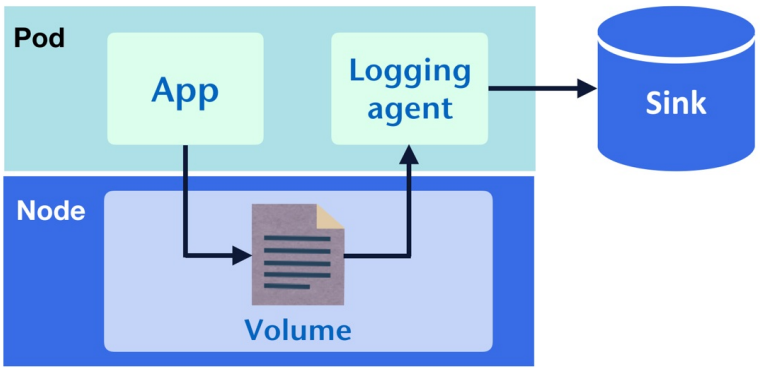
- `kube-proxy.log`
- `kube-apiserver.log`
- `kube-scheduler.log`
- `kube-controller-manager.log`
- `etcd.log`

If the Kubernetes components are managed by `systemd`, the log would be present in `journald`.

Aside from `stdout`/`stderr`, if the logs of an application are stored as files in the container and persisted via the `hostPath` volume, a node logging agent is capable of passing them to a node. However, for each exported log file, we have to customize their corresponding configurations in the logging agent so that they can be dispatched correctly. Moreover, we also need to name log files properly to prevent any collisions and to take care of log rotation manageable, which makes it an unscalable and unmanageable mechanism.

Running a sidecar container to forward written logs

It can be difficult to modify our application to write logs to standard streams rather than log files, and we often want to avoid the troubles brought about by logging to `hostPath` volumes. In this situation, we could run a sidecar container to deal with logging for a pod. In other words, each application pod would have two containers sharing the same `emptyDir` volume, so that the sidecar container can follow logs from the application container and send them outside their pod, as shown in the following diagram:



Although we don't need to worry about managing log files anymore, chores such as configuring logging agents for each pod and attaching metadata from Kubernetes to log entries still takes extra effort. Another option would be to use the sidecar container to output logs to standard streams instead of running a dedicated logging agent, such as the following pod example. In this case, the application container unremittingly writes messages to `/var/log/myapp.log` and the sidecar tails `myapp.log` in the shared volume: **---7-2_logging-sidecar.yml---**
apiVersion: v1
kind: Pod
metadata:
name: myapp
spec:
containers:
- image: busybox
name: application

args:
- /bin/sh
- -c
- >
while true; do
echo "$(date) INFO hello" >> /var/log/myapp.log ;
sleep 1;
done
volumeMounts:
- name: log
mountPath: /var/log
- name: sidecar
image: busybox
args:
- /bin/sh
- -c
- tail -fn+1 /var/log/myapp.log
volumeMounts:
- name: log
mountPath: /var/log
volumes:
- name: log
emptyDir: {}

We can see the written log with `kubectl logs`: **$ kubectl logs -f myapp -c sidecar**
Sun Oct 14 21:26:47 UTC 2018 INFO hello
Sun Oct 14 21:26:48 UTC 2018 INFO hello ...

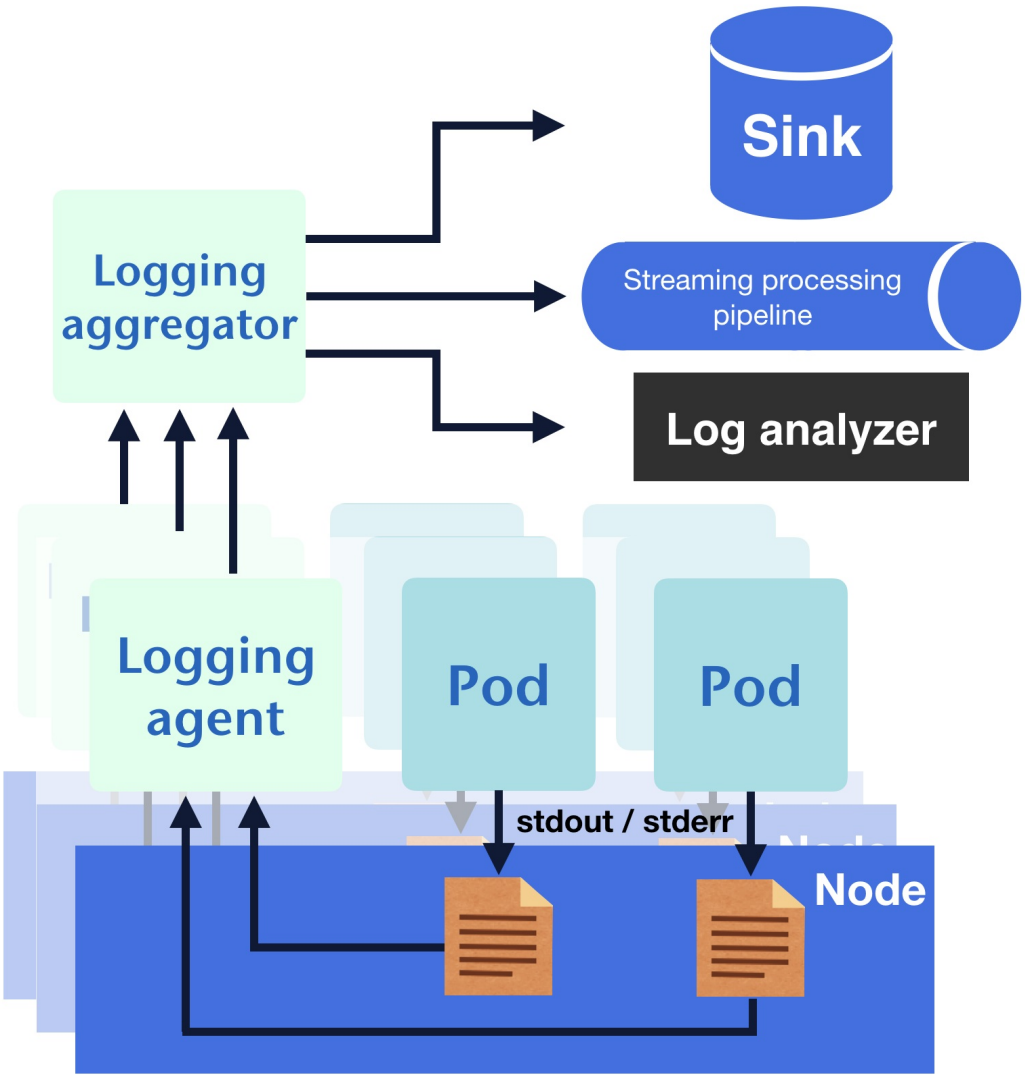
Ingesting Kubernetes state events

The event messages we saw in the output of `kubectl describe` contain valuable information and complement the metrics gathered by `kube-state-metrics`. This allows us to determine exactly what happened to our pods or nodes. Consequently, those event messages should be part of our logging essentials, together with system and application logs. In order to achieve this, we'll need something to watch Kubernetes API servers and aggregate events into a logging sink. The event objects inside Kubernetes are also stored in `etcd`, but tapping into the storage to get those event objects might require a lot of work. Projects such as eventrouter (`https://github.com/heptiolabs/eventrouter`) can help in this scenario. Eventrouter works by translating event objects to structured messages and emitting them to its `stdout`. As a result, our logging system can treat those events as normal logs while keeping the metadata of events.

There are various other alternatives. One is Event Exporter (`https://github.com/GoogleCloudPlatform/k8s-stackdriver/tree/master/event-exporter`), although this only supports StackDriver, a monitoring solution on Google Cloud Platform. Another alternative is the eventer, part of Heapster. This supports Elasticsearch, InfluxDB, Riemann, and Google Cloud Logging as its sink. Eventer can also output to `stdout` directly if the logging system we're using is not supported. However, as Heapster was replaced by the metric server, the development of the eventer was also dropped.

Logging with Fluent Bit and Elasticsearch

So far, we've discussed various logging scenarios that we may encounter in the real world. It's now time to roll up our sleeves and fabricate a logging system. The architectures of logging systems and monitoring systems are pretty much the same in a number of ways: they both have collectors, storage, and consumers such as BI tools or a search engine. The components might vary significantly, depending on the needs. For instance, we might process some logs on the fly to extract real-time information, while we might just archive other logs to durable storage for further use, such as for batch reporting or meeting compliance requirements. All in all, as long as we have a way to ship logs out of our container, we can always integrate other tools into our system. The following diagram depicts some possible use cases:



In this section, we're going to set up the most fundamental logging system. Its components include Fluent Bit, Elasticsearch, and Kibana. The templates for this section can be found under `7-3_efk`, and they are to be deployed to the `logging` namespace.

Elasticsearch is a powerful text search and analysis engine, which makes it an ideal choice for analyzing the logs from everything running in our cluster. The Elasticsearch template for this chapter uses a very simple setup to demonstrate the concept. If you'd like to deploy an Elasticsearch cluster for production use, using the StatefulSet controller to set up a cluster and tuning Elasticsearch with proper configurations, as we discussed in `Chapter 4`, *Managing Stateful Workloads*, is recommended. We can deploy an Elasticsearch instance and a

logging namespace with the following template (https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter7/7-3_efk):

```
$ kubectl apply -f logging-ns.yml
$ kubectl apply -f elasticsearch
```

We know that Elasticsearch is ready if we get a response from `es-logging-svc:9200`.

Elasticsearch is a great document search engine. However, it might not be as good when it comes to persisting a large amount of logs. Fortunately, there are various solutions that allow us to use Elasticsearch to index documents stored in other storage.

The next step is to set up a node logging agent. As we'd run this on every node, we want it to be as light as possible in terms of node resource use; hence why we opted for Fluent Bit (https://fluentbit.io/). Fluent Bit features lower memory footprints, which makes it a competent logging agent for our requirement, which is to ship all the logs out of a node.

As the implementation of Fluent Bit aims to minimize resource usage, it has reduced its functions to a very limited set. If we want to have a greater degree of freedom to combine parsers and filters for different applications in the logging layer, we could use Fluent Bit's sibling project, Fluentd (https://www.fluentd.org/), which is much more extensible and flexible but consumes more resources than Fluent Bit. Since Fluent Bit is able to forward logs to Fluentd, a common method is to use Fluent Bit as the node logging agent and Fluentd as the aggregator, like in the previous figure.

In our example, Fluent Bit is configured as the first logging pattern. This means that it collects logs with a logging agent per node and sends them to Elasticsearch directly:

```
$ kubectl apply -f logging-agent/fluentbit/
```

The ConfigMap for Fluent Bit is already configured to tail container logs under `/var/log/containers` and the logs of certain system components under `/var/log`. Fluent Bit can also expose its stats metrics in Prometheus format on port `2020`, which is configured in the DaemonSet template.

Due to stability issues and the need for flexibility, it is still common to use Fluentd as a logging agent. The templates can be found under `logging-agent/fluentd` in our example, or at the official repository here: https://github.com/fluent/fluentd-kubernetes-daemonset.

To use Kubernetes events, we can use the `eventrouter`:

```
$ kubectl apply -f eventrouter.yml
```

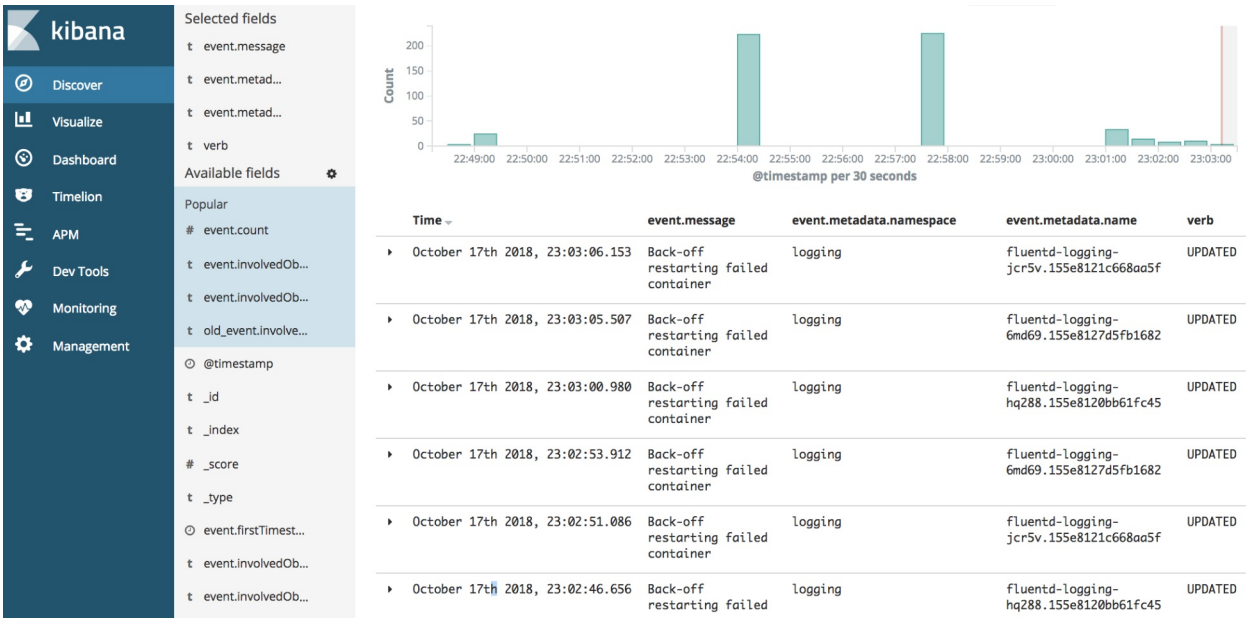
This will start to print events in JSON format at the `stdout` stream, so that we can index them in Elasticsearch.

To see logs emitted to Elasticsearch, we can invoke the search API of Elasticsearch, but there's a better option: Kibana, a web interface that allows us to play with Elasticsearch. Deploy everything under `kibana` in the examples for this section with the following command:

```
$ kubectl apply -f kibana
```

Grafana also supports reading data from Elasticsearch: http://docs.grafana.org/features/datasources/elasticsearch/.

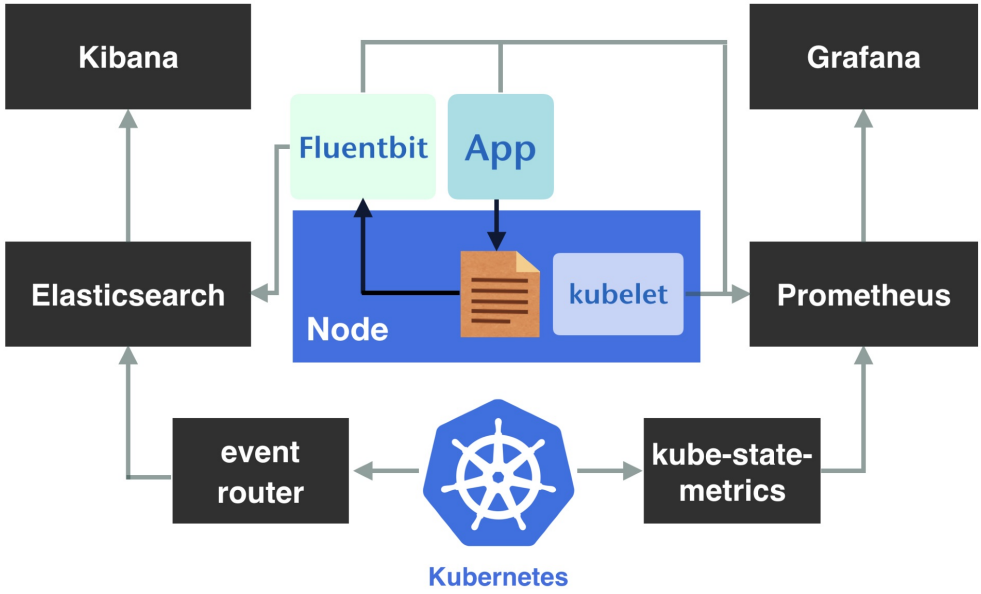
Kibana, in our example, is listening to port `5601`. After exposing the service from your cluster and connecting to it with any browser, you can start to search logs from Kubernetes. In our example Fluent Bit configuration, the logs routed by eventrouter would be under the index named `kube-event-*`, while logs from other containers could be found at the index named `kube-container-*`. The following screenshot shows what a event message looks like in Kibana:



Time	event.message	event.metadata.namespace	event.metadata.name	verb
October 17th 2018, 23:03:06.153	Back-off restarting failed container	logging	fluentd-logging-jcr5v.155e8121c668aa5f	UPDATED
October 17th 2018, 23:03:05.507	Back-off restarting failed container	logging	fluentd-logging-6md69.155e8127d5fb1682	UPDATED
October 17th 2018, 23:03:00.980	Back-off restarting failed container	logging	fluentd-logging-hq288.155e8120bb61fc45	UPDATED
October 17th 2018, 23:02:53.912	Back-off restarting failed container	logging	fluentd-logging-6md69.155e8127d5fb1682	UPDATED
October 17th 2018, 23:02:51.086	Back-off restarting failed container	logging	fluentd-logging-jcr5v.155e8121c668aa5f	UPDATED
October 17th 2018, 23:02:46.656	Back-off restarting failed	logging	fluentd-logging-hq288.155e8120bb61fc45	UPDATED

Extracting metrics from logs

The monitoring and logging system we built around our application on top of Kubernetes is shown in the following diagram:



The logging part and the monitoring part look like two independent tracks, but the value of the logs is much more than a collection of short texts. This is structured data and usually emitted with timestamps; because of this, if we can parse information from logs and project the extracted vector into the time dimension according to the timestamps, it will become a time series metric and be available in Prometheus.

For example, an access log entry from any of the web servers may look as follows: 10.1.5.7 - - [28/Oct/2018:00:00:01 +0200] "GET /ping HTTP/1.1" 200 68 0.002

This consists of data such as the request IP address, the time, the method, the handler, and so on. If we demarcate log segments by their meanings, the counted sections can then be regarded as a metric sample, as follows: {ip:"10.1.5.7",handler:"/ping",method:"GET",status:200,body_size:68,duration:0

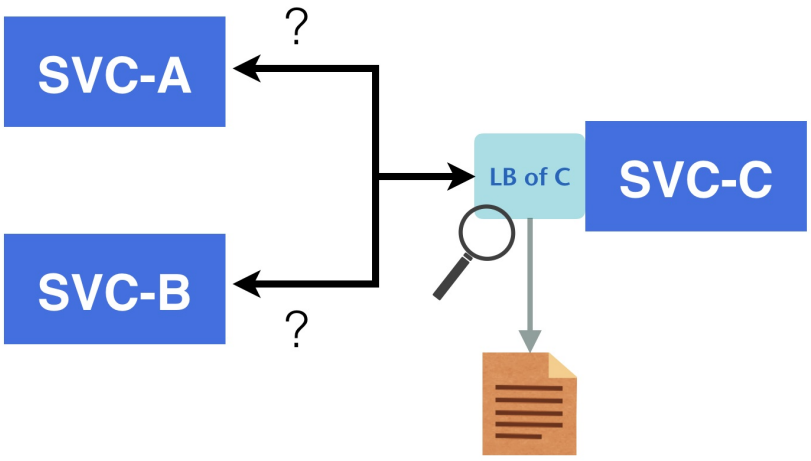
After the transformation, tracing the log over time will be more intuitive.

To organize logs into the Prometheus format, tools such as mtail (https://github.com/google/mtail), Grok Exporter (https://github.com/fstab/grok_exporter), or Fluentd (https://github.com/fluent/fluent-plugin-prometheus) are all widely used to extract log entries into metrics.

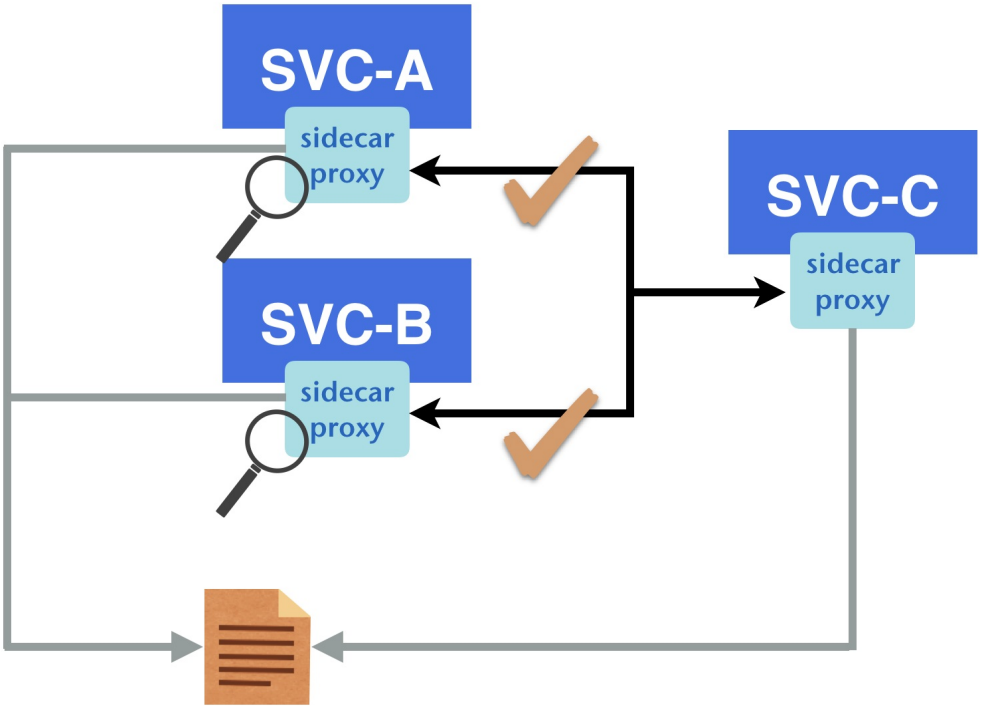
Arguably, lots of applications nowadays support outputting structured metrics directly, and we can always instrument our own application for this type of information. However, not everything in our tech stack provides us with a convenient way to get their internal states, especially operating system utilities, such as `ntpd`. It's still worth having this kind of tool in our monitoring stack to help us improve the observability of our infrastructure.

Incorporating data from Istio

In a service mesh, the gateway between every service is the front proxy. For this reason, the front proxy is, unsurprisingly, a rich information source for things running inside the mesh. However, if our tech stack already has similar components, such as load balancers or reverse proxies for internal services, then what's the difference between collecting traffic data from them and the service mesh proxy? Let's consider the classical setup:



SVC-A and **SVC-B** make requests to **SVC-C**. The data gathered from the load balancer for **SVC-C** represents the quality of **SVC-C**. However, as we don't have any visibility over the path from the clients to **SVC-C**, the only way to measure the quality between **SVC-A** or **SVC-B** and **SVC-C** is either by relying on a mechanism built on the client side, or by putting probes in the network that the clients are in. For a service mesh, take a look at the following diagram:

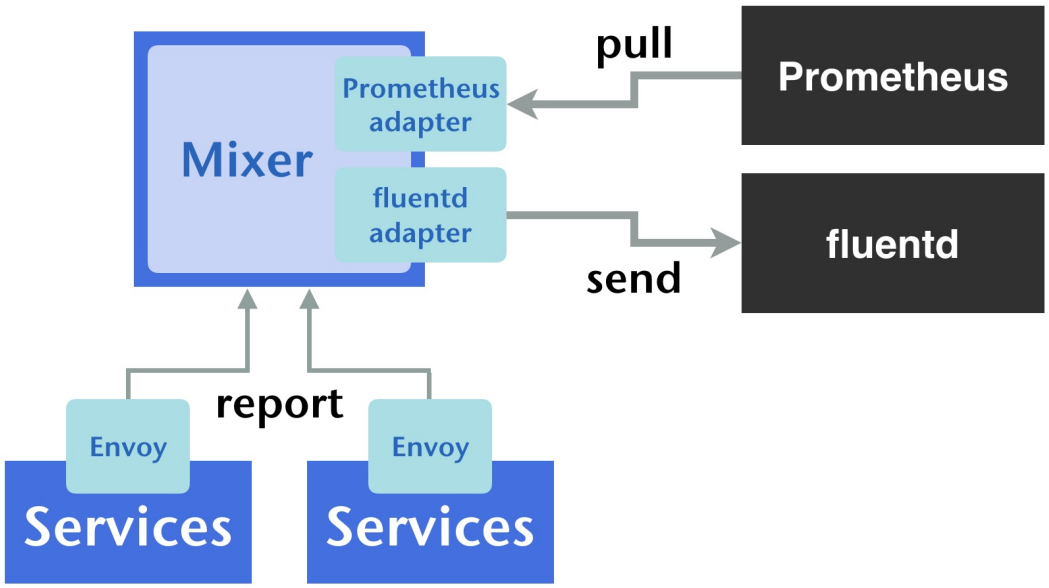


Here, we want to know the quality of **SVC-C**. In this setup, **SVC-A** and **SVC-B** communicate with **SVC-C** via their sidecar proxies, so if we collect metrics about requests that go to **SVC-C** from all client-side proxies, we can also get the same data from the server-side load balancer, plus the missing measurement between **SVC-C** and its clients. In other words, we can have a consolidated way to measure not only how **SVC-C** performs, but also the quality between **SVC-C** and its clients. This augmented information also helps us to locate failures when triaging a problem.

The Istio adapter model

Mixer is the component that manages telemetry in Istio's architecture. It takes the statistics from the side proxy, deployed along with the application container, and interacts with other backend components through its adapters. For instance, our monitoring backend is Prometheus, so we can utilize the Prometheus adapter of mixer to transform the metrics we get from envoy proxies into a Prometheus metrics path.

The way in which access logs go through the pipeline to our Fluentd/Fluent Bit logging backend is the same as in the one we built previously, the one that ships logs into Elasticsearch. The interactions between Istio components and the monitoring backends are illustrated in the following diagram:



Configuring Istio for existing infrastructure

The adapter model allows us to fetch the monitoring data from Mixer components easily. It requires the configurations that we will explore in the following sections.

apiVersion: config.istio.io/v1alpha2
kind: logentry
metadata:
name: accesslog
namespace: istio-system
spec:
severity: '"info"'
timestamp: request.time
variables:
source: source.workload.name | "unknown"
 destination: destination.workload.name | "unknown"

 method: request.method | ""

 url: request.path | ""
 protocol: request.scheme | ""
 responseCode: response.code | 0
 responseSize: response.size | 0
 requestSize: request.size | 0
 latency: response.duration | "0ms"
 monitored_resource_type: '"UNSPECIFIED"'

The complete reference for each kind of template can be found here: https://istio.io/docs/reference/config/policy-and-telemetry/templates/.

apiVersion: config.istio.io/v1alpha2
kind: handler
metadata:
name: fluentd
namespace: istio-system
spec:
compiledAdapter: fluentd
params:
address: fluentd-aggegater-svc.logging:24224

From this code snippet, Mixer knows a destination that can receive the `logentry`. The capabilities of every type of adapter differ significantly. For example, the `fluentd` adapter can only accept the `logentry` template, and `Prometheus` is only able to deal with the `metric` template, while the Stackdriver can take `metric`, `logentry`, and `tracespan` templates. All supported adapters are listed here: https://istio.io/docs/reference/config/policy-and-telemetry/adapters/.

Rules

Rules are the binding between a template and a handler. If we already have an `accesslog`, `logentry` and a `fluentd` handler in the previous examples, then a rule such as this one associates the two entities: **apiVersion: config.istio.io/v1alpha2**
kind: rule
metadata:
name: accesslogtofluentd
namespace: istio-system
spec:
match: "true"
actions:
- handler: fluentd
instances:
- accesslog.logentry

Once the rule is applied, the mixer knows it should send the access logs in the format defined previously to the `fluentd` at `fluentd-aggegater-svc.logging:24224`.

The example of deploying a `fluentd` instance that takes inputs from the TCP socket can be found under `7_3efk/logging-agent/fluentd-aggregator` (https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter7/7-3_efk/logging-agent/fluentd-aggregator), and is configured to forward logs to the Elasticsearch instance we deployed previously. The three Istio templates for access logs can be found under `7-4_istio_fluentd_accesslog.yml` (https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/blob/master/chapter7/7-4_istio_fluentd_accesslog.yml).

Let's now think about metrics. If Istio is deployed by the official chart with Prometheus enabled (it is enabled by default), then there will be a Prometheus instance in your cluster under the `istio-system` namespace. Additionally, Prometheus would be preconfigured to gather metrics from the Istio components. However, for various reasons, we may want to use our own Prometheus deployment, or make the one that comes with Istio dedicated to metrics from Istio components only. On the other hand, we know that the Prometheus architecture is flexible, and as long as the target components expose their metrics endpoint, we can configure our own Prometheus instance to scrape

those endpoints.

Some useful endpoints from Istio components are listed here:

- `<all-components>:9093/metrics`: Every Istio component exposes their internal states on port `9093`.
- `<envoy-sidecar>:15090/stats/prometheus`: Every envoy proxy prints the raw stats here. If we want to monitor our application, it is advisable to use the mixer template to sort out the metrics first.
- `<istio-telemetry-pods>:42422/metrics`: The metrics configured by the Prometheus adapter and processed by mixer will be available here. Note that the metrics from an envoy sidecar are only available in the telemetry pod that the envoy reports to. In other words, we should use the endpoint discovery mode of Prometheus to collect metrics from all telemetry pods instead of scraping data from the telemetry service.

By default, the following metrics will be configured and available in the Prometheus path:

- `requests_total`
- `request_duration_seconds`
- `request_bytes`
- `response_bytes`
- `tcp_sent_bytes_total`
- `tcp_received_bytes_total`

Another way to make the metrics collected by the Prometheus instance, deployed along with official Istio releases, available to our Prometheus is by using the federation setup. This involves setting up one Prometheus instance to scrape metrics stored inside another Prometheus instance. This way, we can regard the Prometheus for Istio as the collector for all Istio-related metrics. The path for the federation feature is at `/federate`. Say we want to get all the metrics with the label `{job="istio-mesh"}`, the query parameter would be as follows:

```
http://<prometheus-for-istio>/federate?match[]={job="istio-mesh"}
```

As a result, by adding a few configuration lines, we can easily integrate Istio metrics into the existing monitoring pipeline. For a full reference on federation, take a look at the official documentation: https://prometheus.io/docs/prometheus/latest

/federation/.

Summary

At the start of this chapter, we described how to get the status of running containers quickly by means of built-in functions such as `kubectl`. Then, we expanded the discussion to look at the concepts and principles of monitoring, including why, what, and how to monitor our application on Kubernetes. Afterward, we built a monitoring system with Prometheus as the core, and set up exporters to collect metrics from our application, system components, and Kubernetes units. The fundamentals of Prometheus, such as its architecture and query domain-specific language were also introduced, so we can now use metrics to gain insights into our cluster, as well as the applications running inside, to not only retrospectively troubleshoot, but also detect potential failures. After that, we described common logging patterns and how to deal with them in Kubernetes, and deployed an EFK stack to converge logs. Finally, we turned to another important piece of infrastructure between Kubernetes and our applications, the service mesh, to get finer precision when monitoring telemetry. The system we built in this chapter enhances the reliability of our service.

In `Chapter 8`, *Resource Management and Scaling*, we'll leverage those metrics to optimize the resources used by our services.

Resource Management and Scaling

Despite the fact that we now have a comprehensive view about everything to do with applications and the cluster thanks to our monitoring system, we are still lacking the ability to handle capacity in terms of computational resources and the cluster. In this chapter, we'll discuss resources, which will include the following topics:

- Kubernetes scheduling mechanisms
- Affinities between resources and workloads
- Scaling smoothly with Kubernetes
- Arranging cluster resources
- Node administration

Scheduling workloads

The term scheduling refers to assigning resources to a task that needs to be carried out. Kubernetes does way more than keeping our containers running; it proactively watches resource usage of a cluster and carefully schedules pods to the available resources. This type of scheduler-based infrastructure is the key that enables us to run workloads more efficiently than a classical infrastructure.

Optimizing resource utilization

Unsurprisingly, the way in which Kubernetes allocates pods to nodes is based on the supply and demand of resources. If a node can provide a sufficient quantity of resources, the node is eligible to run the pod. Hence, the smaller the difference between the cluster capacity and the actual usage, the higher resource utilization we can obtain.

Resource types and allocations

There are two core resource types that participate in the scheduling process, namely CPU and memory. To see the capability of a node, we can check its `.status.allocatable` path:

```
# Here we only show the path with a go template.
$ kubectl get node <node_name> -o go-template --template=\
'{{range $k,$v:=.status.allocatable}}{{printf "%s: %s\n" $k $v}}{{end}}'
cpu: 2
ephemeral-storage: 14796951528
hugepages-2Mi: 0
memory: 1936300Ki
pods: 110
```

As we can see, these resources will be allocated to any pod in need by the scheduler. But how does the scheduler know how many resources a pod will consume? We actually have to instruct Kubernetes about the request and the limit for each pod. The syntax is `spec.containers[].resources.{limits,requests}.{resource_name}` in the pod's manifest:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    resources:
      requests:
        cpu: 100m
        memory: 10Mi
      limits:
        cpu: 0.1
        memory: 100Mi
```

The unit of CPU resources can be either a fractional number or a millicpu expression. A CPU core (or a Hyperthread) is equal to 1,000 millicores, or a simple 1.0 in fractional number notation. Note that the fractional number notation is an absolute quantity. For instance, if we have eight cores on a node, the expression 0.5 means that we are referring to 0.5 cores, rather than four cores. In this sense, in the previous example, the amount of requested CPU, `100m`, and the CPU limit, `0.1`, are equivalent.

Memory is represented in bytes, and Kubernetes accepts the following suffixes and notations:

- **Base 10**: E, P, T, G, M, K
- **Base 2**: Ei, Pi, Ti, Gi, Mi, Ki
- **Scientific notation**: e

Hence, the following forms are roughly the same: `67108864`, `67M`, `64Mi`, and `67e6`.

*Other than CPU and memory, there are many more resource types that are added to Kubernetes, such as **ephemeral storage** and **huge pages**. Vendor-specific resources such as GPU, FPGA, and NICs can be used by the Kubernetes scheduler with device plugins. You can also bind custom resource types, such as licences, to nodes or the cluster and configure pods to consume them. Please refer to the following related references:*
***Ephemeral storage**:*
https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#local-ephemeral-storage
***Huge pages**:*
https://kubernetes.io/docs/tasks/manage-hugepages/scheduling-hugepages/
***Device resources**:*
https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/device-plugins/
***Extended resources**:*
https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#extended-resources

As the name suggests, a request refers to the quantity of resources a pod might take, and Kubernetes uses it to pick a node to schedule the pod. For each type of resource, the sum of requests from all containers on a node will never exceed the allocatable resource of that node. In other words, every container that is successfully scheduled is guaranteed to get the amount of resources it requested.

To maximize the overall resource utilization, as long as the node that a pod is on has spare resources, that pod is allowed to exceed the amount of resources that it requested. However, if every pod on a node uses more resources than they should, the resource that node provides might eventually be exhausted, and this might result in an unstable node. The concept of limits addresses this situation:

- If a pod uses more than a certain percentage of CPU, it will be throttled (not killed)
- If a pod reaches the memory limit, it will be killed and restarted

These limits are hard constraints, so it's always larger or equal to a request of the same resource type.

The requests and limits are configured per container. If a pod has more than one container, Kubernetes will schedule the pod base on the sum of all the

containers' requests. One thing to note is that if the total requests of a pod exceed the capacity of the largest node in a cluster, the pod will never be scheduled. For example, suppose that the largest node in our cluster can provide 4,000 m (four cores) of CPU resources, then neither a single container pod that wants 4,500 m of CPU, or a pod with two containers that request 2,000 m and 2,500 m can be assigned, since no node can fulfil their requests.

Since Kubernetes schedules pods based on requests, what if all pods come without any requests or limits? In this case, as the sum of requests is `0`, which would always be less than the capacity of a node, Kubernetes would keep placing pods onto the node until it exceeds the node's real capability. By default, the only limitation on a node is the number of allocatable pods. It's configured with a kubelet flag, `--max-pods`. In the previous example, this was 110. Another tool to set the default constraint on resources is `LimitRange`, which we'll talk about later on in this chapter.

Other than kubelet's `--max-pods` flag, we can also use the similar flag, `--pods-per-core`, which enforces the maximum pods a core can run.

$ kubectl apply -f chapter8/8-1_qos/qos-pods.yml
pod/besteffort-nothing-specified created
...
$ kubectl get pod -o go-template --template=\
'{{range .items}}{{printf "pod/%s: %s\n" .metadata.name .status.qosClass}}{{end}}'

pod/besteffort-explicit-0-req: BestEffort
pod/besteffort-nothing-specified: BestEffort
pod/burstable-lim-lt-req: Burstable
pod/burstable-partial-req-multi-containers: Burstable
pod/guranteed: Guaranteed
pod/guranteed-lim-only: Guaranteed

Each class has their advantages and disadvantages:

- `BestEffort`: Pods in this class can use all resources on the node if they are available.
- `Burstable`: Pods in this class are guaranteed to get the resource they requested and can still use extra resources on the node if available. Besides, if there are multiple `Burstable` pods that need additional CPU percentages than they originally requested, the remaining CPU resources on the node are distributed by the ratio of requests from all pods. For example, say pod A wants 200 m and pod B wants 300 m and we have 1,000 m on the node. In this case, A can use *200 m + 0.4 * 500 m = 400 m* at most, and B will get *300 m + 300 m = 600 m*.
- `Guaranteed`: Pods are assured to get their requested resources, but they cannot consume resources beyond the set limits.

The priority of QoS classes from highest to lowest is `Guaranteed` > `Burstable` > `BestEffort`. If a node experiences resource pressure, which requires immediate action to reclaim the scarce resource, then pods will be killed or throttled according to their priority. For example, the implementation of memory assurance is done by an **Out-Of-Memory** (**OOM**) killer at the operating system level. Therefore, by adjusting OOM scores of pods according to their QoS class, the node OOM killer will know which pod can be scavenged first when the node is under memory pressure. As such, even though guaranteed pods seem to be the most restricted class, they are also the safest pods in the cluster as their requirements will be fulfilled as far as possible.

Placing pods with constraints

Most of the time, we don't really care about which node our pods are running on as we just want Kubernetes to arrange adequate computing resources to our pods automatically. Nevertheless, Kubernetes isn't aware of factors such as the geographical location of a node, availability zones, or machine types when scheduling a pod. This lack of awareness about the environment makes it hard to deal with situations in which pods need to be bound to nodes under certain conditions, such as deploying testing builds in an isolated instance group, putting I/O intensive tasks on nodes with SSD disks, or arranging pods to be as close as possible. As such, to complete the scheduling, Kubernetes provides different levels of affinities that allow us to actively assign pods to certain nodes based on labels and selectors.

When we type `kubectl describe node`, we can see the labels attached to nodes: **$ kubectl describe node**
Name: gke-mycluster-default-pool-25761d35-p9ds
Roles: <none>
Labels: beta.kubernetes.io/arch=amd64
beta.kubernetes.io/fluentd-ds-ready=true
beta.kubernetes.io/instance-type=f1-micro
beta.kubernetes.io/kube-proxy-ds-ready=true
beta.kubernetes.io/os=linux
cloud.google.com/gke-nodepool=default-pool
cloud.google.com/gke-os-distribution=cos
failure-domain.beta.kubernetes.io/region=europe-west1
failure-domain.beta.kubernetes.io/zone=europe-west1-b
kubernetes.io/hostname=gke-mycluster-default-pool-25761d35-p9ds
...

`kubectl get nodes --show-labels` allows us to get just the label information of nodes instead of everything.

These labels reveal some basic information about a node, as well as its environment. For convenience, there are also well-known labels provided on most Kubernetes platforms:

- `kubernetes.io/hostname`
- `failure-domain.beta.kubernetes.io/zone`
- `failure-domain.beta.kubernetes.io/region`
- `beta.kubernetes.io/instance-type`
- `beta.kubernetes.io/os`
- `beta.kubernetes.io/arch`

The value of these labels might differ from provider to provider. For instance, `failure-domain.beta.kubernetes.io/zone` will be the availability zone name in AWS, such as `eu-west-1b`, or the zone name in GCP, such as `europe-west1-b`. Also, some specialized platforms, such as `minikube`, don't have all of these labels:

```
$ kubectl get node minikube -o go-template --template=\
'{{range $k,$v:=.metadata.labels}}{{printf "%s: %s\n" $k $v}}{{end}}'
beta.kubernetes.io/arch: amd64
beta.kubernetes.io/os: linux
kubernetes.io/hostname: minikube
node-role.kubernetes.io/master:
```

Additionally, if you're working with a self-hosted cluster, you can use the `--node-labels` flag of kubelet to attach labels on a node when joining a cluster. As for other managed Kubernetes clusters, there are usually ways to customize labels, such as the label field in `NodeConfig` on GKE.

Aside from these pre-attached labels from kubelet, we can tag our node manually by either updating the manifest of the node or using the shortcut command, `kubectl label`. The following example tags two labels, `purpose=sandbox` and `owner=alpha`, to one of our nodes: ## **display only labels on the node:**
$ kubectl get node gke-mycluster-default-pool-25761d35-p9ds -o go-template --template='{{range $k,$v:=.metadata.labels}}{{printf "%s: %s\n" $k $v}}{{end}}'
beta.kubernetes.io/arch: amd64
beta.kubernetes.io/fluentd-ds-ready: true
beta.kubernetes.io/instance-type: f1-micro
beta.kubernetes.io/kube-proxy-ds-ready: true
beta.kubernetes.io/os: linux
cloud.google.com/gke-nodepool: default-pool
cloud.google.com/gke-os-distribution: cos
failure-domain.beta.kubernetes.io/region: europe-west1
failure-domain.beta.kubernetes.io/zone: europe-west1-b

kubernetes.io/hostname: gke-mycluster-default-pool-25761d35-p9ds

attach label
**$ kubectl label node gke-mycluster-default-pool-25761d35-p9ds **
purpose=sandbox owner=alpha
node/gke-mycluster-default-pool-25761d35-p9ds labeled

check labels again
$ kubectl get node gke-mycluster-default-pool-25761d35-p9ds -o go-template --template='{{range $k,$v:=.metadata.labels}}{{printf "%s: %s\n" $k $v}}{{end}}'
...
kubernetes.io/hostname: gke-mycluster-default-pool-25761d35-p9ds
owner: alpha
purpose: sandbox

With these node labels, we're capable of describing various notions. For example, we can specify that a certain group of pods should only be put on nodes that are in the same availability zone. This is indicated by the `failure-domain.beta.kubernetes.io/zone: az1` label. Currently, there are two expressions that we can use to configure the condition from pods: `nodeSelector` and pod/node affinity.

Node selector

The node selector of a pod is the most intuitive way to place pods manually. It's similar to the pod selectors of the service object but instead for choosing nodes —that is, a pod would only be put on nodes with matching labels. The corresponding label key-value map is set at the `.spec.nodeSelector` of a pod's manifest in the following form: **...**
spec:
nodeSelector:
<node_label_key>: <label_value>
...

It's possible to assign multiple key-value pairs in a selector, and Kubernetes will find eligible nodes for the pod with the intersection of those key-value pairs. For instance, the following snippet of a `spec` pod tells Kubernetes we want the pod to be on nodes with the `purpose=sandbox` and `owner=alpha` labels:

```
...
spec:
  containers:
  - name: main
    image: my-app
  nodeSelector:
    purpose: sandbox
    owner: alpha
...
```

If Kubernetes can't find a node with such label pairs, the pod won't be scheduled and will be marked as in the `Pending` state. Moreover, since `nodeSelector` is a map, we can't assign two identical keys in a selector, otherwise the value of the keys that appeared previously will be overwritten by later ones.

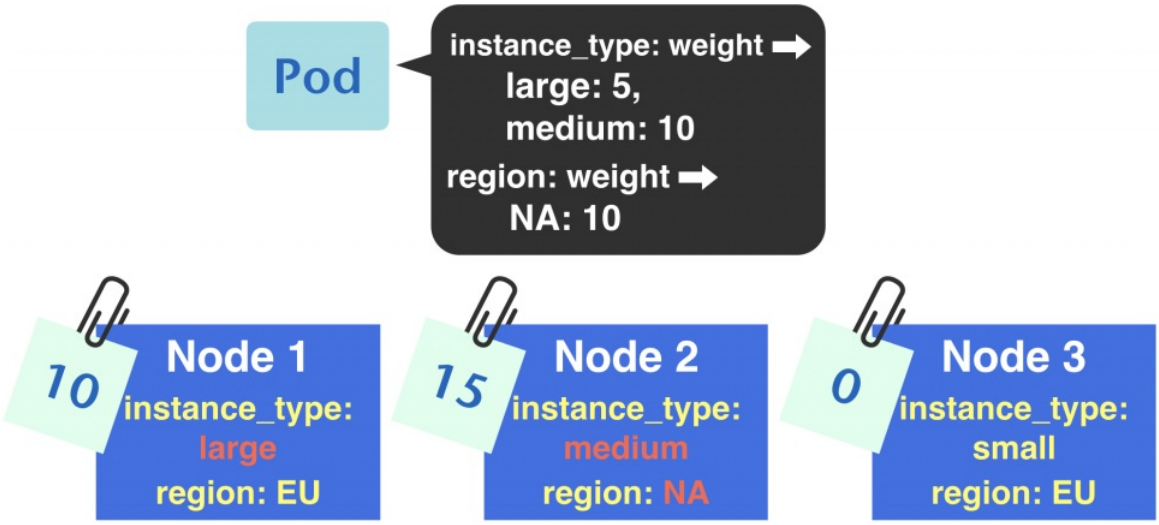
Affinity and anti-affinity

Even though `nodeSelector` is simple and flexible, it's still inept at expressing the complicated needs of real-world applications. For example, we usually don't want pods of a `StatefulSet` be put in the same availability zone to satisfy cross-zone redundancy. It can be difficult to configure such requirements with only node selectors. For this reason, the concept of scheduling under constraints with labels has been extended to include affinity and anti-affinity.

Affinity comes into play in two different scenarios: pods-to-nodes and pods-to-pods. It's configured under the `.spec.affinity` path of a pod. The first option, `nodeAffinity`, is pretty much the same as `nodeSelector`, but formulates the relation between pods and nodes in a more expressive manner. The second option represents inter-pod enforcement in two forms: `podAffinity` and `podAntiAffinity`. For both nodes and inter-pod affinity, there are two different degrees of requirements:

- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`

As can be seen from their names, both requirements take effect during scheduling, not execution—that is, if a pod has already been scheduled on a node, it remains in execution even if the condition of that node becomes ineligible for scheduling the pod. As for `required` and `preferred`, these represent the notion of hard and soft constraints, respectively. For a pod with the required criteria, Kubernetes will find a node that satisfies all requirements to run it; while in the case of the preferred criteria, Kubernetes will try to find a node that has the highest preference to run the pod. If there's no node that matches the preference, then the pod won't be scheduled. The calculation of preference is based on a configurable `weight` associated with all terms of the requirement. For nodes that already satisfy all other required conditions, Kubernetes will iterate through all preferred terms to sum the weight of each matched term as the preference score of a node. Take a look at the following example:



The pod has three weighted preferences on two keys: `instance_type` and `region`. When scheduling the pod, the scheduler will start matching the preferences with labels on nodes. In this example, since **Node 2** has the `instance_type=medium` and `region=NA` labels, it gets a score of 15, which is the highest score out of all nodes. For this reason, the scheduler will place the pod on **Node 2**.

There are differences between the configuration for node affinity and inter-pod affinity. Let's discuss these separately.

Node affinity

The description of a required statement is called `nodeSelectorTerms`, and is composed of one or more `matchExpressions`. `matchExpressions`, which is similar to the `matchExpressions` that is used by other Kubernetes controllers such as `Deployment` and `StatefulSets`, but in this case, the `matchExpressions` node supports the following operators: `In`, `NotIn`, `Exists`, `DoesNotExist`, `Gt`, and `Lt`.

A node affinity requirement looks as follows: **...**
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: <key_1>
operator: <In, NotIn, Exists, DoesNotExist. Gt, or Lt>
values:
- <value_1>
- <value_2>
- ...
- key: <key_2>
...

```
  - matchExpressions:
  ...
...
```

For conditions that have multiple `nodeSelectorTerms` defined (each term is a `matchExpression` object), the required statement will be evaluated as `true` if any `nodeSelectorTerm` is met. But for multiple expressions in a `matchExpression` object, the term will be evaluated as `true` if all `matchExpressions` are satisfied, for instance, if we have the following configuration and their results:

```
nodeSelectorTerms:
- matchExpressions:       <- <nodeSelectorTerm_A>
  - <matchExpression_A1>  : true
  - <matchExpression_A2>  : true
- matchExpressions:       <- <nodeSelectorTerm_B>
  - <matchExpression_B1>  : false
  - <matchExpression_B2>  : true
  - <matchExpression_B3>  : false
```

The evaluation result of the `nodeSelectorTerms` would be `true` after applying the previous `AND`/`OR` rules: **Term_A = matchExpression_A1 && matchExpression_A2**
Term_B = matchExpression_B1 && matchExpression_B2 && matchExpression_B3
nodeSelectorTerms = Term_A || Term_B
>> **(true && true) || (false && true && false)**
>> **true**

The `In` and `NotIn` operators can match multiple values, while `Exists` and `DoesNotExist` don't take any value (`values: []`); `Gt` and `Lt` only take a single integer value in the string type (`values: ["123"]`).

The `require` statement can be a replacement for `nodeSelector`. For instance, the `affinity` section and the following `nodeSelector` section are equivalent: **...**
affinity:
nodeAffinity:
requiredDuringSchedulingIgnoredDuringExecution:
nodeSelectorTerms:
- matchExpressions:
- key: purpose
operator: In
values: ["sandbox"]
- key: owner
operator: In
values: ["alpha"]
nodeSelector:
purpose: sandbox
owner: alpha
...

As well as `matchExpressions`, there's another term, `matchFields`, for selecting values outside labels. As of Kubernetes 1.13, the only supported field is `metadata.name`, which is used to pick a node whose name isn't equal to the value of the `kubernetes.io/hostname` label. Its syntax is basically the same as `matchExpression`: `{"matchFields":[{"key": "metadata.name", "operator": "In", "values": ["target-name"]}]}`.

The configuration of preferences is akin to the required statement as they share `matchExpressions` to express relations. One difference is that a preference has a

`weight` field to denote its importance, and the range of a `weight` field is `1-100`: **...**
preferredDuringSchedulingIgnoredDuringExecution:
- weight: <1-100>
preference:
- matchExpressions:
- key: <key_1>
operator: <In, NotIn, Exists, DoesNotExist. Gt, or Lt>
values:
- <value_1>
- <value_2>
- ...
- key: <key_2>
...
- matchExpressions:
...
...

If we write down the condition specified in the diagram that we used in the previous section in the preference configuration, it would look as follows: **...**
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 5
preference:
matchExpressions:
- key: instance_type
operator: In
values:
- medium
- weight: 10
preference:
matchExpressions:
- key: instance_type
operator: In
values:
- large
- weight: 10
preference:
matchExpressions:

- key: region
operator: In
values:
- NA
...

Inter-pod affinity

Even though the extended functionality of node affinity makes scheduling more flexible, there are still some circumstances that aren't covered. Say we have a simple request, such as dividing the pods of a deployment between different machines—how can we achieve that? This is a common requirement but it's not as trivial as it seems to be. Inter-pod affinity brings us additional flexibility to reduce the effort required to deal with this kind of problem. Inter-pod affinity takes effect on labels of certain running pods in a defined group of nodes. To put it another way, it's capable of translating our needs to Kubernetes. We can specify, for example, that a pod shouldn't be placed along with another pod with a certain label. The following is a definition of an inter-pod affinity requirement:

```
...
affinity:
  podAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
    - labelSelector:
        matchExpressions:
        - key: <key_1>
          operator: <In, NotIn, Exists, or DoesNotExist>
          values:
          - <value_1>
          - <value_2>
          ...
        - key: <key_2>
        ...
      topologyKey: <a key of a node label>
      namespaces:
      - <ns_1>
      - <ns_2>
      ...
...
```

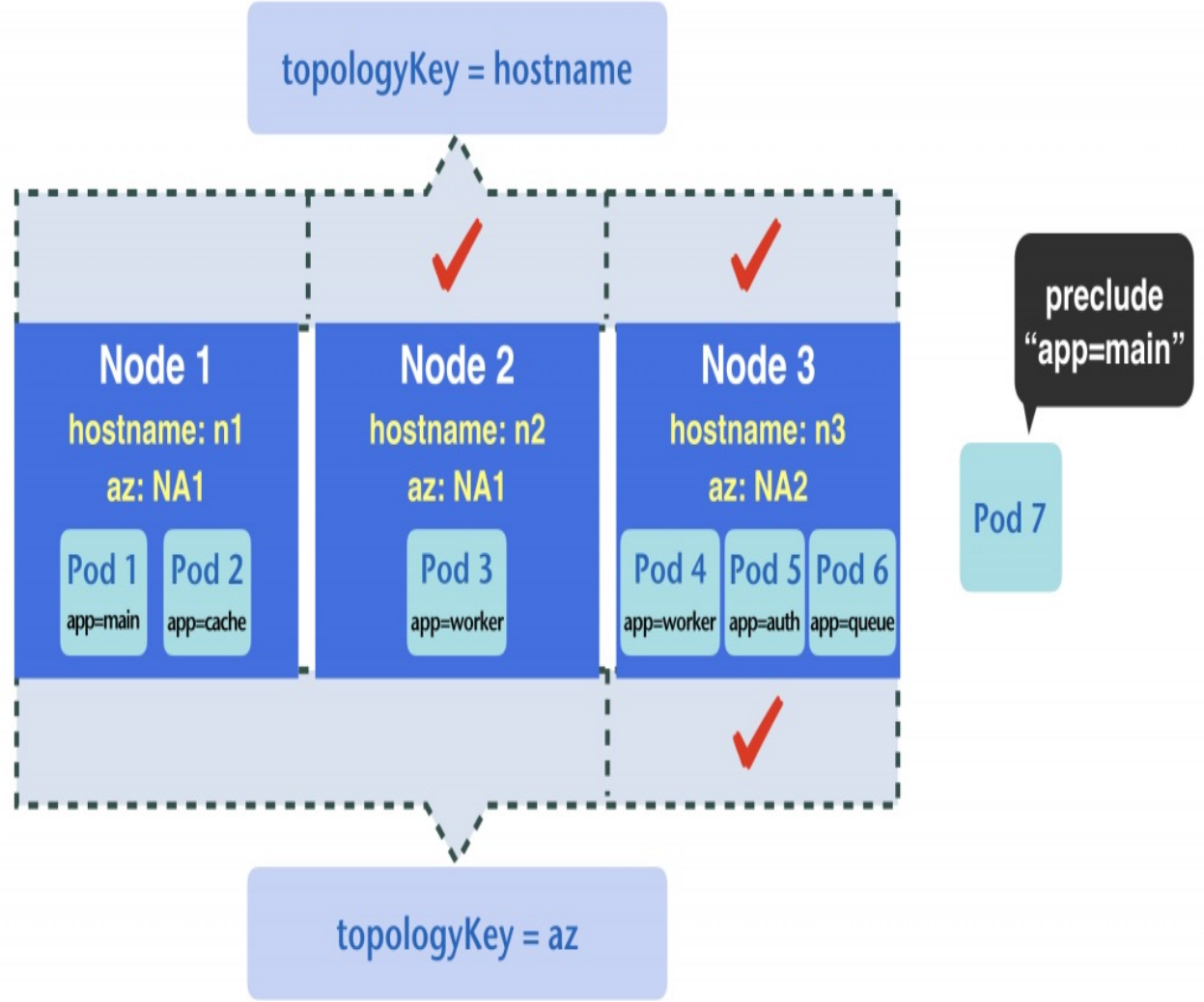
Its structure is almost identical to node affinity. The differences are as follows:

- Inter-pod affinity requires the use of effective namespaces. Unlike a node, a pod is a namespaced object, so we have to tell Kubernetes which namespaces we're referring to. If the namespace field is blank, Kubernetes will assume that the target pod is in the same namespace as the pod that specified the affinity.
- The term to describe a requirement, `labelSelector`, is the same as the one used in controllers such as `Deployment`. The supported operators, therefore, are `In`,

`NotIn`, `Exists`, and `DoesNotExist`.

- `topologyKey`, which is used to define the searching scope of nodes, is a required field. Note that `topologyKey` should be a key of a node label, not the key on a pod label.

To make the idea of `topologyKey` clearer, consider the following diagram:



We want Kubernetes to find a slot for our new pod (**Pod 7**) with the affinity that it can't be placed with other pods that have certain label key-value pairs, say, `app=main`. If the `topologyKey` of the affinity is `hostname`, then the scheduler would evaluate the terms in the labels of **Pod 1** and **Pod 2**, the labels of **Pod 3**, and the labels of **Pod 4**, **Pod 5**, and **Pod 6**. Our new pod would be assigned to either Node 2 or Node 3, which corresponds to the red checks on the upper part of the previous diagram. If the `topologyKey` is `az`, then the searching range would become

the labels of **Pod 1**, **Pod 2**, and **Pod 3** and the labels of **Pod 4**, **Pod 5**, and **Pod 6**. As a consequence, the only possible node is **Node 3**.

The preference and its `weight` parameter for inter-pod affinity and node affinity are the same. The following is an example using a preference to place the pods of a `Deployment` as close to each other as possible:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: colocate
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - image: busybox
        name: myapp
        command: ["sleep", "30"]
      affinity:
        podAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 10
            podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values:
                  - myapp
              topologyKey: "kubernetes.io/hostname"
```

One additional thing that makes inter-pod affinity differ from node affinity is anti-affinity (`podAntiAffinity`). Anti-affinity is the inverse of the evaluated result of a statement. Take the previous co-located `Deployment`; if we change `podAffinity` to `podAntiAffinity`, it becomes a spread out deployment: **...**

affinity:
podAntiAffinity:
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 10
podAffinityTerm:
labelSelector:
matchExpressions:

- key: app
operator: In
values:
- myapp
topologyKey: "kubernetes.io/hostname"
...

The expression is quite flexible. Another example is that if we use `failure-domain.beta.kubernetes.io/zone` as `topologyKey` in the previous preference, the deployment strategy spreads the pods to different availability zones rather than only to different nodes.

Remember that, logically, you can't achieve co-located deployment with `requiredDuringSchedulingIgnoredDuringExecution` pod-affinity in the same manner as the previous example because, if there's no pod with the desired labels on any node, then none will be scheduled.

However, the cost of freedom is high. The computing complexity of pod-affinity is quite high. As a consequence, if we're running a cluster with hundreds of nodes and thousands of pods, the scheduling speed with pod-affinity would be significantly slower. Meanwhile, there are some constraints of `topologyKey` to keep the performance of scheduling with pod-affinity at a reasonable level:

- An empty `topologyKey` isn't allowed
- The `topologyKey` of pod anti-affinity of `requiredDuringSchedulingIgnoredDuringExecution` can be restricted to only use `kubernetes.io/hostname` with the `LimitPodHardAntiAffinityTopology` admission controller

Prioritizing pods in scheduling

Quality of service assures that a pod can access the appropriate resources, but the philosophy doesn't take the pod's importance into consideration. To be more precise, QoS only comes into play when a pod is scheduled, not during scheduling. Therefore, we need to introduce an orthogonal feature to denote the pod's criticality or importance.

Before 1.11, making a pod's criticality visible to Kubernetes was done by putting the pod in the `kube-system` *namespace and annotating it with* `scheduler.alpha.kubernetes.io/critical-pod`*, which is going to be deprecated in the newer version of Kubernetes. See* `https://kubernetes.io/docs/tasks/administer-cluster/guaranteed-scheduling-critical-addon-pods/` *for more information.*

The priority of a pod is defined by the priority class it belongs to. A priority class uses a 32-bit integer that is less than 1e9 (one billion) to represent the priority. A larger number means a higher priority. Numbers larger than one billion are reserved for system components. For instance, the priority class for critical components uses two billion:

```
apiVersion: scheduling.k8s.io/v1beta1
kind: PriorityClass
metadata:
  name: system-cluster-critical
value: 2000000000
description: Used for system critical pods that must run in the cluster, but can be move
```

As the priority class isn't cluster-wide (it is unnamespaced), the optional description field helps cluster users know whether they should use a class. If a pod is created without specifying its class, its priority would be the value of the default priority class or `0`, depending on whether there's a default priority class in the cluster. A default priority class is defined by adding a `globalDefault:true` field in the specification of a priority class. Note that there can only be one default priority class in the cluster. The configuration counterpart at a pod is at the `.spec.priorityClassName` path.

The principle of the priority feature is simple: if there are waiting pods to be scheduled, Kubernetes will pick higher priority pods first rather than by the order of the pods in the queue. But what if all nodes are unavailable to new pods? If pod preemption is enabled in the cluster (enabled by default from

Kubernetes 1.11 onward), then the preemption process would be triggered to make room for higher priority pods. More specifically, the scheduler will evaluate the affinity or the node selector from the pod to find eligible nodes. Afterwards, the scheduler finds pods to be evicted on those eligible nodes according to their priority. If removing *all* pods with a priority lower than the priority of the pending pod on a node can fit the pending pod, then some of those lower priority pods will be preempted.

Removing all pods sometimes causes unexpected scheduling results while considering the priority of a pod and its affinity with other pods at the same time. For example, let's say there are several running pods on a node, and a pending pod called Pod-P. Assume the priority of Pod-P is higher than all pods on the node, it can preempt every running pod on the target node. Pod-P also has a pod-affinity that requires it to be run together with certain pods on the node. Combine the priority and the affinity, and we'll find that Pod-P won't be scheduled. This is because all pods with a lower priority would be taken into consideration, even if Pod-P doesn't need all the pods to be removed to run on the node. As a result, since removing the pod associated with the affinity of Pod-P breaks the affinity, the node would be seen to not be eligible for Pod-P.

The preemption process doesn't take the QoS class into consideration. Even if a pod is in the guaranteed QoS class, it could still be preempted by best-effort pods with higher priorities. We can see how preemption works with QoS classes with an experiment. Here, we'll use `minikube` for demonstration purposes because it has only one node, so we can make sure that the scheduler will try to run everything on the same node. If you're going to do the same experiment but on a cluster with multiple nodes, affinity might help.

First, we'll need some priority classes, which can be found in the `chapter8/8-1_scheduling/prio-demo.yml` file. Just apply the file as follows:

```
$ kubectl apply -f chapter8/8-1_scheduling/prio-demo.yml
priorityclass.scheduling.k8s.io/high-prio created
priorityclass.scheduling.k8s.io/low-prio created
```

After that, let's see how much memory our `minikube` node can provide: **$ kubectl describe node minikube | grep -A 6 Allocated**
Allocated resources:
(Total limits may be over 100 percent, i.e., overcommitted.)
Resource Requests Limits
-------- -------- ------
cpu 675m (33%) 20m (1%)
memory 150Mi (7%) 200Mi (10%)

Our node has around 93% of allocatable memory. We can arrange two pods with 800 MB memory requests each in low-priority classes, and one higher priority pod with an 80 MB request and limit (and certain CPU limits). The example templates for the two deployments can be found at `chapter8/8-1_scheduling/{lowpods-gurantee-demo.yml,highpods-burstable-demo.yml}`, respectively. Create the two deployments:

```
$ kubectl apply -f lowpods-gurantee-demo.yml
deployment.apps/lowpods created
$ kubectl apply -f highpods-burstable-demo.yml
deployment.apps/highpods created
$ kubectl get pod -o wide
NAME               READY STATUS RESTARTS AGE   IP      NODE NOMINATED NODE
highpods-77dd55b549-sdpbv  1/1 Running 0  6s 172.17.0.9 minikube <none>
lowpods-65ff8966fc-xnv4v   1/1 Running 0 23s 172.17.0.7 minikube <none>
lowpods-65ff8966fc-xswjp   1/1 Running 0 23s 172.17.0.8 minikube <none>
$ kubectl describe node | grep -A 6 Allocated
Allocated resources:
  (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource  Requests      Limits
  --------  --------      ------
  cpu       775m (38%)    120m (6%)
  memory    1830Mi (96%)  1800Mi (95%)

$ kubectl get pod -o go-template --template='{{range .items}}{{printf "pod/%s: %s, prior
pod/highpods-77dd55b549-sdpbv: Burstable, priorityClass:high-prio(100000)
pod/lowpods-65ff8966fc-xnv4v: Guaranteed, priorityClass:low-prio(-1000)
pod/lowpods-65ff8966fc-xswjp: Guaranteed, priorityClass:low-prio(-1000)
```

We can see that the three pods are running on the same node. Meanwhile, the node is in danger of running out of capacity. The two lower priority pods are in the `guaranteed` QoS class, while the higher one is in the `burstable` class. Now, we just need to add one more high priority pod: **$ kubectl scale deployment --replicas=2 highpods**
deployment.extensions/highpods scaled
$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE
highpods-77dd55b549-g2m6t 0/1 Pending 0 3s <none> <none> minikube
highpods-77dd55b549-sdpbv 1/1 Running 0 20s 172.17.0.9 minikube <none>
lowpods-65ff8966fc-rsx7j 0/1 Pending 0 3s <none> <none> <none>
lowpods-65ff8966fc-xnv4v 1/1 Terminating 0 37s 172.17.0.7 minikube <none>
lowpods-65ff8966fc-xswjp 1/1 Running 0 37s 172.17.0.8 minikube <none>
$ kubectl describe pod lowpods-65ff8966fc-xnv4v
...

Events:
...
Normal Started 41s kubelet, minikube Started container
Normal Preempted 16s default-scheduler by default/highpods-77dd55b549-g2m6t on node minikube

As soon as we add a higher priority pod, one of the lower priorities is killed. From the event messages, we can clearly see that the reason the pod is terminated is that the pod is being preempted, even if it's in the `guaranteed` class. One thing to be noted is that the new lower priority pod, `lowpods-65ff8966fc-rsx7j`, is started by its deployment rather than a `restartPolicy` on the pod.

Elastically scaling

When an application reaches its capacity, the most intuitive way to tackle the problem is by adding more power to the application. However, over provisioning resources to an application is also a situation we want to avoid, and we would like to appropriate any excess resources for other applications. For most applications, scaling out is a more recommended way of resolving insufficient resources than scaling up due to physical hardware limitations. In terms of Kubernetes, from a service owner's point of view, scaling in/out can be as easy as increasing or decreasing the pods of a deployment, and Kubernetes has built-in support for performing such operations automatically, namely, the **Horizontal Pod Autoscaler** (**HPA**).

Depending on the infrastructure you're using, you can scale the capacity of the cluster in many different ways. There's an add-on ***cluster autoscaler*** *to increase or decrease a cluster's nodes based on your requirements,* `https://github.com/kubernetes/autoscaler/tree/master/cluster-autoscaler`*, if the infrastructure you are using is supported.*
Another add-on, ***vertical pod autoscaler****, can also help us to adjust the requests of a pod automatically:* `https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler`.

Horizontal pod autoscaler

An HPA object watches the resource consumption of pods that are managed by a controller (`Deployment`, `ReplicaSet`, or `StatefulSet`) at a given interval and controls the replicas by comparing the desired target of certain metrics with their real usage. For instance, suppose that we have a `Deployment` controller with two pods initially, and they are currently using 1,000 m of CPU on average while we want the CPU percentage to be 200 m per pod. The associated HPA would calculate how many pods are needed for the desired target with *2*(1000 m/200 m) = 10*, so it will adjust the replicas of the controller to 10 pods accordingly. Kubernetes will take care of the rest to schedule the eight new pods.

The evaluation interval is 15 seconds by default, and its configuration is at the controller-manager's flag, `--horizontal-pod-autoscaler-sync-period`*.*

A manifest of an HPA is as shown in the following code block: **apiVersion: autoscaling/v2beta2**
kind: HorizontalPodAutoscaler
metadata:
name: someworkload-scaler
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: someworkload
minReplicas: 1
maxReplicas: 10
metrics:
- type: Resource
resource:
name: cpu
target:
type: Utilization
averageUtilization: 50

The `.spec.scaleTargetRef` field refers to the controller that we want to scale with the

HPA and supports both `Deployment` and `StatefulSet`. The `minReplicas`/`maxReplicas` parameters set a limit to prevent a workload from over-scaling so that all resources in a cluster are exhausted. The `metrics` fields tells an HPA what metrics it should keep an eye on and what our target is for a workload. There are four valid types for a metric, which represent different sources. These are `Resource`, `Pods`, `Object`, and `External`, respectively. We'll discuss the three latter metrics in the next section.

In a `Resource` type metric, we can specify two different core metrics: `cpu` and `memory`. As a matter of fact, the source of these two metrics are the same as we saw with `kubectl top`—to be more specific, the **resource metrics** API (`metics.k8s.io`). Therefore, we'll need a metrics server deployed in our cluster to profit from the HPA. Lastly, the target type (`.resource.target.*`) specifies how Kubernetes should aggregate the recorded metrics. The supported methods are as follows:

- `Utilization`: The utilization of a pod is the ratio between a pod's actual usage and its request on a resource. That is to say, if a pod doesn't set the request on the resource we specified here, the HPA won't do anything:

```
target:
 type: Utilization
 averageUtilization: <integer>, e.g. 75
```

- `AverageValue`: This is the average value across all related pods of a resource. The denotation of the quantity is the same as how we specify a request or a limit:

```
target:
  type: AverageValue
  averageValue: <quantity>, e.g. 100Mi
```

We can also specify multiple metrics in an HPA to scale out pods based on different situations. Its resultant replicas in this case will be the largest number among all individual evaluated targets.

There is another older version (`autoscaling/v1`) of a horizontal pod autoscaler, which supports far fewer options than the v2. Please be careful about using the API version when using the HPA.

Let's walk through a simple example to see an HPA in action. The template file for this part can be found at `chapter8/8-2_scaling/hpa-resources-metrics-demo.yml`. The workload will start from one pod, and the pod will consume 150 m CPU for

three minutes: **$ kubectl apply -f chapter8/8-2_scaling/hpa-resources-metrics-demo.yml**
deployment.apps/someworkload created
horizontalpodautoscaler.autoscaling/someworkload-scaler created

After the metrics have been collected by the metrics server, we can see the scaling event of an HPA by using `kubectl describe`: **$ kubectl describe hpa someworkload-scaler**
...(some output are omitted)...
Reference: Deployment/someworkload
Metrics: (current / target)
resource cpu on pods (as a percentage of request): 151% (151m) / 50%
Min replicas: 1
Max replicas: 5
Deployment pods: 1 current / 4 desired
Conditions:
Type Status Reason Message
---- ------ ------ -------
AbleToScale True SucceededRescale the HPA controller was able to update the target scale to 4
ScalingActive True ValidMetricFound the HPA was able to successfully calculate a replica count from cpu resource utilization (percentage of request)
ScalingLimited False DesiredWithinRange the desired count is within the acceptable range
Events:
Type Reason Age From Message
---- ------ ---- ---- -------
...
Normal SuccessfulRescale 4s horizontal-pod-autoscaler New size: 4; reason: cpu resource utilization (percentage of request) above target

Although the limit is 150 m, the request is 100 m, so we can see that the measured CPU percentage is 151%. Since our target utilization is 50%, the desired replicas yielded would be *ceil(1*151/50)=4*, which can be observed at the bottom of the event message. Notice that the HPA applies ceil for decimal results. Because our workload is so greedy, the average utilization would still be

150%, even if we have three new pods. After a few seconds, the HPA decides to scale out again: **$ kubectl describe hpa someworkload-scaler**
...
Normal SuccessfulRescale 52s horizontal-pod-autoscaler New size: 5; reason: cpu resource utilization (percentage of request) above target

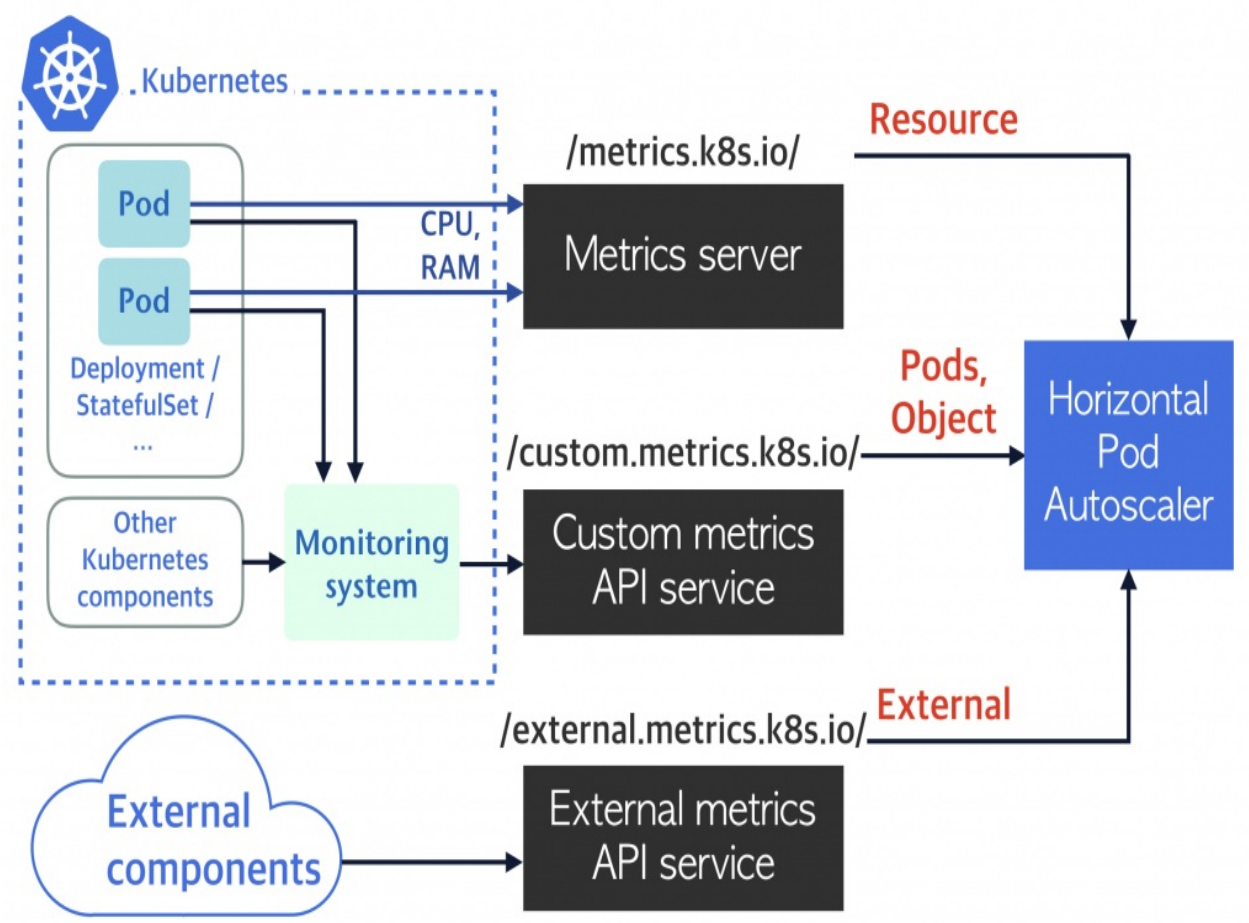
The target number this time is `5`, which is less than the estimated number, *3* (150/50) = 9*. Certainly, this is bounded by `maxReplicas`, which can save us from disrupting other containers in the cluster. As 180 seconds have passed, the workload starts to sleep, and we should see the HPA adjust the pods to one gradually:

```
$ kubectl describe hpa someworkload-scaler
Normal SuccessfulRescale 14m horizontal-pod-autoscaler New size: 3; reason: All metrics
  Normal SuccessfulRescale 13m horizontal-pod-autoscaler New size: 1; reason: All metric
```

Incorporating custom metrics

Although scaling pods on CPU and memory usages is quite intuitive, sometimes it's inadequate to cover situations such as scaling with network connections, disk IOPS, and database transactions. As a consequence, the custom metrics API and external metrics API were introduced for Kubernetes components to access metrics that aren't supported. We've mentioned that, aside from `Resource`, there are still `Pods`, `Object`, and `External` type metrics in an HPA.

The `Pods` and `Object` metrics refer to metrics that are produced by objects inside Kubernetes. When an HPA queries a metric, the related metadata such as the pod name, namespaces, and labels are sent to the custom metrics API. On the other hand, `External` metrics refer to things not in the cluster, such as the metrics of databases services from the cloud provider, and they are fetched from the external metrics API with the metric name only. Their relation is illustrated as follows:



We know that the metrics server is a program that runs inside the cluster, but what exactly are the custom metrics and external metrics API services? Kubernetes doesn't know every monitoring system and external service, so it provides API interfaces to integrate those components instead. If our monitoring system supports these interfaces, we can register our monitoring system as the provider of the metrics API, otherwise we'll need an adapter to translate the metadata from Kubernetes to the objects in our monitoring system. In the same manner, we'll need to add the implementation of the external metrics API interface to use it.

In `Chapter 7`, *Monitoring and Logging*, we built a monitoring system with Prometheus, but it doesn't support both custom and external metric APIs. We'll need an adapter to bridge the HPA and Prometheus, such as the Prometheus adapter (`https://github.com/DirectXMan12/k8s-prometheus-adapter`).

For other monitoring solutions, there is a list of adapters for different monitoring

providers: https://github.com/kubernetes/metrics/blob/master/IMPLEMENTATIONS.md#custom-metrics-api.

If none of the listed implementations support your monitoring system, there's still an API service template for building your own adapter for both custom and external metrics: https://github.com/kubernetes-incubator/custom-metrics-apiserver.

To make a service available to Kubernetes, it has to be registered as an API service under the aggregation layer. We can find out which service is the backend for an API service by showing the related `apiservices` objects:

- `v1beta1.metrics.k8s.io`
- `v1beta1.custom.metrics.k8s.io`
- `v1beta1.external.metrics.k8s.io`

We can see that the `metrics-server` service in `kube-system` is serving as the source of the `Resource` metrics: **$ kubectl describe apiservices v1beta1.metrics.k8s.io**
Name: v1beta1.metrics.k8s.io
...
Spec:
Group: metrics.k8s.io
Group Priority Minimum: 100
Insecure Skip TLS Verify: true
Service:
Name: metrics-server
Namespace: kube-system

```
  Version:           v1beta1
...
```

The example templates based on the deployment instructions of the Prometheus adapter are available in our repository (`chapter8/8-2_scaling/prometheus-k8s-adapter`) and are configured with the Prometheus service that we deployed in `Chapter 7`, *Monitoring and Logging*. You can deploy them in the following order: **$ kubectl apply -f custom-metrics-ns.yml**
$ kubectl apply -f gen-secrets.yml
$ kubectl apply -f configmap.yml
$ kubectl apply -f adapter.yml

Only default metric translation rules are configured in the example. If you want to make your own metrics available to Kubernetes, you have to custom your own configurations based on your needs with the projects' instructions (`https://github.com/DirectXMan12/k8s-prometheus-adapter/blob/master/docs/config.md`).

To verify the installation, we can query the following path to see whether any metrics are returned from our monitoring backend (`jq` is only used for formatting the result): **$ kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/" | jq '.resources[].name'**
"namespaces/network_udp_usage"
"pods/memory_usage_bytes"
"namespaces/spec_cpu_period"
...

Back to the HPA, the configuration of non-resource metrics is quite similar to resource metrics. The `Pods` type specification snippet is as follows: **...**
metrics:
- type: Pods
pods:
metric:
name: <metrics-name>
selector: <optional, a LabelSelector object>
target:
type: AverageValue or Value
averageValue or value: <quantity>
...

The definition of an `Object` metric is as follows: **...**
metrics:
- type: Object
pods:
metric:
name: <metrics-name>
selector: <optional, a LabelSelector object>
describedObject:
apiVersion: <api version>
kind: <kind>
name: <object name>
target:

type: AverageValue or Value
averageValue or value: <quantity>
...

The syntax for the `External` metrics is almost identical to the `Pods` metrics, except for the following part: **- type: External**
external:

Let's say that we specify a `Pods` metric with the metric name `fs_read` and the associated controller, which is `Deployment`, that selects `app=worker`. In that case, the HPA would make queries to the custom metric server with the following information:

- `namespace`: HPA's namespace
- Metrics name: `fs_read`
- `labelSelector`: `app=worker`

Furthermore, if we have the optional metric selector `<type>.metirc.selector` configured, it would be passed to the backend as well. A query for the previous example, plus a metric selector, `app=myapp`, could look like this:
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/pods/*/fs_read?labelSelector=app=worker&metricLabelSelector=app=myapp

After the HPA gets values for a metric, it aggregates the metric with either `AverageValue` or the raw `Value` to decide whether to scale something or not. Bear in mind that the `Utilization` method isn't supported here.

For an `Object` metric, the only difference is that the HPA would attach the information of the referenced object into the query. For example, we have the following configuration in the `default` namespace: **spec:**
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: gateway
...
metric:
name: rps
selector:

matchExpressions:
- key: app
operator: In
values:
- gwapp
describedObject:
apiVersion: extensions/v1beta1
kind: Ingress
name: cluster-ingress

The query to the monitoring backend would then be as follows:
/apis/custom.metrics.k8s.io/v1beta1/namespaces/default/ingresses.extensions/clus
ingress/rps?metricLabelSelector=app+in+(gwapp)

Notice that there wouldn't be any information about the target controller being passed, and we can't reference objects in other namespaces.

Managing cluster resources

As our resource utilization increases, it's more likely to run out of capacity for our cluster. Additionally, when lots of pods dynamically scale in and out independently, predicting the right time to add more resources to the cluster could be extremely difficult. To prevent our cluster from being paralyzed, there are various things we can do.

Resource quotas of namespaces

By default, pods in Kubernetes are resource-unbounded. The running pods might use up all of the computing or storage resources in a cluster. `ResourceQuota` is a resource object that allows us to restrict the resource consumption that a namespace could use. By setting up the resource limit, we could reduce the noisy neighbor symptom and ensure that pods can keep running.

Three kinds of resource quotas are currently supported in Kubernetes:

Compute resources	• `requests.cpu` • `requests.memory` • `limits.cpu` • `limits.memory`
Storage resources	• `requests.storage` • `<sc>.storageclass.storage.k8s.io/requests` • `<sc>.storageclass.storage.k8s.io/persistentvolumeclaims`

Object count	• `count/<resource>.<group>`, for example, the following:
	  ◦ `count/deployments.apps`
	  ◦ `count/persistentvolumeclaims`
	• `services.loadbalancers`
	• `services.nodeports`

Compute resources are quite intuitive, restricting the sum of given resources

across all related objects. One thing that should be noted is that once a compute quota has been set, any creation of pods that don't have resource requests or limits will be rejected.

For storage resources, we can associate storage classes in a quota. For example, we can have the two quotas, `fast.storageclass.storage.k8s.io/requests: 100G` and `meh.storageclass.storage.k8s.io/requests: 700G`, configured simultaneously to distinguish the resource classes that we installed for reasonably allocating resources.

Existing resources won't be affected by newly created resource quotas. If the resource creation request exceeds the specified `ResourceQuota`, the resources won't be able to start up.

Creating a ResourceQuota

The syntax of `ResourceQuota` is shown as follows. Note that it's a namespaced object: **apiVersion: v1**
kind: ResourceQuota
metadata:
name: <name>
spec:
hard:
<quota_1>: <count> or <quantity>
...
scopes:
- <scope name>
...
scopeSelector:
- matchExpressions:
scopeName: PriorityClass
operator: <In, NotIn, Exists, DoesNotExist>
values:
- <PriorityClass name>

Only `.spec.hard` is a required field; `.spec.scopes` and `.spec.scopeSelector` are optional. The quota names for `.spec.hard` are those listed in the preceding table, and only counts or quantities are valid for their values. For example, `count/pods: 10` limits pod counts to 10 in a namespace, and `requests.cpu: 10000m` makes sure that we don't have more requests than the amount specified.

The two optional fields are used to associated a resource quota on certain scopes, so only objects and usages within the scope would be taken into account for the associated quota. Currently, there are four different scopes for the `.spec.scopes` field:

- `Terminating`/`NotTerminating`: The `Terminating` scope matches pods with their `.spec.activeDeadlineSeconds >= 0`, while `NotTerminating` matches pods without the field set. Bear in mind that `Job` also has the deadline field, but it won't be

propagated to the pods created by the `Job`.

- `BestEffort`/`NotBestEffort`: The former works on pods at the `BestEffort` QoS class and another one is for pods at other QoS classes. Since setting either requests or limits on a pod would elevate the pod's QoS class to non-`BestEffort`, the `BestEffort` scope doesn't work on compute quotas.

Another scope configuration, `scopeSelector`, is for choosing objects with a more free and flexible syntax, despite the fact that only `PriorityClass` is supported as of Kubernetes 1.13. With `scopeSelector`, we're able to bind a resource quota to certain priority classes with a corresponding `PriorityClassName`.

So, let's see how a quota works in an example, which can be found at `chapter8/8-3_management/resource_quota.yml`. In the template, two resource quotas restrict pod numbers (`quota-pods`) and resources requests (`quota-resources`) for `BestEffort` and other QoS, respectively. In this configuration, the desired outcome is confining workloads without requests by pod numbers and restricting the resource amount for those workloads that have requests. As a result, both jobs, `capybara` and `politer-capybara`, in the example, which set high parallelism but in different QoS classes, will be capped by two different resource quotas: **$ kubectl apply -f resource_quota.yml**
namespace/team-capybara created
resourcequota/quota-pods created
resourcequota/quota-resources created
job.batch/capybara created
job.batch/politer-capybara created

$ kubectl get pod -n team-capybara
NAME READY STATUS RESTARTS AGE
capybara-4wfnj 0/1 Completed 0 13s
politer-capybara-lbf48 0/1 ContainerCreating 0 13s
politer-capybara-md9c7 0/1 ContainerCreating 0 12s
politer-capybara-xkg7g 1/1 Running 0 12s
politer-capybara-zf42k 1/1 Running 0 12s

As we can see, only a few pods are created for the two jobs, even though their parallelism is 20 pods. The messages from their controller confirms that they reached the resource quota:

```
$ kubectl describe jobs.batch -n team-capybara capybara
...
Events:
  Type     Reason            Age   From            Message
  ----     ------            ----  ----            -------
  Normal   SuccessfulCreate  98s   job-controller  Created pod: capybara-4wfnj
  Warning  FailedCreate      97s   job-controller  Error creating: pods "capybara-ds7zk"
...

$ kubectl describe jobs.batch -n team-capybara politer-capybara
...
Events:
  Type     Reason            Age                From            Message
  ----     ------            ----               ----            -------
  Warning  FailedCreate      86s                job-controller  Error creating: pods "p
...
```

We can also find the consumption stats with `describe` on `Namespace` or `ResourceQuota`:

from namespace
$ kubectl describe namespaces team-capybara
Name: team-capybara
...
Resource Quotas
Name: quota-pods
Scopes: BestEffort
*** Matches all pods that do not have resource requirements set. These pods have a best effort quality of service.**
Resource Used Hard
-------- --- ---
count/pods 1 1

Name: quota-resources
Scopes: NotBestEffort
*** Matches all pods that have at least one resource requirement set. These pods have a burstable or guaranteed quality of service.**
Resource Used Hard
-------- --- ---
requests.cpu 100m 100m
requests.memory 100M 1Gi

No resource limits.

from resourcequotas
$ kubectl describe -n team-capybara resourcequotas

Name: quota-pods
Namespace: team-capybara
...
(information here is the same as above)

Request pods with default compute resource limits

We could also specify default resource requests and limits for a namespace. The default setting will be used if we don't specify the requests and limits during pod creation. The trick is using a `LimitRange` object, which contains a set of `defaultRequest` (requests) and `default` (limits).

`LimitRange` is controlled by the `LimitRange` admission controller plugin. Be sure that you enable this if you launch a self-hosted solution. For more information, check out the Admission Controller section of this chapter.

The following is the example that can be found at `chapter8/8-3_management/limit_range.yml`: **apiVersion: v1**
kind: LimitRange
metadata:
name: limitcage-container
namespace: team-capybara
spec:
limits:
- default:
cpu: 0.5
memory: 512Mi
defaultRequest:
cpu: 0.25
memory: 256Mi
type: Container

When we launch pods inside this namespace, we don't need to specify the `cpu` and `memory` requests and limits anytime, even if we have a total limitation set inside the `ResourceQuota`.

We can also set minimum and maximum CPU and memory values for containers in `LimitRange`. `LimitRange` acts differently from default values. Default values are only used if a pod spec doesn't contain any requests and limits. The minimum

and maximum constraints are used to verify whether a pod requests too many resources. The syntax is `spec.limits[].min` and `spec.limits[].max`. If the request exceeds the minimum and maximum values, `forbidden` will be thrown from the server: **...**

```
spec:
limits:
- max:
cpu: 0.5
memory: 512Mi
min:
cpu: 0.25
memory: 256Mi
```

Other than `type: Container`, there are also `Pods` and `PersistentVolumeClaim` types of `LimitRange`. For a `Container` type limit range, it asserts containers of `Pods` individually, so the `Pod` limit range checks all containers in a pod as a whole. But unlike the `Container` limit range, `Pods` and `PersistentVolumeClaim` limit ranges don't have `default` and `defaultRequest` fields, which means they are used only for verifying the requests from associated resource types. The resource for a `PersistentVolumeClaim` limit range is `storage`. You can find a full definition at the `chapter8/8-3_management/limit_range.yml` template.

Aside from absolute requests and the limit constraints mentioned previously, we can also restrict a resource with a ratio: `maxLimitRequestRatio`. For instance, if we have `maxLimitRequestRatio:1.2` on the CPU, then a pod with a CPU of `requests:50m` and a CPU of `limits: 100m` would be rejected as *100m/50m > 1.2*.

As with `ResourceQuota`, we can view the evaluated settings by describing either `Namespace` or `LimitRange`: **$ kubectl describe namespaces <namespace name>**

```
...
Resource Limits
Type Resource Min Max Default Request Default Limit Max Limit/Request Ratio
---- -------- --- --- --------------- ------------- -----------------------
Container memory - - 256Mi 512Mi -
Container cpu - - 250m 500m -
Pod cpu - - - - 1200m
PersistentVolumeClaim storage 1Gi 10Gi - - -
```

Node administration

No matter how carefully we allocate and manage resources in our cluster, there's always a chance that resource exhaustion might happen on a node. Even worse than this, rescheduled pods from a dead host could take down other nodes and cause all nodes to oscillate between stable and unstable states. Fortunately, we are using Kubernetes, and kubelet has ways of dealing with these unfortunate events.

Pod eviction

To keep a node stable, kubelet reserves some resources as buffers to ensure it can take actions before a node's kernel acts. There are three configurable segregations or thresholds for different purposes:

- `kube-reserved`: Reserves resources for node components of Kubernetes
- `system-reserved`: Reserves resources for system daemons
- `eviction-hard`: A threshold for when to evict pods

Hence, a node's allocatable resources are calculated by the following equation:
Allocatable =
[Node Capacity] - <kube-reserved> - <system-reserved> - <eviction-hard>

The Kubernetes and system reservations apply on `cpu`, `memory`, and `ephemeral-storage` resources, and they're configured by the kubelet flags, `--kube-reserved` and `--system-reserved`, with syntax such as `cpu=1,memory=500Mi,ephemeral-storage=10Gi`. Aside from the resource configurations, manually assigning the pre-configured `cgroups` name as `--kube-reserved-cgroup=<cgroupname>` and `--system-reserved-cgroup=<cgroupname>` is required. Also, as they are implemented with `cgroups`, it's possible that system or Kubernetes components will get capped by inappropriate small resource reservations.

The eviction threshold takes effect on five critical eviction signals:

Eviction signal	Default values
• `memory.available` • `nodefs.available` • `nodefs.inodesFree` • `imagefs.available` • `imagefs.inodesFree`	• `memory.available<100Mi` • `nodefs.available<10%` • `nodefs.inodesFree<5%` • `imagefs.available<15%`

We can verify the default value at the node's `/configz` endpoint: **## run a proxy at**

the background
$ kubectl proxy &
pipe to json_pp/jq/fx for a more beautiful formatting
**$ curl -s http://127.0.0.1:8001/api/v1/nodes/minikube/proxy/configz | **
jq '.[].evictionHard'
{
"imagefs.available": "15%",
"memory.available": "100Mi",
"nodefs.available": "10%",
"nodefs.inodesFree": "5%"
}

We can also check the difference between node's allocatable resource and total capacity, which should match the allocatable equation we mentioned previously. A `minikube` node doesn't have Kubernetes or system reservations set by default, so the difference in the memory would be the `memory.available` threshold:

```
$ VAR=$(kubectl get node minikube -o go-template --template='{{printf "%s-%s\n" .status.capacity.memory
.status.allocatable.memory}}') && printf $VAR= && tr -d 'Ki' <<< ${VAR} | bc
2038700Ki-1936300Ki=102400
```

If any of the resources noted in the eviction threshold starves, a system would start to behave strangely, which could endanger the stability of a node. Therefore, once the node condition breaks a threshold, kubelet marks the node with either of the following two conditions:

- `MemoryPressure`: If the `memory.available` exceeds its threshold
- `DiskPressure`: If any `nodefs.*/imagefs.*` go beyond their threshold

The Kubernetes scheduler and kubelet will adjust their strategy to the node as long as they perceive the condition. The scheduler will stop scheduling `BestEffort` pods onto the node if the node is experiencing memory pressure, and stop scheduling all pods to the node if there's an undergoing disk pressure condition. kubelet will take immediate actions to reclaim the starving resource, that is, evict pods on a node. Unlike a killed pod, which could only be restarted on the same node by its `RestartPolicy`, an evicted pod would eventually be rescheduled on another node if there's sufficient capacity.

Aside from hard thresholds, we can configure soft thresholds for signals with `eviction-soft`. When a soft threshold is reached, kubelet will wait an amount of time first (`eviction-soft-grace-period`) and afterwards it would try to gracefully remove pods with a maximum waiting time (`eviction-max-pod-grace-period`). For

example, let's say we have the following configurations:

- `eviction-soft=memory.available<1Gi`
- `eviction-soft-grace-period=memory.available=2m`
- `eviction-max-pod-grace-period=60s`

In this case, before kubelet acts, it would wait two minutes if the node's available memory is less then 1 Gi but is still in line with the preceding hard eviction threshold. Afterwards, kubelet would start to purge pods on the node. If a pod doesn't exit after 60 seconds, kubelet will kill it straight away.

The eviction order of pods is ranked using the pod's QoS class on the starved resource and then the pod's priority class. Let's assume that kubelet now perceives the `MemoryPressure` condition, so it starts comparing all their attributes and real usage on memory:

Pod	Request	Real usage	Above request	QoS	Priority
A	100Mi	50Mi	-	`Burstable`	100
B	100Mi	200Mi	100Mi	`Burstable`	10
C	-	150Mi	150Mi	`BestEffort`	100
D	-	50Mi	50Mi	`BestEffort`	10
E	100Mi	100Mi	-	`Guaranteed`	50

The comparison begins with whether a pod uses more memory than requested, and this is the case for B, C, and D. Bear in mind that although B is in the `Burstable` class, it's still being picked in the first group of victims due to its excess consumption of starved resources. The next thing to consider is priority, so B and D will be picked. Lastly, since B's memory usages in the preceding request are more than D, it will be the first pod to be killed.

Most of the time, pods using resources within their requested range, such as A and E, wouldn't be evicted. But if the node's non-Kubernetes components exhausted their memory and have to be moved to other node, they will be ranked

by their priority classes. The final eviction order of the five pods would be D, B, C, E, and then A.

If kubelet can't catch up with releasing the node memory before the node's OOM killer acts, QoS classes still can preserve rank using the pre-assigned OOM scores (`oom_score_adj`) on the pods we mentioned earlier in this chapter. The OOM score is related to processes and is visible to the Linux OOM killer. The higher the score, the more likely a process is to be killed first. Kubernetes assigns the score -998 to `Guaranteed` pods and 1,000 to `BestEffort` pods. `Burstable` pods are assigned a score between 2 and 999 based on their memory requests; the more requested memory, the lower the score they get.

Taints and tolerations

A node can decline pods by taints unless pods tolerate all the taints a node has. Taints are applied to nodes, while tolerations are specific to pods. A taint is a triplet with the form `key=value:effect`, and the effect could be `PreferNoSchedule`, `NoSchedule`, or `NoExecute`.

Suppose that we have a node with some running pods and those running pods don't have the toleration on a taint, `k_1=v_1`, and different effects result in the following conditions:

- `NoSchedule`: No new pods without tolerating `k_1=v_1` will be placed on the node
- `PreferNoSchedule`: The scheduler would try not to place new pods without tolerating `k_1=v_1` to the node
- `NoExecute`: The running pods would be repelled immediately or after a period that is specified in the pod's `tolerationSeconds` has passed

Let's see an example. Here, we have three `nodes`: **$ kubectl get nodes**
NAME STATUS ROLES AGE VERSION
gke-mycluster-default-pool-1e3873a1-jwvd Ready <none> 2m v1.11.2-gke.18
gke-mycluster-default-pool-a1eb51da-fbtj Ready <none> 2m v1.11.2-gke.18
gke-mycluster-default-pool-ec103ce1-t0l7 Ready <none> 2m v1.11.2-gke.18

Run a `nginx` pod: **$ kubectl run --generator=run-pod/v1 --image=nginx:1.15 ngx**
pod/ngx created

$ kubectl describe pods ngx
Name: ngx
Node: gke-mycluster-default-pool-1e3873a1-jwvd/10.132.0.4
...
Tolerations: node.kubernetes.io/not-ready:NoExecute for 300s
node.kubernetes.io/unreachable:NoExecute for 300s

By the pod description, we can see it's been put on the `gke-mycluster-default-pool-`

`1e3873a1-jwvd` node, and it has two default tolerations. Literally, this means if the node becomes not ready or unreachable, we have to wait for 300 seconds before the pod is evicted from the node. These two tolerations are applied by the `DefaultTolerationSeconds` admission controller plugin. Now, we add a taint to the node with `NoExecute`:: **$ kubectl taint nodes gke-mycluster-default-pool-1e3873a1-jwvd **
experimental=true:NoExecute
node/gke-mycluster-default-pool-1e3873a1-jwvd tainted

Since our pod doesn't tolerate `experimental=true` and the effect is `NoExecute`, the pod will be evicted from the node immediately and restarted somewhere if it's managed by controllers. Multi-taints can also be applied to a node. The pods must match all the tolerations to run on that node. The following is an example that could pass the tainted node: **$ cat chapter8/8-3_management/pod_tolerations.yml**
apiVersion: v1
kind: Pod
metadata:
name: pod-with-tolerations
spec:
containers:
- name: web
image: nginx
tolerations:
- key: "experimental"
value: "true"
operator: "Equal"
effect: "NoExecute"
$ kubectl apply -f chapter8/8-3_management/pod_tolerations.yml
pod/pod-with-tolerations created

$ kubectl get pod -o wide
NAME READY STATUS RESTARTS AGE IP NODE
pod-with-tolerations 1/1 Running 0 7s 10.32.1.4 gke-mycluster-default-pool-1e3873a1-jwvd

As we can see, the new pod can now run on the tainted node, `gke-mycluster-`

`default-pool-1e3873a1-jwvd`.

As well as the `Equal` operator, we can also use `Exists`. In that case, we don't need to specify the value field. As long as the node is tainted with the specified key and the desired effect matches, the pod is eligible to run on that tainted node.

According to a node's running status, some taints could be populated by the node controller, kubelet, cloud providers, or cluster admins to move pods from the node. These taints are as follows:

- `node.kubernetes.io/not-ready`
- `node.kubernetes.io/unreachable`
- `node.kubernetes.io/out-of-disk`
- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/network-unavailable`
- `node.cloudprovider.kubernetes.io/uninitialized`
- `node.kubernetes.io/unschedulable`

If there's any critical pod that needs to be run even under those circumstances, we should explicitly tolerate the corresponding taints. For example, pods managed by `DaemonSet` will tolerate the following taints in `NoSchedule`:

- `node.kubernetes.io/memory-pressure`
- `node.kubernetes.io/disk-pressure`
- `node.kubernetes.io/out-of-disk`
- `node.kubernetes.io/unschedulable`
- `node.kubernetes.io/network-unavailable`

For node administrations, we can utilize `kubectl cordon <node_name>` to taint the node as unschedulable (`node.kubernetes.io/unschedulable:NoSchedule`), and use `kubectl uncordon <node_name>` to revert the action. Another command, `kubectl drain`, would evict pods on the node and also mark the node as unschedulable.

Summary

In this chapter, we explored topics surrounding how Kubernetes manages cluster resources and schedules our workloads. With concepts such as Quality of Services, priority, and node out of resource handling in mind, we can optimize our resource utilization while keeping our workloads stable. Meanwhile, `ResourceQuota` and `LimitRange` add additional layers of shields to running workloads in a multi-tenant but sharing resources environment. With all of this protection we've built, we can confidently count on Kubernetes to scale our workloads with autoscalers and maximize resource utilization to the limit.

In `Chapter 9`, *Continuous Delivery*, we're moving on and setting up a pipeline to deliver our product continuously in Kubernetes.

Continuous Delivery

At the beginning of this book, we started by containerizing our applications, orchestrating them with Kubernetes, persisting their data, and exposing our service to the outside world. Later, we gained more confidence in our services by setting up monitoring and logging, and we made them scale in and out in a fully automatic manner. We'd now like to set our service on course by delivering our latest features and improvements to our services continuously in Kubernetes. We'll learn about the following topics in this chapter:

- Updating Kubernetes resources
- Setting up a delivery pipeline
- How to improve the deployment process

Updating resources

Continuous Delivery (**CD**), as we described in `Chapter 1`, *Introduction to DevOps*, is a set of operations including **Continuous Integration** (**CI**) and the ensuing deployment tasks. The CI flow is made up of elements such as version control systems, buildings, and different levels of validation, which aim to eliminate the effort to integrate every change in the main release line. Tools to implement functions are usually at the application layer, which might be independent to the underlying infrastructure. Even so, when it comes to the deployment part, understanding and dealing with infrastructure is still inevitable. Deployment tasks are tightly coupled with the platform our application is running on, no matter which practice, continuous delivery or continuous deployment, we're implementing. For instance, in an environment where the software runs on baremetal or virtual machines, we'd utilize configuration management tools, orchestrators, and scripts to deploy our software. However, if we're running our service on an application platform such as Heroku, or even in the serverless pattern, designing the deployment pipeline would be a totally different story. All in all, the goal of deployment tasks is about making sure our software works properly in the right places. In Kubernetes, it's about knowing how to correctly update resources, in particular pods.

Triggering updates

In `Chapter 3`, *Getting Started with Kubernetes*, we discussed the rolling update mechanism of the pods in a deployment. Let's recap what happens after the update process is triggered:

- The deployment creates a new `ReplicaSet` with `0` pods, according to the updated manifest
- The new `ReplicaSet` is scaled up gradually while the previous `ReplicaSet` keeps shrinking
- The process ends after all of the old pods are replaced

This mechanism is implemented automatically by Kubernetes, meaning we don't have to supervise the updating process. To trigger it, all we need to do is inform Kubernetes that the pod specification of a deployment is updated; that is to say, we modify the manifest of a resource in Kubernetes. Suppose we have a deployment, `my-app` (see `ex-deployment.yml` under the example directory for this section), we can modify the manifest with the sub–commands of `kubectl` as follows:

- `kubectl patch`: This patches a manifest of an object partially according to the input JSON parameter. If we'd like to update the image of `my-app` from `alpine:3.7` to `alpine:3.8`, it'd be as follows:

```
$ kubectl patch deployment my-app -p '{"spec":{"template":{"spec":{"containers":
```

- `kubectl set`: This makes changes to certain properties of an object. This is a shortcut to change some properties directly. The image of `deployment` is one of the properties it supports:

```
$ kubectl set image deployment my-app app=alpine:3.8
```

- `kubectl edit`: This opens an editor and dumps the current manifest so that we can edit it interactively. The modified manifest will take effect immediately after being saved. To change the default editor for this command, use the `EDITOR` environment variable. For example, `EDITOR`="`code --wait`" kubectl edit deployments my-app opens Visual Studio Code.

- `kubectl replace`: This replaces one manifest with another submitted template file. If a resource isn't created yet or contains properties that can't be changed, it yields errors. For instance, there are two resources in our example template, `ex-deployment.yml`, namely the deployment, `my-app`, and its `Service`, `my-app-svc`. Let's replace them with a new specification file:

```
$ kubectl replace -f ex-deployment.yml
deployment.apps/my-app replaced
The Service "my-app-svc" is invalid: spec.clusterIP: Invalid value: "": field is
$ echo $?
1
```

  After they're replaced, we see that the error code is `1` as expected, so we are updating `deployment` rather than `Service`. This behavior is particularly important when composing automation scripts for the CI/CD flow.

- `kubectl apply`: This applies the manifest file anyway. In other words, if a resource exists in Kubernetes, it'd be updated; otherwise, it'd be created. When `kubectl apply` is used to create resources, it is roughly equal to `kubectl create --save-config` in terms of functionality. The applied specification file would be saved to the annotation field, `kubectl.kubernetes.io/last-applied-configuration`, accordingly, and we can manipulate it with the sub-commands `edit-last-applied`, `set-last-applied`, and `view-last-applied`. For example, we can view the template we submitted previously with the following:

```
$ kubectl apply -f ex-deployment.yml view-last-applied
```

The saved manifest information will be exactly the same as what we've sent, unlike the information we retrieve via `kubectl get <resource> -o <yaml or json>`, which contains an object's live status, in addition to specifications.

Although in this section we are only focusing on manipulating a deployment, the commands here also work for updating all other Kubernetes resources, such as `service` and `role`.

Depending on the convergence speed of `etcd`, changes to `ConfigMap` and `secret` usually take a couple of seconds to propagate to pods.

The recommended way to interact with a Kubernetes API server is by using `kubectl`. If you're in a confined environment or you want to implement your own operator controllers, there are also RESTful APIs for manipulating resources in Kubernetes. For instance, the `kubectl patch` command we used before would look

as follows: **$ curl -X PATCH -H 'Content-Type: application/strategic-merge-patch+json' --data '{"spec":{"template":{"spec":{"containers":[{"name":"app","image":"alpine:3.8"}]}}}}' 'https://$KUBEAPI/apis/apps/v1/namespaces/default/deployments/my-app'**

Here, the `$KUBEAPI` variable is the endpoint of the API server. See the API reference material for more information: https://kubernetes.io/docs/reference/kubernetes-api/.

Managing rollouts

Once the rollout process is triggered, Kubernetes silently completes all tasks in the background. Let's try some hands-on experiments. Again, the rolling update process won't be triggered even if we've modified something with the commands mentioned earlier, unless the associated pod's specification is changed. The example we prepared is a simple script that will respond to any request with its hostname and the Alpine version it runs on. First, we create `deployment` and check its response in another Terminal: **$ kubectl apply -f ex-deployment.yml**
deployment.apps/my-app created
service/my-app-svc created
$ kubectl proxy &
[1] 48334
Starting to serve on 127.0.0.1:8001
switch to another terminal, #2
$ while :; do curl
http://localhost:8001/api/v1/namespaces/default/services/my-app-svc:80/proxy/; sleep 1; done
my-app-5fbdb69f94-5s44q-v-3.7.1 is running...
my-app-5fbdb69f94-g7k7t-v-3.7.1 is running...
...

Now, we change its image to another version and see what the responses are: **## go back to terminal#1**
$ kubectl set image deployment.apps my-app app=alpine:3.8
deployment.apps/my-app image updated

switch to terminal#2
...
my-app-5fbdb69f94-7fz6p-v-3.7.1 is running...
my-app-6965c8f887-mbld5-v-3.8.1 is running...
...

Messages from version 3.7 and 3.8 are interleaved until the updating process ends. In order to immediately determine the status of updating processes from

Kubernetes, rather than polling the service endpoint, we can use `kubectl rollout` to manage the rolling update process, including inspecting the progress of ongoing updates. Let's see the acting `rollout` with the `status` sub-command: **## if the previous rollout has finished,**
you can make some changes to my-app again:

$ kubectl rollout status deployment my-app
Waiting for deployment "my-app" rollout to finish: 3 out of 5 new replicas have been updated...
...
Waiting for deployment "my-app" rollout to finish: 3 out of 5 new replicas have been updated...
Waiting for deployment "my-app" rollout to finish: 3 of 5 updated replicas are available...
Waiting for deployment "my-app" rollout to finish: 3 of 5 updated replicas are available...
Waiting for deployment "my-app" rollout to finish: 3 of 5 updated replicas are available...
deployment "my-app" successfully rolled out

At this moment, the output at `terminal#2` should be from version 3.6. The `history` sub-command allows us to review previous changes to `deployment`: **$ kubectl rollout history deployment.app my-app**
deployment.apps/my-app
REVISION CHANGE-CAUSE
1 <none>
2 <none>

However, the `CHANGE-CAUSE` field doesn't show any useful information that helps us to see the details of the revision. To profit from the rollout history feature, add a `--record` flag after each command that leads to a change, such as `apply` or `patch`. `kubectl create` also supports the `record` flag.

Let's make some changes to the `deployment`, such as modifying the `DEMO` environment variable on pods in `my-app`. As this causes a change in the pod's specification, `rollout` will start right away. This sort of behavior allows us to trigger an update without building a new image. For simplicity, we use `patch` to modify the variable: **$ kubectl patch deployment.apps my-app -p '{"spec":**

{"template":{"spec":{"containers":[{"name":"app","env":
[{"name":"DEMO","value":"1"}]}]}}}}' --record
deployment.apps/my-app patched
$ kubectl rollout history deployment.apps my-app
deployment.apps/my-app
REVISION CHANGE-CAUSE
1 <none>
2 <none>
3 kubectl patch deployment.apps my-app --patch={"spec":{"template":
{"spec":{"containers":[{"name":"app","env":
[{"name":"DEMO","value":"1"}]}]}}}} --record=true

`CHANGE-CAUSE` of `REVISION 3` notes the committed command clearly. Only the command will be recorded, which means that any modification inside `edit`/`apply`/`replace` won't be marked down explicitly. If we want to get the manifest of the former revisions, we could retrieve the saved configuration, as long as our changes are made with `apply`.

The `CHANGE-CAUSE` field is actually stored in the `kubernetes.io/change-cause` annotation of an object.

For various reasons, we sometimes want to roll back our application even if the `rollout` is successful to a certain extent. This can be achieved with the `undo` sub-command :

```
$ kubectl rollout undo deployment my-app
```

The whole process is basically identical to updating—that is, applying the previous manifest—and performing a rolling update. We can also utilize the `--to-revision=<REVISION#>` flag to roll back to a specific version, but only retained revisions are able to be rolled back. Kubernetes determines how many revisions it keeps according to the `revisionHistoryLimit` parameter in the `deployment` object.

The progress of an update is controlled by `kubectl rollout pause` and `kubectl rollout resume`. As their names indicate, they should be used in pairs. Pausing a deployment involves not only stopping an ongoing `rollout`, but also freezing any triggering of updates even if the specification is modified, unless it's resumed.

Updating DaemonSet and StatefulSet

Kubernetes supports various ways to orchestrate pods for different types of workloads. In addition to deployments, we also have `DaemonSet` and `StatefulSet` for long-running and non-batch workloads. As pods spawned by these have more constraints than the ones from deployments, there are a few caveats that we need to be aware of in order to handle their updates.

DaemonSet

`DaemonSet` is a controller designed for system daemons, as its name suggests. Consequently, a `DaemonSet` controller launches and maintains exactly one pod per node; the total number of pods launched by a `DaemonSet` controller adheres to the number of nodes in a cluster. Due to this limitation, updating `DaemonSet` isn't as straightforward as updating a deployment. For instance, `deployment` has a `maxSurge` parameter (`.spec.strategy.rollingUpdate.maxSurge`) that controls how many redundant pods over the desired number can be created during updates, but we can't employ the same strategy for pods managed by `DaemonSet`. Because daemon pods usually come with special concerns that might occupy a host's resources, such as ports, it could result in errors if we have two or more system pods simultaneously on a node. As such, the update is in the form that a new pod is created after the old pod is terminated on a host.

Kubernetes implements two update strategies for `DaemonSet`:

- `OnDelete`: Pods are only updated after they are deleted manually.
- `RollingUpdate`: This works like `OnDelete`, but the deletion of pods is performed by Kubernetes automatically. There is one optional parameter, `.spec.updateStrategy.rollingUpdate.maxUnavailable`, which is similar to the one in `deployment`. Its default value is `1`, which means Kubernetes replaces one pod at a time, node by node.

You can find an example that demonstrates how to write a template of `DaemonSet` at https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/blob/master/chapter9/9-1_updates/ex-daemonset.yml. The update strategy is set at the `.spec.updateStrategy.type` path, and its default is `RollingUpdate`. The way to trigger the rolling update is identical to the way in which we trigger a deployment. We can also utilize `kubectl rollout` to manage rollouts of our `DaemonSet` controller, but `pause` and `resume` aren't supported.

StatefulSet

The updating of `StatefulSet` and `DaemonSet` are pretty much the same; they don't create redundant pods during an update and their update strategies also behave in a similar way. There's a template file at `9-1_updates/ex-statefulset.yml` that you can use for practice. The options of the update strategy are set at the `.spec.updateStrategy.type` path:

- `OnDelete`: Pods are only updated after they're manually deleted.
- `RollingUpdate`: Like rolling updates for other controllers, Kubernetes deletes and creates pods in a managed fashion. Kubernetes knows the order matters in `StatefulSet`, so it replaces pods in reverse order. Say we have three pods in `StatefulSet`: `my-ss-0`, `my-ss-1`, and `my-ss-2`. The update order will start at `my-ss-2` and run to `my-ss-0`. The deletion process doesn't respect the pod management policy of `StatefulSet`; even if we set the pod management policies to `Parallel`, the updates would still be performed one by one.

The only parameter for the `RollingUpdate` type is `partition` (`.spec.updateStrategy.rollingUpdate.partition`). If this is specified, any pod with an ordinal less than the partition number will keep its current version and won't be updated. For instance, if we set `partition` to `1` in a `StatefulSet` with three pods, only pod-1 and pod-2 would be updated after a rollout. This parameter allows us to control the progress to a certain extent and it's particularly handy for scenarios such as waiting for data synchronization, carrying out a canary test, or staging an update.

Building a delivery pipeline

Implementing a CD pipeline for containerized applications is quite simple. Let's recall what practices we learned about Docker and Kubernetes so far and organize those practices into the CD pipeline. Suppose we've finished our code, Dockerfile, and corresponding Kubernetes templates. To deploy these to our cluster, we'd go through the following steps:

1. `docker build`: Produces an executable and immutable artifact
2. `docker run`: Verifies whether the build works with a simple test
3. `docker tag`: Tags the build with meaningful versions if it's good

4. `docker push`: Moves the build to the `artifacts` repository for distribution
5. `kubectl apply`: Deploys the build to a desired environment
6. `kubectl rollout status`: Tracks the progress of deployment tasks

This is all we need for a simple but viable delivery pipeline.

Here, we use the term continuous delivery instead of continuous deployment because there are still gaps between the steps described previously, which can be implemented as either human-controlled or fully automatic deployments. The consideration may differ from team to team.

Choosing tools

The steps we're going to implement are quite simple. However, when it comes to chaining them as a pipeline, there's no generic pipeline that suits all scenarios. It might differ by factors such as the form of an organization, the development workflow a team is running, or the interaction between the pipeline and other systems in the existing infrastructure. In light of this, setting a goal and choosing tools are the first things we have to think about.

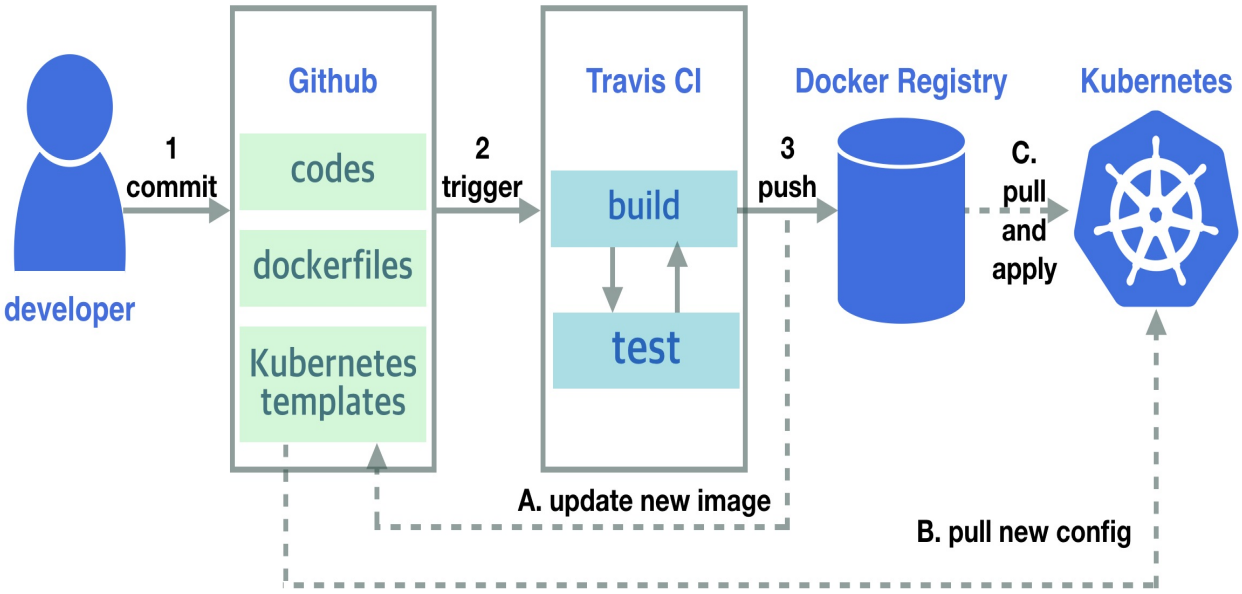
Generally speaking, to make the pipeline ship builds continuously, we'll need at least three kinds of tools: version control systems, build servers, and a repository for storing container artifacts. In this section, we will set a reference CD pipeline based on the SaaS tools we introduced in previous chapters:

- GitHub (`https://github.com`)
- Travis CI (`https://travis-ci.com`)
- Docker Hub (`https://hub.docker.com`)

All of these are free for open source projects. Certainly, there are numerous alternatives for each tool we used here, such as GitLab for VCS, hosting Jenkins for CI, or even dedicated deployment tools such as Spinnaker (`https://www.spinnaker.io/`). In addition to these large building blocks, we can also benefit from tools such as Helm (`https://github.com/kubernetes/helm`) to help us to organize templates and their instantialized releases. All in all, it's up to you to choose the tools that best suit your needs. We'll focus on how these fundamental components interact with our deployments in Kubernetes.

End-to-end walk-through of the delivery pipeline

The following diagram is our CD flow based on the three services mentioned earlier:



The workflow for code integration is as follows:

1. Code is committed to a repository on GitHub.
2. The commit triggers a build job on Travis CI:
    - A Docker image will be built.
    - To ensure that the quality of the build is solid and ready to be integrated, different levels of tests are usually performed at this stage on the CI server. Furthermore, as running an application stack with Docker Compose or Kubernetes is easier than ever, running tests involving many components in a build job is also possible.
3. The verified image is tagged with identifiers and pushed to Docker Hub.

As for the deployment in Kubernetes, this can be as simple as updating the image path in a template and then applying the template to a production cluster, or as complex as a series of operations including traffic distribution and canary deployment. In our example, a rollout starts from manually publishing a new Git SemVer tag, and the CI script repeats the same flow as in the integration part until the image pushing step. As a CI server sometimes may not be able to touch the production environment, we put an agent inside our cluster to watch and apply the changes in the configuration branch.

*A dedicated config repository is a popular pattern for segregating an application and its infrastructure. There are many **Infrastructure as Code** (**IaC**) tools that help us to express infrastructure and their states in a way that can be recorded in a version control system. Additionally, by tracking everything in a version control system, we can translate every change made to the infrastructure into Git operations. For the sake of simplicity, we use another branch in the same repository for the config changes.*

Once the agent observes the change, it pulls the new template and updates the corresponding controller accordingly. Finally, the delivery is finished after the rolling update process of Kubernetes ends.

The steps explained

Our example, `okeydokey`, is a web service that always echoes `OK` to every request, and the code as well as the files for deployment are committed in our repository over in GitHub: `https://github.com/DevOps-with-Kubernetes/okeydokey`.

Before configuring our builds on Travis CI, let's create an image repository in Docker Hub first for later use. After signing in to Docker Hub, press the huge Create Repository button at the top right, and then follow the steps onscreen to create one. The image repository of `okeydokey` is at `devopswithkubernetes/okeydokey` (`https://hub.docker.com/r/devopswithkubernetes/okeydokey/)`.

Connecting Travis CI with a GitHub repository is quite simple; all we need to do is authorize Travis CI to access our GitHub repositories and enable it to build the repository in the settings page (`https://travis-ci.com/account/repositories`). Another thing we'll need is a GitHub access token or a deploy key that has write permission to our repository. This will be put on the Travis CI so that the CI script can update the built image back into the config branch. Please refer to the GitHub official documentation (`https://developer.github.com/v3/guides/managing-deploy-keys/#deploy-keys`) to obtain a deploy key.

The definition of a job in Travis CI is configured in a file, `.travis.yml`, placed under the same repository. The definition is a YAML format template consisting of blocks of shell scripts that tell us what Travis CI should do during a build.

The full Travis CI document can be found here: `https://docs.travis-ci.com/user/tutorial/`.

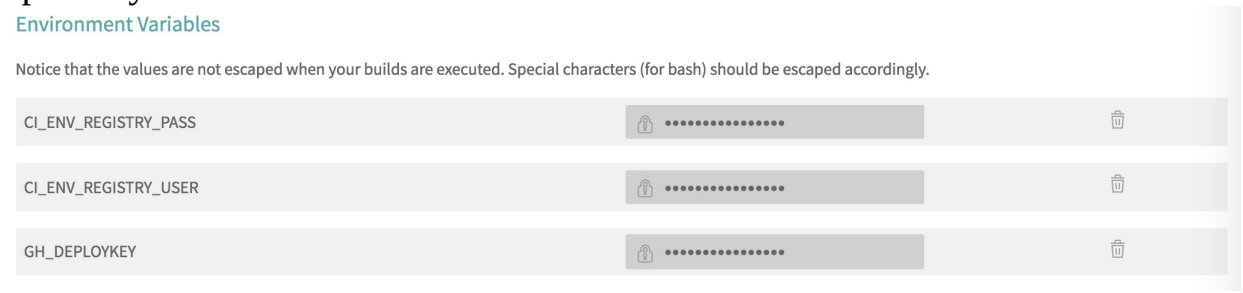
You can find explanations for the blocks of our `.travis.yml` file at the following URL: `https://github.com/DevOps-with-Kubernetes/okeydokey/blob/master/.travis.yml`.

env

This section defines environment variables that are visible throughout a build:
DOCKER_REPO=devopswithkubernetes/okeydokey
BUILD_IMAGE_PATH=${DOCKER_REPO}:build-${TRAVIS_COMMIT}
RELEASE_IMAGE_PATH=${DOCKER_REPO}:${TRAVIS_TAG}

Here, we set some variables that might be changed, such as the Docker registry path where the built image is heading. There's also metadata about a build passed from Travis CI in the form of environment variables, which is documented here: `https://docs.travis-ci.com/user/environment-variables/#default-environment-variables`. For example, `TRAVIS_COMMIT` represents the hash of the current commit, and we use it as an identifier to distinguish our images across builds.

The other source of environment variables is configured manually on Travis CI. Because the variables configured there would be hidden from public view, we stored some sensitive data such as credentials for Docker Hub and our GitHub repository there:



Every CI tool has its own best practices to deal with secrets. For instance, some CI tools also allow us to save variables in the CI server, but these are still printed in the building logs, so we're unlikely to save secrets there in such cases.

Key management systems such as Vault (`https://www.vaultproject.io/`) or similar services by cloud providers such as GCP KMS (`https://cloud.google.com/kms/`), AWS KMS (`https://aws.amazon.com/kms/`), and Azure Key Vault (`https://azure.microsoft.com/en-us/services/key-vault/`), are recommended for storing sensitive credentials.

docker build -t my-app .
docker run --rm --name app -dp 5000:5000 my-app
sleep 10
CODE=$(curl -IXGET -so /dev/null -w "%{http_code}" localhost:5000)
'[${CODE} -eq 200] && echo "Image is OK"'
docker stop app

As we're on Docker, the build only takes up one line of script. Our test is quite simple; it involves launching a container with the built image and making some requests to determine its integrity. We can do everything, including adding unit tests or running an automated integration test to improve the resultant artifacts, in this stage.

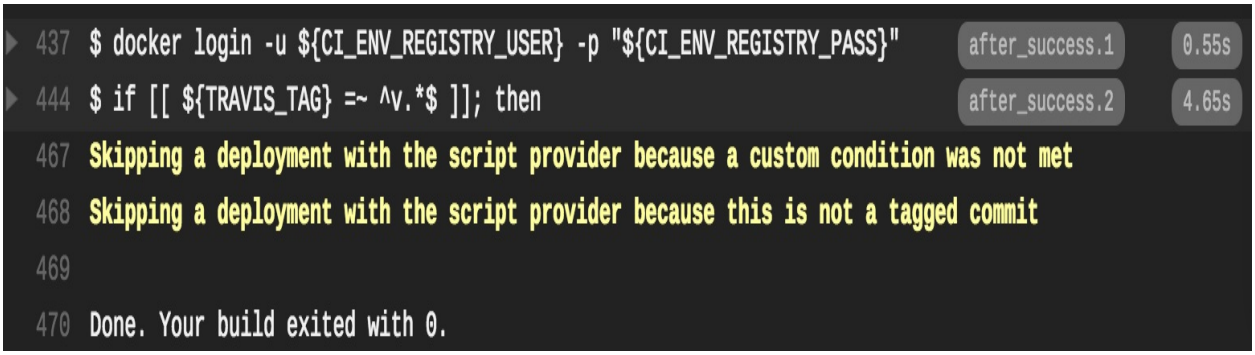
after_success

This block is executed only if the previous stage ends without any error. Once the block is executed, we are ready to publish our image: **docker login -u ${CI_ENV_REGISTRY_USER} -p "${CI_ENV_REGISTRY_PASS}" if [[${TRAVIS_TAG} =~ ^v.*$]]; then**
docker tag my-app ${RELEASE_IMAGE_PATH}
docker push ${RELEASE_IMAGE_PATH}
else
docker tag my-app ${BUILD_IMAGE_PATH}
docker push ${BUILD_IMAGE_PATH}
fi

Our image tag uses the commit hash for ordinary builds and uses a manually tagged version for releases. There's no absolute rule for tagging an image, but using the default `latest` tag for your business service is strongly discouraged as it could result in version confusion, such as running two different images that have the same name. The last conditional block is used to publish the image on certain branch tags, and we want to keep building and releasing on separate tracks. Remember to authenticate to Docker Hub before pushing an image.

Kubernetes decides whether the image should be pulled using `imagePullPolicy`, which defaults to `IfNotPresent`:
`IfNotPresent`: kubelet pulls images if they aren't present on the node. If the image tag is `:latest` and the policy isn't `Never`, then kubelet falls back to `Always`.
`Always`: kubelet always pulls images.
`Never`: kubelet never pulls images; it will find out whether the desired image is on the node or not.

Because we set our project deployments to actual machines only on a release, a build may stop and be returned at that moment. Let's have a look into the log of this build: `https://travis-ci.com/DevOps-with-Kubernetes/okeydokey/builds/93296022`. The log retains the executed scripts and outputs from every line of the script during a CI build:



As we can see, our build is successful, so the image is then published here: `https://hub.docker.com/r/devopswithkubernetes/okeydokey/tags/`. The build refers to the `build-842eb66b2fa612598add8e19769af5c56b922532` tag and we can now run it outside the CI server: **$ docker run --name test -dp 5000:5000 devopswithkubernetes/okeydokey:build-842eb66b2fa612598add8e19769af5c56b922532**
3d93d6505e369286c3f072ef4f04e15db2638f280c4615be95bff47379a70388
$ curl localhost:5000
OK

deploy

Although we can achieve a fully automated pipeline from end to end, we often encounter situations that hold up the deployment of a new build due to business concerns. Consequently, we tell Travis CI to run deployment scripts only when we want to release a new version.

As we stated earlier, the deployment in this example on Travis CI is merely to write the built image back to the template to be deployed. Here, we utilize the script provider to make Travis CI run our deployment script (`deployment/update-config.sh`) and the script does the following:

- Locates the config repository and corresponding branch
- Updates the image tag
- Commits the updated template back

After the updated image tag is committed into the repository, the job on Travis CI is done.

The other end of the pipeline is our agent inside the cluster. It is responsible for the following tasks:

- Periodically monitoring the change of our configs on GitHub
- Pulling and applying the updated image to our pod controller

The former is quite simple, but for the latter, we have to grant the agent sufficient permissions so that it can manipulate resources inside the cluster. Our example uses a service account, `cd-agent`, under a dedicated namespace, `cd`, to create and update our deployments, and the related RBAC configurations can be found under `chapter9/9-2_service-account-for-ci-tool/cd-agent` in the repository for this book.

Here, we grant the service account the permission to read and modify resources across namespaces, including the secrets of the whole cluster. Due to security concerns, it's always encouraged to restrict the permissions of a service account to the resources that the account actually uses, or it could be a potential vulnerability.

The agent itself is merely a long-running script at chapter9/9-2_service-account-for-ci-tool/utils/watcher/watcher.sh. To carry out updates, it uses `apply` and `rollout`:

```
...
apply_to_kube() {
  kubectl apply -f <template_path> -n <namespace>
  kubectl rollout status -f <template_path> -n <namespace> --timeout 5m
}
...
```

Let's deploy `agent` and its related `config` before rolling out our application:

```
$ kubectl apply -f chapter9/9-2_service-account-for-ci-tool/cd-agent
clusterrole.rbac.authorization.k8s.io/cd-role created
clusterrolebinding.rbac.authorization.k8s.io/cd-agent created
namespace/cd created
serviceaccount/cd-agent created
deployment.apps/state-watcher created
```

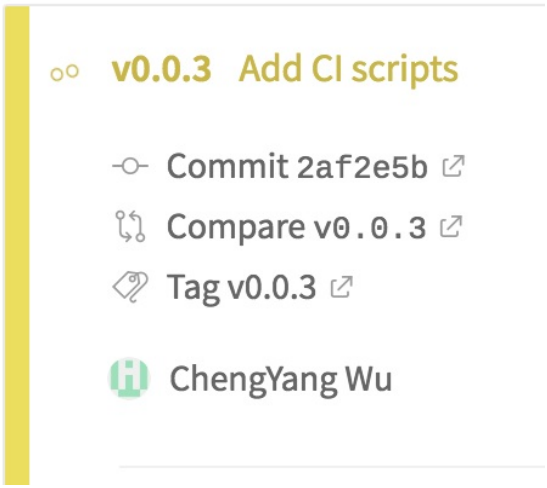
The `state-watcher` deployment is our `agent`, and it has been configured to monitor our config repository for environment variables:

```
$ cat chapter9/9-2_service-account-for-ci-tool/cd-agent/watcher-okeydokey.yml
...
        env:
        - name: WORK_PATH
          value: /repo
        - name: TEMPLATE_PATH
          value: /repo/deployment
        - name: REMOTE_GIT_REPO
          value: https://github.com/DevOps-with-Kubernetes/okeydokey.git
        - name: WATCH_BRANCH
          value: config
        - name: RELEASE_TARGET_NAMESPACE
          value: default
        - name: RELEASE_TARGET_CONTROLLER_TEMPLATE
          value: deployment.yml
...
```

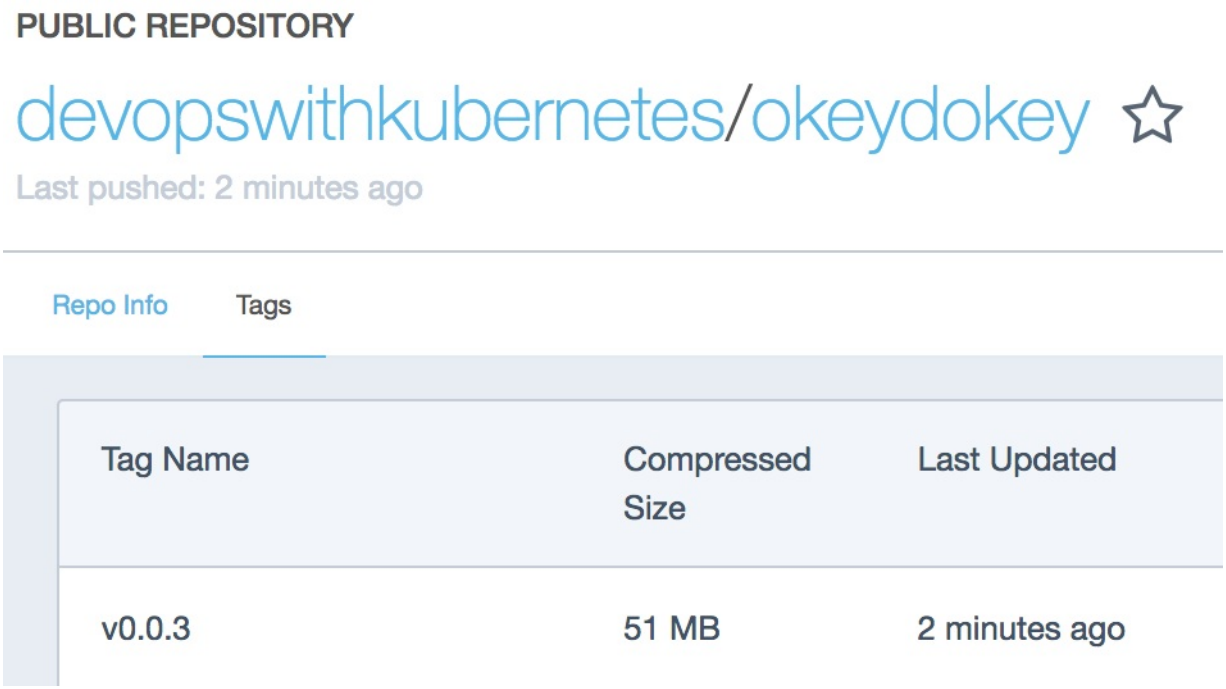
Everything is ready. Let's see the entire flow in action.

We publish a release with a `v0.0.3` tag at GitHub (https://github.com/DevOps-with-Kubernetes/okeydokey/releases/tag/v0.0.3):

Travis CI starts to build our job right after being triggered by the new tag:



If it fails, we can check the build log to see what went wrong: `https://travis-ci.com/DevOps-with-Kubernetes/okeydokey/jobs/162862675`. Fortunately, we get a green flag, so the built image will be pushed onto Docker Hub after a while:



At this moment, our `agent` should also notice the change in `config` and act upon it:

```
$ kubectl logs -f -n cd state-watcher-69bfdd578-8nn9s -f
...
From https://github.com/DevOps-with-Kubernetes/okeydokey
 * branch config -> FETCH_HEAD
 * [new branch] config -> origin/config
Tue Dec 4 23:44:56 UTC 2018: No update detected.
deployment.apps/okeydokey created
service/okeydokey-svc created
Waiting for deployment "okeydokey" rollout to finish: 0 of 2 updated replicas are availa
Waiting for deployment "okeydokey" rollout to finish: 0 of 2 updated replicas are availa
Waiting for deployment "okeydokey" rollout to finish: 1 of 2 updated replicas are availa
deployment "okeydokey" successfully rolled out
...
```

As we can see, our application has rolled out successfully, and it should start to welcome everyone with `OK`:

```
$ kubectl proxy &
$ curl localhost:8001/api/v1/namespaces/default/services/okeydokey-svc:80/proxy/
OK
```

The pipeline we built and demonstrated in this section is a classic flow to deliver code continuously in Kubernetes. However, as the work style and culture varies from team to team, designing a tailor-made continuous delivery pipeline for your team can improve efficiency. For example, the built-in update strategy of a

deployment is the rolling update. Teams that prefer other types of deployment strategies such as blue/green or canary have to change the pipeline to fit their needs. Fortunately, Kubernetes is extremely flexible, and we can implement various strategies by compositing `Deployment`, `Service`, `Ingress`, and so on.

In the previous edition of this book, we demonstrated a similar flow but applied the configuration from the CI server directly. Both approaches have their pros and cons. If you don't have any security concerns related to putting cluster information on the CI server and just need a really easy CD flow, then the push-based pipeline is still an option. You can find a script for exporting the tokens of a service account and another script for applying configuration to Kubernetes here: `https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter9/9-2_service-account-for-ci-tool/utils/push-cd`.

Gaining a deeper understanding of pods

Although birth and death are merely a blink during a pod's lifetime, they're also the most fragile points of a service. We want to avoid common situations such as routing requests to an unready box or brutally cutting all in-flight connections to a terminating machine. As a consequence, even if Kubernetes takes care of most things for us, we should know how to configure our service properly to make sure every feature is delivered perfectly.

Starting a pod

By default, Kubernetes moves a pod's state to Running as soon as a pod launches. If the pod is behind a service, the endpoint controller registers an endpoint to Kubernetes immediately. Later on, `kube-proxy` observes the change of endpoints and configures the host's `ipvs` or `iptables` accordingly. Requests from the outside world now go to pods. These operations happen very quickly, so it's quite possible that requests arrive at a pod before the application is ready, especially with bulky software. If a pod fails while running, we should remove it from the pool of a service instantly to make sure no requests reach a bad endpoint.

The `minReadySeconds` field of deployment and other controllers doesn't postpone a pod from becoming ready. Instead, it delays a pod from becoming available. A rollout is only successful if all pods are available.

Liveness and readiness probes

A probe is an indicator of a container's health. It judges health through periodically performing diagnostic actions against a container via kubelet. There are two kinds of probes for determining the state of a container:

- **Liveness probe**: This indicates whether or not a container is alive. If a container fails on this probe, kubelet kills it and may restart it based on the `restartPolicy` of a pod.
- **Readiness probe**: This indicates whether a container is ready for incoming traffic. If a pod behind a service isn't ready, its endpoint won't be created until the pod is ready.

The `restartPolicy` tells us how Kubernetes treats a pod on failures or terminations. It has three modes: `Always`, `OnFailure`, or `Never`. The default is set to `Always`.

Three kinds of action handlers can be configured to diagnose a container:

- `exec`: This executes a defined command inside the container. It's considered to be successful if the exit code is `0`.
- `tcpSocket`: This tests a given port via TCP and is successful if the port is opened.

- `httpGet`: This performs `HTTP GET` on the IP address of the target container. Headers in the request to be sent are customizable. This check is considered to be healthy if the status code satisfies `400 > CODE >= 200`.

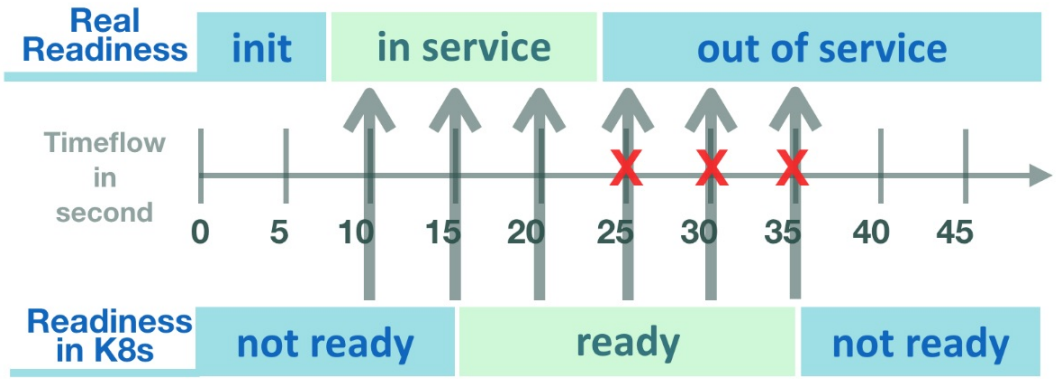
Additionally, there are five parameters that define a probe's behavior:

- `initialDelaySeconds`: How long kubelet should wait for before the first probing
- `successThreshold`: A container is considered to be healthy only if it got consecutive times of probing successes over this threshold
- `failureThreshold`: The same as the previous one, but defines the negative side instead
- `timeoutSeconds`: The time limitation of a single probe action
- `periodSeconds`: Intervals between probe actions

The following code snippet demonstrates the use of a readiness probe. The full template can be found at https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/blob/master/chapter9/9-3_on_pods/probe.yml: **...**

```
containers:
- name: main
image: devopswithkubernetes/okeydokey:v0.0.4
readinessProbe:
httpGet:
path: /
port: 5000
periodSeconds: 5
initialDelaySeconds: 10
successThreshold: 2
failureThreshold: 3
timeoutSeconds: 1
command:
...
```

In this example, we used some tricks with our main application, which set the starting time of the application to around six seconds and replace the application after 20 seconds with another one that echoes `HTTP 500`. The application's interaction with the readiness probe is illustrated in the following diagram:



The upper timeline is a pod's real readiness, and the other one is its readiness from Kubernetes' perspective. The first probe executes 10 seconds after the pod is created, and the pod is regarded as ready after two probing successes. A few seconds later, the pod goes out of service due to the termination of our application, and it becomes unready after the next three failures. Try to deploy

the preceding example and observe its output: **$ kubectl logs -f my-app-6759578c94-kkc5k**
1544137145.530593922 - [sys] pod is created.
1544137151.658855438 - [app] starting server.
1544137155.164726019 - [app] GET / HTTP/1.1
1544137160.165020704 - [app] GET / HTTP/1.1
1544137161.129309654 - [app] GET /from-tester
1544137165.141985178 - [app] GET /from-tester
1544137165.165597677 - [app] GET / HTTP/1.1
1544137168.533407211 - [app] stopping server.
1544137170.169371453 - [500] readiness test fail#1
1544137175.180640604 - [500] readiness test fail#2
1544137180.171766986 - [500] readiness test fail#3
...

In our example file, there is another pod, `tester`, which is constantly making requests to our service and the log entries. `/from-tester` in our service represents the requests from the tester. From the tester's activity logs, we can observe that the traffic from `tester` is stopped after our service becomes unready (notice the activities of two pods around the time `1544137180`): **$ kubectl logs tester**
1544137141.107777059 - timed out
1544137147.116839441 - timed out
1544137154.078540367 - timed out
1544137160.094933434 - OK
1544137165.136757412 - OK
1544137169.155453804 -
1544137173.161426446 - HTTP/1.1 500
1544137177.167556193 - HTTP/1.1 500
1544137181.173484008 - timed out
1544137187.189133495 - timed out
1544137193.198797682 - timed out
...

Since we didn't configure the liveness probe in our service, the unhealthy container won't be restarted unless we kill it manually. In general, we would use both probes together to automate the healing process.

Custom readiness gate

The testing targets of the readiness probe are always containers, which means that it can't be used to disable a pod from a service by using external states. Since a service selects pods by their labels, we can control the traffic to pods by manipulating pod labels to a certain extent. However, pod labels are also read by other components inside Kubernetes, so building complex toggles with labels could lead to unexpected results.

The pod readiness gate is the feature that allows us to mark whether a pod is ready or not, based on the conditions we defined. With the pod readiness gates defined, a pod is regarded as ready only if its readiness probe passes and the status of all readiness gates associated with the pod is `True`. We can define the readiness gate as shown in the following snippet: **...**
spec:
readinessGates:
- conditionType: <value>
containers:
- name: main
...

The value must follow the format of a label key such as `feature_1` or `myorg.com/fg-2`.

When a pod starts, a condition type we defined will be populated as a condition under a pod's `.status.conditions[]` path, and we have to explicitly set the condition to `True` to mark a pod ready. As for Kubernetes 1.13, the only way to edit the condition is with the `patch` API. Let's see an example at https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/blob/master/chapter9/9-3_on_pods/readiness_gates.yml: **$ cat chapter9/9-3_on_pods/readiness_gates.yml | grep readinessGates -C 1**
spec:
readinessGates:
- conditionType: "MY-GATE-1"

$ kubectl apply -f chapter9/9-3_on_pods/readiness_gates.yml
deployment.apps/my-2nd-app created

service/my-2nd-app-svc created

$ kubectl get pod -o custom-columns=NAME:.metadata.name,IP:.status.podIP
NAME IP
my-2nd-app-78786c6d5d-t4564 172.17.0.2

$ kubectl logs my-2nd-app-78786c6d5d-t4564
1544216932.875020742 - [app] starting server.

$ kubectl describe ep my-2nd-app-svc
Name: my-2nd-app-svc
Namespace: default
Labels: app=my-2nd-app
Annotations: <none>
Subsets:
Addresses: <none>
NotReadyAddresses: 172.17.0.2
...

Here, our custom condition is called `MY-GATE-1` and the application is the one that we have used throughout this chapter. As we can see, even if the pod has started, its address is still listed in `NotReadyAddresses`. This means that the pod isn't taking any traffic. We can verify its status with `describe` (or `wide`/`json`/`yaml`): **$ kubectl describe pod my-2nd-app-78786c6d5d-t4564**
...
Readiness Gates:
Type Status
MY-GATE-1 <none>
Conditions:
Type Status
Initialized True
Ready False
ContainersReady True
PodScheduled True
...

The container in the pod is ready, but the pod itself isn't ready due to the readiness gates. To toggle it on, we'll need to make a request to the API server with a **JSON Patch** payload to `/api/v1/namespaces/<namespace>/pods/<pod_name>/status`:

```
$ kubectl proxy &
$ curl http://localhost:8001/api/v1/namespaces/default/pods/my-2nd-app-78786c6d5d-t4564/
-XPATCH -H "Content-Type: application/json-patch+json" -d \
'[{"op":"add","path":"/status/conditions/-","value":{"type":"MY-GATE-1", "status": "True
...
  "status": {
    "phase": "Running",
    "conditions": [
...
      {
        "type": "MY-GATE-1",
        "status": "True",
        "lastProbeTime": null,
        "lastTransitionTime": null
      }
...
```

We'll see that an entry will be inserted into the `.status.conditions` list. Now, if we check the endpoints of the service, we can see that the pod has started to serve requests: **$ kubectl describe ep my-2nd-app-svc**
Name: my-2nd-app-svc
Namespace: default
Labels: app=my-2nd-app
Annotations: <none>
Subsets:
Addresses: 172.17.0.2
...
also the status of the gates:
$ kubectl describe pod my-2nd-app-78786c6d5d-t4564 | grep -A2 Readiness
Readiness Gates:
Type Status
MY-GATE-1 True

To put the pod in the other way around, we could use the `replace` or `remove` operation of JSON Patch to set the condition's status to `False` or `<none>`: **## we need the index of our gate, and we use jq to query it here:**
$ export RG_NAME="MY-GATE-1"
$ kubectl get pod my-2nd-app-78786c6d5d-t4564 -o json | jq --arg R "$RG_NAME" 'foreach .status.conditions[] as $e (-1; .+1; select($e.type == $R))'

```
0
$ kubectl proxy &

## fill the queried "0" to the path parameter
$ curl http://localhost:8001/api/v1/namespaces/default/pods/my-2nd-app-78786c6d5d-t4564/status \
-XPATCH -H "Content-Type: application/json-patch+json" -d \
'[{"op":"replace","path":"/status/conditions/0","value":{"type":"MY-GATE-1", "status": "False"}}]'
...
$ kubectl describe ep my-2nd-app-svc | grep -B2 NotReadyAddresses
Subsets:
Addresses: <none>
NotReadyAddresses: 172.17.0.2
```

The pod now becomes unready again. With the readiness gate, we can nicely separate the logic of toggling business features and managing labels.

...
spec:
 containers:
 - name: my-app
 image: <my-app>
 initContainers:
 - name: init-my-app
 image: <init-my-app>
...

They only differ in the following respects:

- `init` containers don't have readiness probes as they run to completion.
- The port defined in `init` containers won't be captured by the service in front of the pod.

- The request limit of resources are calculated with `max(sum(regular containers)`, and `max(init containers))`, which means if one of the `init` containers sets a higher resource limit than other `init` containers, as well as the sum of the resource limits of all regular containers, Kubernetes schedules the pod according to the `init` container's resource limit.

The usefulness of `init` containers is more than blocking the application containers. For instance, we can utilize an `init` container to configure an image by sharing an `emptyDir` volume with `init` containers and application containers, instead of building another image that only runs `awk`/`sed` on the base image. Also, it grants us the flexibility to use different images for initialization tasks and the main application.

Terminating a pod

The sequence of shutdown events is similar to events while starting a pod. After receiving a deletion invocation, Kubernetes sends `SIGTERM` to the pod that is going to be deleted, and the pod's state becomes terminating. Meanwhile, Kubernetes removes the endpoint of that pod to stop further requests if the pod is backing a service. Occasionally, there are pods that don't quit at all. It could be that the pods don't honor `SIGTERM`, or simply because their tasks aren't completed. Under such circumstances, Kubernetes will send `SIGKILL` to forcibly kill those pods after the termination period. The period length is set at `.spec.terminationGracePeriodSeconds` under the pod specification. Even though Kubernetes has mechanisms to reclaim such pods anyway, we still should make sure our pods can be closed properly.

Handling SIGTERM

Graceful termination isn't a new idea; it is a common practice in programming. Killing a pod forcibly while it's still working is like suddenly unplugging the power cord of a running computer, which could harm the data.

The implementation principally includes three steps:

1. Register a handler to capture termination signals.
2. Do everything required in the handler, such as freeing resources, writing data to external persistent layers, releasing distribution locks, or closing connections.

3. Perform a program shutdown. Our previous example demonstrates the idea: closing the controller thread on `SIGTERM` in the `graceful_exit_handler` handler. The code can be found here: https://github.com/DevOps-with-Kubernetes/okeydokey/blob/master/app.py.

Due to the fact that Kubernetes can only send signals to the `PID 1` process in a container, there are some common pitfalls that could fail the graceful handler in our program.

SIGTERM isn't sent to the application process

In `Chapter 2`, *DevOps with Containers*, we learned there are two forms to invoke our program when writing a Dockerfile: the shell form and the exec form. The shell to run the shell form commands defaults to `/bin/sh -c` on Linux containers. Hence, there are a few questions related to whether `SIGTERM` can be received by our applications:

- How is our application invoked?
- What shell implementation is used in the image?
- How does the shell implementation deal with the `-c` parameter?

Let's approach these questions one by one. The Dockerfile used in the following example can be found here: `https://github.com/PacktPublishing/DevOps-with-Kubernetes-Second-Edition/tree/master/chapter9/9-3_on_pods/graceful_docker`.

Say we're using the shell form command, `CMD python -u app.py`, in our Dockerfile to execute our application. The starting command of the container would be `/bin/sh -c "python3 -u app.py"`. When the container starts, the structure of the processes inside it is as follows: **# the image is from "graceful_docker/Dockerfile.shell-sh"**
**$ kubectl run --generator=run-pod/v1 **
--image=devopswithkubernetes/ch93:shell-sh my-app
pod/my-app created
$ kubectl exec my-app ps ax
PID TTY STAT TIME COMMAND
1 ? Ss 0:00 /bin/sh -c python3 -u app.py
6 ? S 0:00 python3 -u app.py
7 ? Rs 0:00 ps ax

We can see that the `PID 1` process isn't our application with handlers; it's the shell instead. When we try to kill the pod, `SIGTERM` will be sent to the shell rather than to our application, and the pod will be terminated after the grace period expires. We can check the log in our application when deleting it to see whether it received

`SIGTERM`: **$ kubectl delete pod my-app &**
pod "my-app" deleted
$ kubectl logs -f my-app
$ 1544368565.736720800 - [app] starting server.
rpc error: code = Unknown desc = Error: No such container: 2f007593553cfb700b0aece1f8b6045b4096b2f50f97a42e684a98e502af29ed

Our application exited without going to the stop handler in the code. There are a couple of ways to properly promote our application to `PID 1`. For example, we can explicitly call `exec` in the shell form, such as `CMD exec python3 -u app.py`, so that our program will inherit `PID 1`. Or, we can choose the `exec` form, `CMD [ "python3", "-u", "app.py" ]`, to execute our program directly: **## shell form with exec**
**$ kubectl run --generator=run-pod/v1 **
--image=devopswithkubernetes/ch93:shell-exec my-app-shell-exec
pod/my-app-shell-exec created
$ kubectl exec my-app-exec ps ax
PID TTY STAT TIME COMMAND
1 ? Ss 0:00 python3 -u app.py
5 ? Rs 0:00 ps ax
delete the pod in another terminal
$ kubectl logs -f my-app-shell-exec
1544368913.313778162 - [app] starting server.
1544369448.991261721 - [app] stopping server.
rpc error: code = Unknown desc =...

exec form
**$ kubectl run --generator=run-pod/v1 **
--image=devopswithkubernetes/ch93:exec-sh my-app-exec
pod/my-app-exec created
PID TTY STAT TIME COMMAND
1 ? Ss 0:00 python3 -u app.py
5 ? Rs 0:00 ps ax
$ kubectl logs -f my-app-exec
1544368942.935727358 - [app] starting server.
1544369503.846865654 - [app] stopping server.
rpc error: code = Unknown desc =...

The program, executed in either way, can now receive `SIGTERM` properly. Besides, if we need to set up the environment with a shell script for our program, we should either trap signals in the script to propagate them to our program, or use the exec call to invoke our program so that the handler in our application is able to work as desired.

The second and the third questions are about the shell implication: how could it affect our graceful handler? Again, the default command of a Docker container in Linux is `/bin/sh -c`. As `sh` differs among popular Docker images, the way it handles `-c` could also affect the signals if we're using the shell form. For example, Alpine Linux links `ash` to `/bin/sh`, and the Debian family of distributions use `dash`. Before Alpine 3.8 (or BusyBox 1.28.0), `ash` forks a new process when using `sh -c`, and it uses `exec` in 3.8. We can observe the difference with `ps`, where we can see the one in 3.7 gets `PID 6` while it's `PID 1` in 3.8: **$ docker run alpine:3.7 /bin/sh -c "ps ax"**
PID USER TIME COMMAND
1 root 0:00 /bin/sh -c ps ax
6 root 0:00 ps ax
$ docker run alpine:3.8 /bin/sh -c "ps ax"
PID USER TIME COMMAND
1 root 0:00 ps ax

How do `dash` and `bash` handle these cases? Let's take a look: **## there is no ps inside the official debian image, Here we reuse the one from above, which is also based on the debian:**
$ docker run devopswithkubernetes/ch93:exec-sh /bin/sh -c "ps ax"
PID TTY STAT TIME COMMAND
1 ? Ss 0:00 /bin/sh -c ps ax
6 ? R 0:00 ps ax
$ docker run devopswithkubernetes/ch93:exec-sh /bin/bash -c "ps ax"
PID TTY STAT TIME COMMAND
1 ? Rs 0:00 ps ax

As we can see, their results are different as well. Our application can now respond to the terminating event appropriately. There is one more thing, however, that could potentially harm our system if our application is run as `PID 1` and it uses more than one process inside the container.

On Linux, a child process becomes a zombie if its parent doesn't wait for its execution. If the parent dies before its child process ends, the `init` process should adopt those orphaned processes and reap processes that become zombies. System programs know how to deal with orphaned processes, so zombie processes are not a problem most of the time. However, in a containerized context, the process that holds `PID 1` is our application, and the operating system would expect our application to reap zombie processes. Because our application isn't designed to act as a proper `init` process, however, handling the state of child processes is unrealistic. If we just ignore it, at worst the process table of the node will be filled with zombie processes and we won't be able to launch new programs on the node anymore. In Kubernetes, if a pod with zombie processes is gone, then all zombie processes inside will be cleaned. Another possible scenario is if our application performs some tasks frequently through scripts in the background, which could potentially fork lots of processes. Let's consider the following simple example:

```
$ kubectl run --generator=run-pod/v1 \
--image=devopswithkubernetes/ch93:exec-sh my-app-exec
pod/my-app-exec created
## let's enter our app pod and run sleep in background inside it
$ kubectl exec -it my-app-exec /bin/sh
# ps axf
  PID TTY      STAT   TIME COMMAND
    5 pts/0    Ss     0:00 /bin/sh
   10 pts/0    R+     0:00  \_ ps axf
    1 ?        Ss     0:00 python3 -u app.py
# sleep 30 &
# ps axf
  PID TTY      STAT   TIME COMMAND
    5 pts/0    Ss     0:00 /bin/sh
   11 pts/0    S      0:00  \_ sleep 30
   12 pts/0    R+     0:00  \_ ps axf
    1 ?        Ss     0:00 python3 -u app.py

## now quit kubectl exec, wait 30 seconds, and check the pod again
$ kubectl exec my-app-exec ps axf
  PID TTY      STAT   TIME COMMAND
   23 ?        Rs     0:00 ps axf
    1 ?        Ss     0:00 python3 -u app.py
   11 ?        Z      0:00 [sleep] <defunct>
```

`sleep 30` is now a zombie in our pod. In `Chapter 2`, *DevOps with Containers*, we mentioned that the `docker run --init` parameter can set a simple `init` process for our container. In Kubernetes, we can make the `pause` container, a special container that deals with those chores silently for us, be present in our pod by specifying `.spec.shareProcessNamespace` in the pod specification: **$ kubectl apply -f chapter9/9-3_on_pods/sharepidns.yml**

pod/my-app-with-pause created
$ kubectl exec my-app-with-pause ps ax
1 ? Ss 0:00 /pause
6 ? Ss 0:00 python3 -u app.py
10 ? Rs 0:00 ps ax

The `pause` process ensures that zombies are reaped and `SIGTERM` goes to our application process. Notice that by enabling process namespace sharing, aside from our application no longer having `PID 1`, there are two other key differences:

- All containers in the same pod share process information with each other, which means a container can send signals to another container
- The filesystem of containers can be accessed via the `/proc/$PID/root` path

If the described behaviors aren't feasible to your application while an `init` process is still needed, you can opt for Tini (`https://github.com/krallin/tini`), or dump-init (`https://github.com/Yelp/dumb-init`), or even write a wrapper script to resolve the zombie reaping problem.

SIGTERM doesn't invoke the termination handler

In some cases, the termination handler of a process isn't triggered by `SIGTERM`. For instance, sending `SIGTERM` to `nginx` actually causes a fast shutdown. To gracefully close an `nginx` controller, we have to send `SIGQUIT` with `nginx -s quit` instead.

The full list of supported actions on the signal of `nginx` is listed here: `http://nginx.org/en/docs/control.html`.

Now, another problem arises: how do we send signals other than `SIGTERM` to a container when deleting a pod? We can modify the behavior of our program to trap `SIGTERM`, but there's nothing we can do about popular tools such as `nginx`. For such a situation, we can use life cycle hooks.

Container life cycle hooks

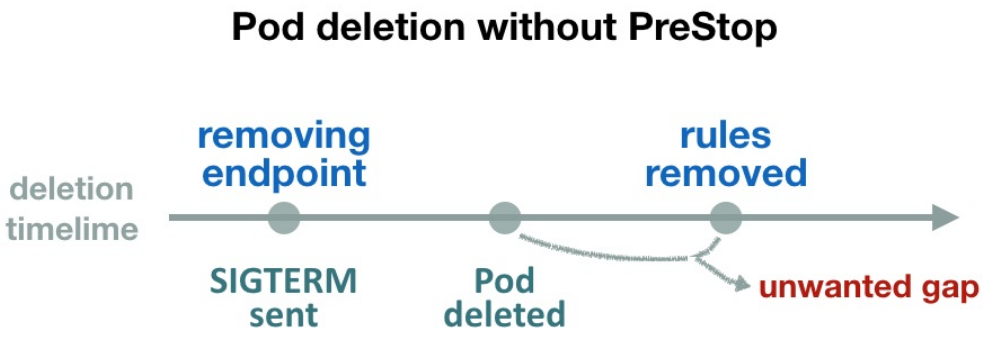
Life cycle hooks are actions triggered on certain events and performed against containers. They work like a single Kubernetes probing action, but they'll be fired at least once per event during a container's lifetime. Currently, two events are supported:

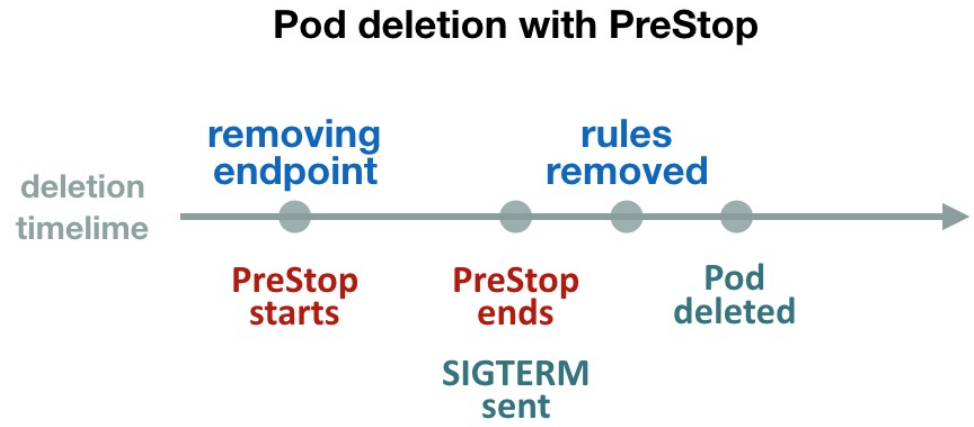
- `PostStart`: This executes right after a container is created. Since this hook and the entry point of a container are fired asynchronously, there's no guarantee that the hook will be executed before the container starts. As such, we're unlikely to use it to initialize resources for a container.
- `PreStop`: This executes right before sending `SIGTERM` to a container. One difference from the `PostStart` hook is that the `PreStop` hook is a synchronous call; in other words, `SIGTERM` is only sent after a `PreStop` hook exited.

We can easily solve our `nginx` shutdown problem with a `PreStop` hook: **...**
containers:
- name: main
image: nginx
life cycle:
preStop:
exec:
command: ["nginx", "-s", "quit"]
...

An important property of hooks is they can affect the state of a pod in certain ways: a pod won't be running unless its `PostStart` hook exits successfully. A pod is set to terminate immediately on deletion, but `SIGTERM` won't be sent unless the `PreStop` hook exits successfully. Therefore, we can resolve a situation that a pod quits before its proxy rules are removed on the node by the `PreStop` hook.

The following diagram illustrates how to use the hook to eliminate the unwanted





gap:

The implementation is to just add a hook that sleeps for a few seconds: **...**
containers:
- name: main
image: my-app
life cycle:
preStop:
exec:
command: ["/bin/sh", "-c", "sleep 5"]
...

Tackling pod disruptions

Ideally, we'd like to keep the availability of our service as high as we can. However, there're always lots of events that cause the pods that are backing our service to go up and down, either voluntarily or involuntarily. Voluntary disruptions include `Deployment` rollouts, planned node maintenance, or the accidental killing of a pod with the API. On the whole, every operation that goes through the Kubernetes master counts. On the other hand, any unexpected outage that leads to the termination of our service belongs to the category of involuntary disruptions.

In previous chapters, we discussed how to prevent involuntary disruptions by replicating pods with `Deployment` and `StatefulSet`, appropriately configuring resource requests and limits, scaling an application's capacity with the autoscaler, and distributing pods to multiple locations with affinities and anti-affinities. Since we've already put a lot of effort into our service, what could go wrong when it comes to these expected voluntary disruptions? In fact, because they're events that are likely to happen, we ought to pay more attention to them.

In `Deployment` and other similar objects, we can use the `maxUnavailable` and `maxSurge` fields that help us roll out our updates in a controlled manner. As for other cases, such as node maintenance tasks performed by cluster administrators who don't have domain knowledge about all the applications run in the cluster, the service owner can utilize `PodDisruptionBudget` to tell Kubernetes how many pods are required for a service to meet its service level.

A pod disruption budget has the following syntax: **apiVersion: policy/v1beta1**
kind: PodDisruptionBudget
metadata:
name: <pdb name>
spec:
maxUnavailable: <desired number or percentage of pods>
minAvailable: <desired number or percentage of pods>
selector:
<matchLabels> or <matchExpressions>

There are two configurable fields in a pod disruption budget, but they can't be used together. The selector is identical to the one in `Deployment` or other places. Note that a pod disruption budget is immutable, which means it can't be updated after its creation. The `minAvailable` and `maxUnavailable` fields are mutually exclusive, but they're the same in some ways. For example, `maxUnavailable:0` means zero tolerance of any pod losses, and it's roughly equivalent to `minAvailable:100%`, which means that all pods should be available.

Pod disruption budgets work by evicting events such as draining nodes or pod preemption. They don't interfere with the rolling update process performed by controllers such as `Deployment` or `StatefulSet`. Suppose that we want to temporarily remove one node from the cluster with `kubectl drain`, but this would violate certain pod disruption budgets of running applications. In this case, the draining operation would be blocked unless all pod disruption budgets can be satisfied. However, if the Kubernetes scheduler is going to evict a victim pod to fulfill high priority pods, the scheduler would only try to meet all of the pod disruption budgets if possible. If the scheduler can't find a target without breaking any pod disruption budgets, it would still pick a pod with the lowest priority.

Summary

In this chapter, we've discussed topics related to building a continuous delivery pipeline and how to strengthen our deployment tasks. The rolling update of a pod is a powerful tool that allows us to perform updates in a controlled fashion. To trigger a rolling update, what we need to do is change the pod's specification in a controller that supports that rolling update. Additionally, although the update is managed by Kubernetes, we can still control it with `kubectl rollout` to a certain extent.

Later on, we fabricated an extensible continuous delivery pipeline using `GitHub`/`DockerHub`/`Travis-CI`. We then moved on to learn more about the life cycle of pods to prevent any possible failures, including using the readiness and liveness probes to protect a pod; initializing a pod with `init` containers; handling `SIGTERM` properly by picking the right composition of invocation commands of the entry point of our program and the shell to run it; using life cycle hooks to stall a pod's readiness, as well as its termination for the pod to be removed from a service at the right time; and assigning pod disruption budgets to ensure the availability of our pods.

In `Chapter 10`, *Kubernetes on AWS*, we'll move on to learn the essentials of how to deploy the cluster on AWS, the major player among all public cloud providers.

Kubernetes on AWS

Using Kubernetes on the public cloud is flexible and scalable for your application. AWS is one of the most popular services in the public cloud industry. In this chapter, you'll learn what AWS is and how to set up Kubernetes on AWS along with the following topics:

- Understanding the public cloud
- Using and understanding AWS components
- Using Amazon EKS to set up a Kubernetes cluster on AWS
- Using EKS to manage Kubernetes

Introduction to AWS

When you run your application on the public network, you need an infrastructure such as networks, Virtual Machines (VMs), and storage. Obviously, companies borrow or build their own data center to prepare those infrastructures, and then hire data center engineers and operators to monitor and manage those resources.

However, purchasing and maintaining those assets requires a large capital expense as well as an operational expense for data center engineers/operators. You also need a lead time to fully set up those infrastructures, such as buying a server, mounting to a data center rack, cabling a network, and then the initial configuration/installation of the OS and so on.

Consequently, rapidly allocating an infrastructure with appropriate resource capacity is one of the important factors that dictates the success of your business.

To make infrastructure management easier and quicker, there's a lot that technology can do to help data centers, for example, for virtualization, **Software Defined Network** (**SDN**) and **Storage Area Network** (**SAN**). But combining this technology has some sensitive compatibility issues and is difficult to stabilize; therefore it's necessary to hire experts in this industry, which makes operation costs higher eventually.

Public cloud

There are some companies that have provided an online infrastructure service. AWS is a well known service that provides online infrastructure, which is called cloud or **public cloud**. Back in the year 2006, AWS officially launched the virtual machine service, which was called **Elastic Computing Cloud** (**EC2**); an online object store service, which was called **Simple Storage Service** (**S3**); and an online messaging queue service, which was called **Simple Queue Service** (**SQS**).

These services are simple enough, but from a data center management point of view, they relieve infrastructure pre-allocation and reduce read time, because of pay-as-you-go pricing models (paying hourly or yearly for usage to AWS). Consequently, AWS is getting so popular that many companies have switched from their own data centers to the public cloud.

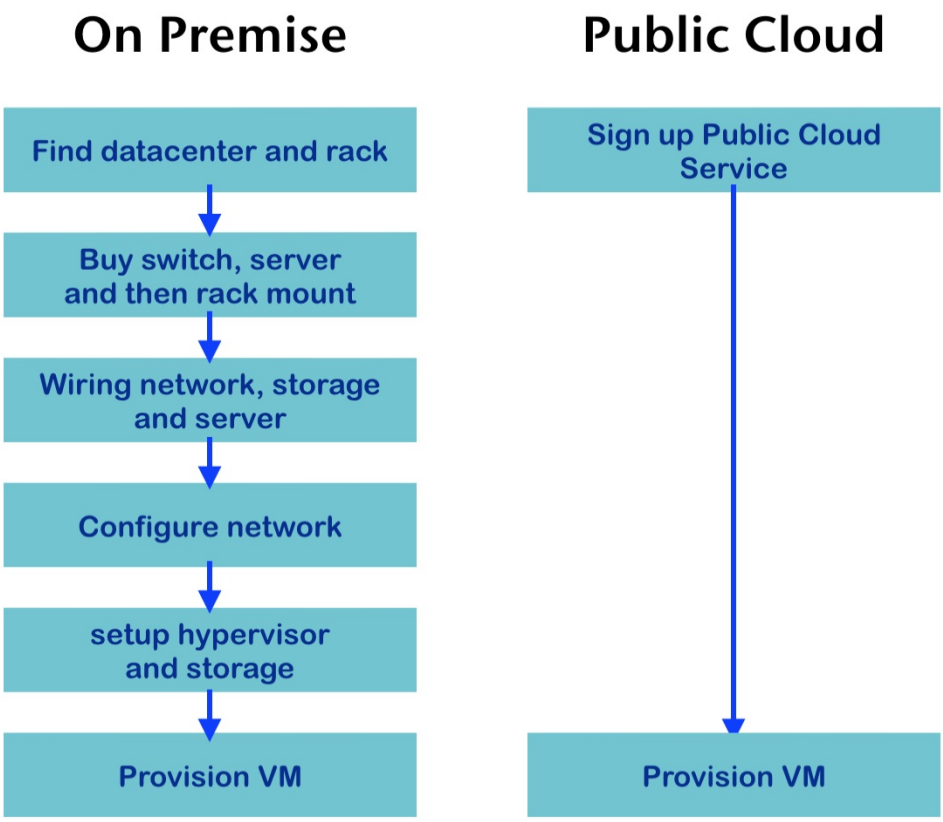
*An antonym of the public cloud, your own data center is referred to as **on-premises**.*

API and infrastructure as code

One of the unique benefits of using a public cloud instead of on-premises data centers is that public cloud provides an API to control infrastructure. AWS provides command-line tools (**AWS CLI**) to control AWS infrastructure. For example, after signing up to AWS (`https://aws.amazon.com/free/`), install AWS CLI (`http://docs.aws.amazon.com/cli/latest/userguide/installing.html`); then, if you want to launch one virtual machine (EC2 instance), use the AWS CLI as follows:

As you can see, it only takes a few minutes to access your virtual machine after signing up to AWS. On the other hand, what if you set up your own on-premises data center from scratch? The following diagram shows a comparison of what

happens if you use on-premises data centers or if you use the public cloud:



As you can see, the public cloud is very simple and quick; this is why it's flexible and convenient, not only for emerging, but also for permanent usage.

AWS components

AWS has some components to configure network and storage. These are important for understanding how the public cloud works as well as how to configure Kubernetes.

//specify CIDR block as 10.0.0.0/16
//the result, it returns VPC ID as "vpc-0ca37d4650963adbb"

$ aws ec2 create-vpc --cidr-block 10.0.0.0/16
{
 "Vpc": {
 "CidrBlock": "10.0.0.0/16",
 "DhcpOptionsId": "dopt-3d901958",
 "State": "pending",
 "VpcId": "vpc-0ca37d4650963adbb",
...

A subnet is a logical network block. This must belong to one VPC as well as one availability zone, for example, the `vpc-0ca37d4650963adbb` VPC and `us-east-1b`. Then, the network CIDR must be within the VPC's CIDR. For example, if the VPC CIDR is `10.0.0.0/16` (`10.0.0.0–10.0.255.255`), then one subnet CIDR could be `10.0.1.0/24` (`10.0.1.0–10.0.1.255`).

In the following example, we'll create two subnets on the `us-east-1a` availability zone and another two subnets on the `us-east-1b` availability zone. Therefore, a total of four subnets will be created on `vpc-0ca37d4650963adbb` according to the following steps:

1. Create the first subnet, `10.0.1.0/24`, on the `us-east-1a` availability zone:

```
$ aws ec2 create-subnet --vpc-id vpc-0ca37d4650963adbb --c
{
    "Subnet": {
        "AvailabilityZone": "us-east-1a",
        "AvailabilityZoneId": "use1-az6",
        "AvailableIpAddressCount": 251,
```

```
            "CidrBlock": "10.0.1.0/24",
            "DefaultForAz": false,
            "MapPublicIpOnLaunch": false,
            "State": "pending",
            "SubnetId": "subnet-09f8f7f06c27cb0a0",
            "VpcId": "vpc-0ca37d4650963adbb",
...
```

2. Create a second subnet, `10.0.2.0/24`, on the `us-east-1b` availability zone:

```
$ aws ec2 create-subnet --vpc-id vpc-0ca37d4650963adbb --c
{
    "Subnet": {
        "AvailabilityZone": "us-east-1b",
        "AvailabilityZoneId": "use1-az1",
        "AvailableIpAddressCount": 251,
        "CidrBlock": "10.0.2.0/24",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "pending",
        "SubnetId": "subnet-04b78ed9b5f96d76e",
        "VpcId": "vpc-0ca37d4650963adbb",
...
```

3. Create a third subnet, `10.0."3".0/24`, on `us-east-1b` again:

```
$ aws ec2 create-subnet --vpc-id vpc-0ca37d4650963adbb --c
{
    "Subnet": {
        "AvailabilityZone": "us-east-1b",
        "AvailabilityZoneId": "use1-az1",
        "AvailableIpAddressCount": 251,
        "CidrBlock": "10.0.3.0/24",
        "DefaultForAz": false,
        "MapPublicIpOnLaunch": false,
        "State": "pending",
        "SubnetId": "subnet-026058e32f09c28af",
        "VpcId": "vpc-0ca37d4650963adbb",
...
```

4. Create a fourth subnet, `10.0.4.0/24`, on `us-east-1a` again:

```
$ aws ec2 create-subnet --vpc-id vpc-0ca37d4650963adbb --c
{
    "Subnet": {
        "AvailabilityZone": "us-east-1a",
        "AvailabilityZoneId": "use1-az6",
        "AvailableIpAddressCount": 251,
```

```
            "CidrBlock": "10.0.4.0/24",
            "DefaultForAz": false,
            "MapPublicIpOnLaunch": false,
            "State": "pending",
            "SubnetId": "subnet-08e16157c15cefcbc",
            "VpcId": "vpc-0ca37d4650963adbb",
    ...
```

Let's make the first two subnets public-facing ones and the last two subnets private ones. This means the public-facing subnet can be accessible from the internet, which allows it to have a public IP address. On the other hand, the private subnet doesn't have a reachability from the internet. To do that, you need to set up gateways and routing tables.

Internet gateways and NAT-GW

In most cases, your VPC needs to have a connection with the public internet. In this case, you need to create an **Internet Gateway** (**IGW**) to attach to your VPC.

In the following example, an IGW is created and attached to `vpc-0ca37d4650963adbb`:

```
//create IGW, it returns IGW id as igw-01769bff334dcc035
$ aws ec2 create-internet-gateway
{
"InternetGateway": {
"Attachments": [],
"InternetGatewayId": "igw-01769bff334dcc035",
"Tags": []
}
}
//attach igw-01769bff334dcc035 to vpc-0ca37d4650963adbb
$ aws ec2 attach-internet-gateway --vpc-id vpc-0ca37d4650963adbb --internet-gateway-id igw-01769bff334dcc035
```

Once the IGW is attached, set a routing table (default gateway) for a subnet that points to the IGW. If a default gateway points to an IGW, this subnet is able to have a public IP address and access from/to the internet. Therefore, if the default gateway doesn't point to IGW, it's determined as a private subnet, which means no public access.

In the following example, a routing table is created that points to IGW and is set to the public subnet:

```
//create route table within vpc-0ca37d4650963adbb
//it returns route table id as rtb-0f45fc46edec61d8f
$ aws ec2 create-route-table --vpc-id vpc-0ca37d4650963adbb
{
"RouteTable": {
"Associations": [],
"PropagatingVgws": [],
"RouteTableId": "rtb-0f45fc46edec61d8f",
...
```

//then set default route (0.0.0.0/0) as igw-01769bff334dcc035
$ aws ec2 create-route --route-table-id rtb-0f45fc46edec61d8f --gateway-id igw-01769bff334dcc035 --destination-cidr-block 0.0.0.0/0

//finally, update public 2 subnets to use this route table
$ aws ec2 associate-route-table --route-table-id rtb-0f45fc46edec61d8f --subnet-id subnet-09f8f7f06c27cb0a0

$ aws ec2 associate-route-table --route-table-id rtb-0f45fc46edec61d8f --subnet-id subnet-026058e32f09c28af

//public subnet can assign public IP when launch EC2 $ aws ec2 modify-subnet-attribute --subnet-id subnet-09f8f7f06c27cb0a0 --map-public-ip-on-launch

$ aws ec2 modify-subnet-attribute --subnet-id subnet-026058e32f09c28af --map-public-ip-on-launch

On the other hand, the private subnet doesn't need a public IP address. However, a private subnet sometimes needs to access the internet, for example, to download some packages and access the AWS service. In this case, we still have an option to connect to the internet. This is called a **Network Address Translation Gateway** (**NAT-GW**).

A NAT-GW allows private subnets to access the public internet through the NAT-GW. Consequently, the NAT-GW must be located at a public subnet, and the private subnet routing table points to the NAT-GW as a default gateway. Note that in order to access a NAT-GW on the public network, it needs an **Elastic IP** (**EIP**) attached to the NAT-GW.

In the following example, a NAT-GW is created: **//allocate EIP, it returns allocation id as eipalloc-044f4dbafe870a04a $ aws ec2 allocate-address**
{
"PublicIp": "54.161.228.168",
"AllocationId": "eipalloc-044f4dbafe870a04a",
"PublicIpv4Pool": "amazon",
"Domain": "vpc"

}

//create NAT-GW on public subnet (subnet-09f8f7f06c27cb0a0) //also assign EIP eipalloc-044f4dbafe870a04a $ aws ec2 create-nat-gateway --subnet-id subnet-09f8f7f06c27cb0a0 --allocation-id eipalloc-044f4dbafe870a04a
{
"NatGateway": {
"CreateTime": "2018-12-09T20:17:33.000Z",
"NatGatewayAddresses": [
{
"AllocationId": "eipalloc-044f4dbafe870a04a"
}
],
"NatGatewayId": "nat-05e34091f53f10172",
"State": "pending",
"SubnetId": "subnet-09f8f7f06c27cb0a0",
"VpcId": "vpc-0ca37d4650963adbb"
}
}

*Unlike an IGW, you can deploy a NAT-GW on single **Availability Zone** (**AZ**). If you need a high availability NAT-GW, you need to deploy a NAT-GW on each AZ. However, AWS charges you an additional hourly cost for an Elastic IP and NAT-GW. Therefore, if you wish to save costs, launch a single NAT-GW on a single AZ, as in the preceding example.*

Creating a NAT-GW takes a few minutes. Once it's created, update a private subnet routing table that points to the NAT-GW, and then any EC2 instances are able to access the internet; again, however, due to no public IP address on the private subnet, there's no chance of access from the public internet to the private subnet EC2 instances.

In the following example, an update routing table for the private subnet points to a NAT-GW as the default gateway: **//as same as public route, need to create a route table first $ aws ec2 create-route-table --vpc-id vpc-0ca37d4650963adbb**
{
"RouteTable": {
"Associations": [],
"PropagatingVgws": [],

"RouteTableId": "rtb-08572c332e7e4f14e",
...

//then assign default gateway as NAT-GW $ aws ec2 create-route --route-table-id rtb-08572c332e7e4f14e --nat-gateway-id nat-05e34091f53f10172 --destination-cidr-block 0.0.0.0/0

//finally update private subnet routing table $ aws ec2 associate-route-table --route-table-id rtb-08572c332e7e4f14e --subnet-id subnet-04b78ed9b5f96d76e

$ aws ec2 associate-route-table --route-table-id rtb-08572c332e7e4f14e --subnet-id subnet-08e16157c15cefcbc

Overall, there are four subnets that have been configured as two public subnets and two private subnets. Each subnet has a default route to use IGW and NAT-GW as follows. Note that the ID varies because AWS assigns a unique identifier:

Types of subnet	CIDR block	Availability zone	Subnet ID	Route table ID	
Public	10.0.1.0/24	us-east-1a	subnet-09f8f7f06c27cb0a0	rtb-0f45fc46edec61d8f	igw 017 (IC
Private	10.0.2.0/24	us-east-1b	subnet-04b78ed9b5f96d76e	rtb-08572c332e7e4f14e	nat 05e (N
					igw

Public	10.0.3.0/24	us-east-1b	subnet- 026058e32f09c28af	rtb- 0f45fc46edec61d8f	017 (I(
Private	10.0.4.0/24	us-east-1a	subnet- 08e16157c15cefcbc	rtb- 08572c332e7e4f14e	nat 05e (N

Technically, you can still assign a public IP to a private subnet EC2 instance, but there's no default gateway to the internet (IGW). Therefore, a public IP will just be wasted and it won't have connectivity from the internet.

Now if you launch an EC2 instance on the public subnet, it becomes public facing, so you can serve your application from this subnet.

On the other hand, if you launch an EC2 instance on the private subnet, it can still access the internet through the NAT-GW, but there will be no access from the internet. However, it can still access it from the EC2 host on the public subnet. So, ideally, you can deploy internal services such as databases, middleware, and monitoring tools on the private subnet.

Security group

Once VPC and subnets with related gateways/routes are ready, you can create EC2 instances. However, at least one access control needs to be created beforehand; this is called a **security group**. It can define ingress (incoming network access) and egress (outgoing network access) firewall rules.

In the following example, a security group and a rule for `public` subnet hosts are created that allows SSH from your machine's IP address, as well as open HTTP (`80/tcp`) world-wide: **//create one security group for public subnet $ aws ec2 create-security-group --vpc-id vpc-0ca37d4650963adbb --group-name public --description "public facing host"**
{
"GroupId": "sg-03973d9109a19e592"
}
//check your machine's public IP (if not sure, use 0.0.0.0/0 as temporary) $ curl ifconfig.co 98.234.106.21
//public facing machine allows ssh only from your machine $ aws ec2 authorize-security-group-ingress --group-id sg-03973d9109a19e592 --protocol tcp --port 22 --cidr 98.234.106.21/32
//public facing machine allow HTTP access from any host (0.0.0.0/0) $ aws ec2 authorize-security-group-ingress --group-id sg-03973d9109a19e592 --protocol tcp --port 80 --cidr 0.0.0.0/0

Next, create a security group for a `private` subnet host that allows SSH from the `public` subnet host. In this case, specifying a public subnet security group ID (`sg-03973d9109a19e592`) instead of a CIDR block is convenient: **//create security group for private subnet $ aws ec2 create-security-group --vpc-id vpc-0ca37d4650963adbb --group-name private --description "private subnet host"**
{
"GroupId": "sg-0f4058a729e2c207e"
}

//private subnet allows ssh only from public subnet host security group $ aws ec2 authorize-security-group-ingress --group-id sg-0f4058a729e2c207e --protocol tcp --port 22 --source-group sg-03973d9109a19e592

//it also allows HTTP (80/TCP) from public subnet security group $ aws ec2 authorize-security-group-ingress --group-id sg-0f4058a729e2c207e --protocol tcp --port 80 --source-group sg-03973d9109a19e592

When you define a security group for a public subnet, it's highly recommended that it's reviewed by a security expert. This is because, once you deploy an EC2 instance onto the public subnet, it has a public IP address and then everyone including crackers and bots are able to access your instances directly.

Overall, there are two security groups that have been created, as follows:

Name	**Security group ID**	**Allow ssh (22/TCP)**	**Allow HTTP (80/TCP)**
Public	`sg-03973d9109a19e592`	Your machine `(98.234.106.21)`	`0.0.0.0/0`
Private	`sg-0f4058a729e2c207e`	`public sg (sg-03973d9109a19e592)`	`public sg (sg-03973d9109a19e592)`

EC2 and EBS

EC2 is one important service in AWS that you can use to launch a VM on your VPC. Based on hardware spec (CPU, memory, and network), there are several types of EC2 instances that are available on AWS. When you launch an EC2 instance, you need to specify VPC, subnet, security group, and SSH keypair. Consequently, all of these must be created beforehand.

Because of previous examples, the only last step is `ssh-keypair`. Let's make `ssh-keypair`:

```
//create keypair (aws_rsa, aws_rsa.pub)
$ ssh-keygen -f ~/.ssh/aws_rsa -N ""

//register aws_rsa.pub key to AWS
$ aws ec2 import-key-pair --key-name=my-key --public-key-material "`cat ~/.ssh/aws_rsa.p
{
    "KeyFingerprint": "73:89:80:1f:cc:25:94:7a:ba:f4:b0:81:ae:d8:bb:92",
    "KeyName": "my-key"
}

//launch public facing host, using Amazon Linux (ami-009d6802948d06e52) on us-east-1
$ aws ec2 run-instances --image-id ami-009d6802948d06e52 --instance-type t2.nano --key-r

//launch private subnet host
$ aws ec2 run-instances --image-id ami-009d6802948d06e52 --instance-type t2.nano --key-r
```

After a few minutes, check the EC2 instance's status on the AWS web console; this shows a `public` subnet host that has a public IP address. On the other hand, a private subnet host doesn't have a public IP address:

IPv4 Public IP	Key Name	Security Groups	VPC ID	Subnet ID	Private IP Addr
-	my-key	private	vpc-0ca37d4650963adbb	subnet-04b78ed9b5f96d76e	10.0.2.116
54.208.77.168	my-key	public	vpc-0ca37d4650963adbb	subnet-09f8f7f06c27cb0a0	10.0.1.41

Let's use your SSH private key to log in to the EC2 instance using the IPv4 public IP address, as follows:

```
//add private keys to ssh-agent
$ ssh-add ~/.ssh/aws_rsa


//ssh to the public subnet host with -A (forward ssh-agent) option
$ ssh -A ec2-user@54.208.77.168
...
```

```
       __| __|_ )
       _| ( / Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
1 package(s) needed for security, out of 5 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-0-1-41 ~]$
```

Now you're in the public subnet host (`54.208.77.168`), but this host also has an internal (private) IP address because it's deployed in the `10.0.1.0/24` subnet, therefore the private address range must be `10.0.1.1`—`10.0.1.254`:

```
[ec2-user@ip-10-0-1-41 ~]$ ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 9001
        inet 10.0.1.41 netmask 255.255.255.0 broadcast 10.0.1.255
        inet6 fe80::cf1:1ff:fe9f:c7b2 prefixlen 64 scopeid 0x20<link>
...
```

Let's install the `nginx` web server on the public host as follows:

```
$ amazon-linux-extras |grep nginx
  4 nginx1.12 available [ =1.12.2 ]
$ sudo amazon-linux-extras install nginx1.12
$ sudo systemctl start nginx
```

Then, go back to your machine and check the website for `54.208.77.168`:

```
[ec2-user@ip-10-0-1-41 ~]$ exit
logout
Connection to 54.208.77.168 closed.

$ curl -I 54.208.77.168
HTTP/1.1 200 OK
Server: nginx/1.12.2
...
```

In addition, within the same VPC, there's reachability for other availability zones; therefore, you can SSH from the EC2 host on the public subnet to the private subnet host (`10.0.2.116`). Note that we're using the `ssh -A` option that forwards `ssh-agent`, so there's no need to create a `~/.ssh/id_rsa` file on the EC2 host:

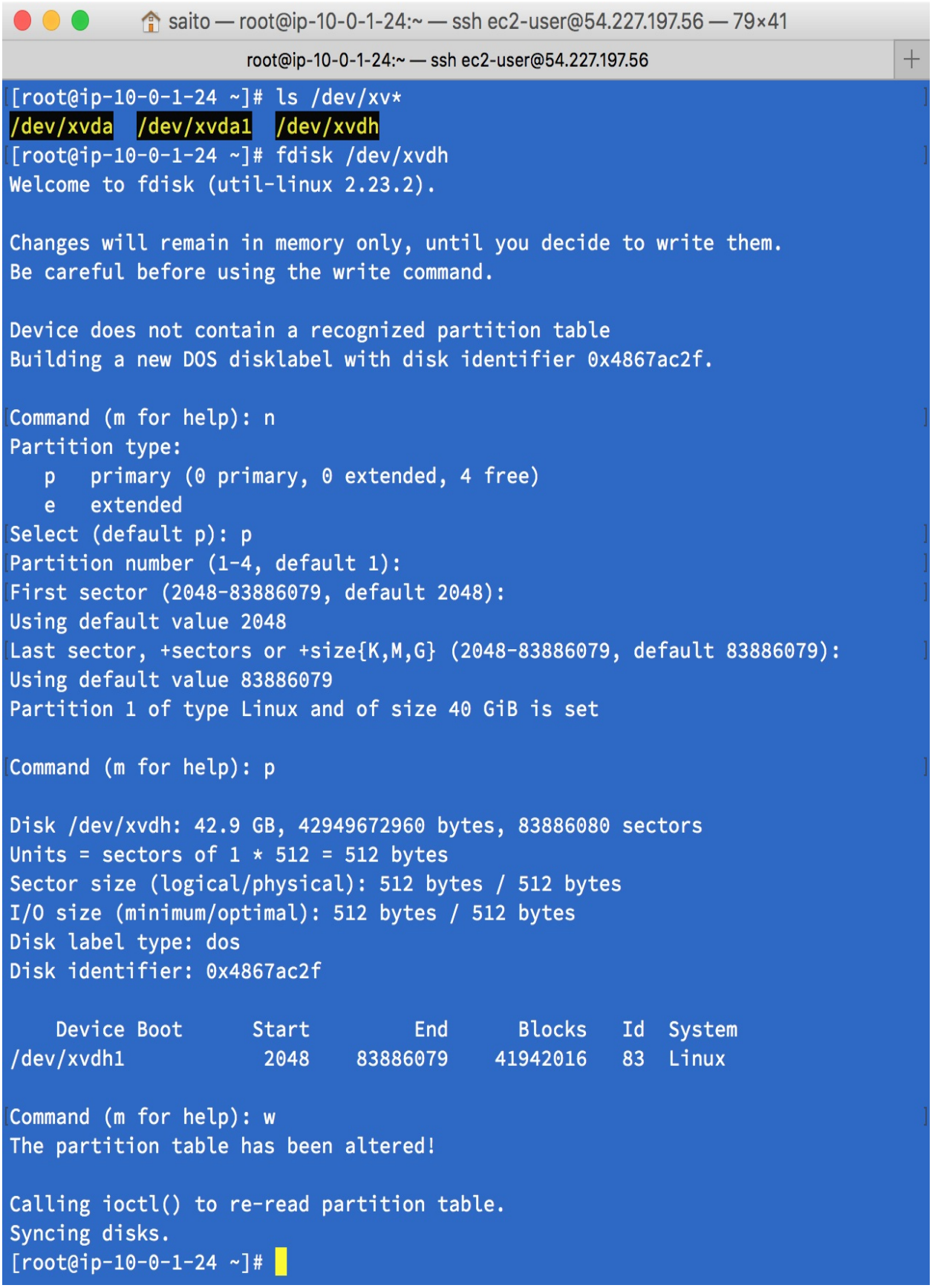
```
[ec2-user@ip-10-0-1-41 ~]$ ssh 10.0.2.116
...

       __| __|_ )
       _| ( / Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
1 package(s) needed for security, out of 5 available
Run "sudo yum update" to apply all updates.
[ec2-user@ip-10-0-2-116 ~]$
```

In addition to EC2, there's another important functionality named disk management. AWS provides a flexible disk management service called **Elastic Block Store** (**EBS**). You may create one or more persistent data storage that can attach to an EC2 instance. From an EC2 point of view, EBS is one of HDD/SSD. Once you terminate (delete) an EC2 instance, EBS and its contents may remain and then reattach to another EC2 instance.

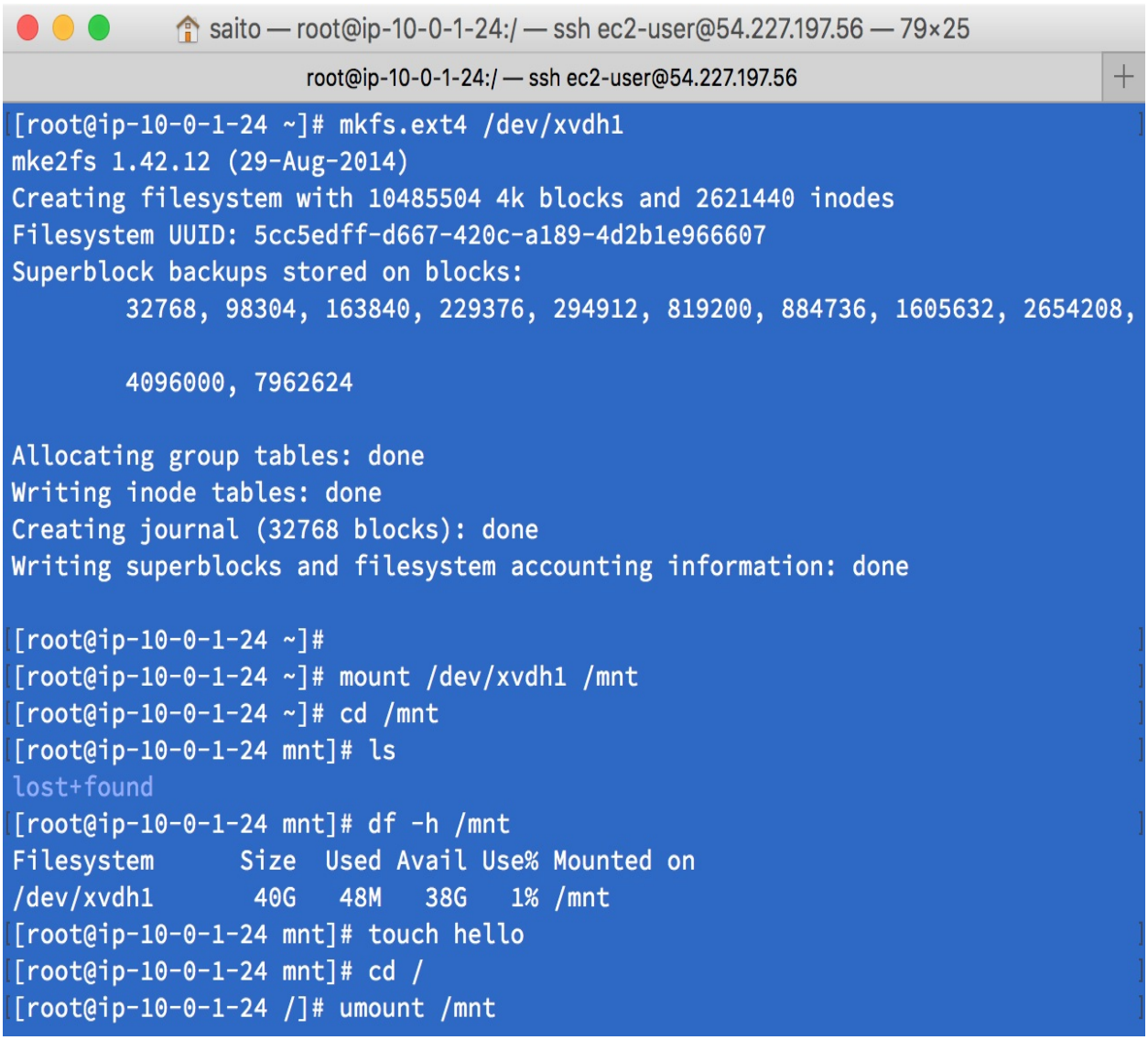
In the following example, one volume that has 40 GB capacity is created and then attached to a public subnet host (instance ID, `i-0f2750f65dd857e54`):

```
//create 40GB disk at us-east-1a (as same as EC2 public subnet instance)
$ aws ec2 create-volume --availability-zone us-east-1a --size 40 --volume-type standard
{
    "CreateTime": "2018-12-09T22:13:41.000Z",
    "VolumeType": "standard",
    "SnapshotId": "",
    "VolumeId": "vol-006aada6fa87c0060",
    "AvailabilityZone": "us-east-1a",
    "Size": 40,
    "State": "creating",
    "Encrypted": false
}


//attach to public subnet host as /dev/xvdh
$ aws ec2 attach-volume --device xvdh --instance-id i-0f2750f65dd857e54 --volume-id vol-
{
    "State": "attaching",
    "InstanceId": "i-0f2750f65dd857e54",
    "AttachTime": "2018-12-09T22:15:32.134Z",
    "VolumeId": "vol-006aada6fa87c0060",
    "Device": "xvdh"
}
```

After attaching the EBS volume to the EC2 instance, the Linux kernel recognizes `/dev/xvdh` as specified, and then you need to do partitioning in order to use this device, as follows:


saito — root@ip-10-0-1-24:~ — ssh ec2-user@54.227.197.56 — 79×41
root@ip-10-0-1-24:~ — ssh ec2-user@54.227.197.56
[root@ip-10-0-1-24 ~]# ls /dev/xv*
/dev/xvda /dev/xvda1 /dev/xvdh
[root@ip-10-0-1-24 ~]# fdisk /dev/xvdh
Welcome to fdisk (util-linux 2.23.2).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.
Device does not contain a recognized partition table
Building a new DOS disklabel with disk identifier 0x4867ac2f.
Command (m for help): n
Partition type:
p primary (0 primary, 0 extended, 4 free)
e extended
Select (default p): p
Partition number (1-4, default 1):
First sector (2048-83886079, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-83886079, default 83886079):
Using default value 83886079
Partition 1 of type Linux and of size 40 GiB is set
Command (m for help): p
Disk /dev/xvdh: 42.9 GB, 42949672960 bytes, 83886080 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0x4867ac2f
Device Boot Start End Blocks Id System
/dev/xvdh1 2048 83886079 41942016 83 Linux
Command (m for help): w
The partition table has been altered!
Calling ioctl() to re-read partition table.
Syncing disks.
[root@ip-10-0-1-24 ~]#

In this example, we made one partition as `/dev/xvdh1`, so you can create a filesystem in `ext4` format on `/dev/xvdh1` and then you can mount to use this device on an EC2 instance:



After unmounting the volume, you are free to detach this volume and then re-attach it whenever needed:

```
$ aws ec2 detach-volume --volume-id vol-006aada6fa87c0060
{
    "InstanceId": "i-0f2750f65dd857e54",
    "VolumeId": "vol-006aada6fa87c0060",
    "State": "detaching",
    "Device": "xvdh",
    "AttachTime": "2018-12-09T22:15:32.000Z"
}
```

ELB

AWS provides a powerful software-based load balancer called **classic load balancer**. This was known as **Elastic Load Balancer** (**ELB**), which allows you to load balance network traffic to one or multiple EC2 instances. In addition, ELB can offload SSL/TLS encryption/decryption and it supports multi-availability zone.

So, why is it a classic load balancer in particular? This is because AWS introduced new types of load balancers: network load balancer (for L4) and application load balancer (for L7). Therefore, ELB became classic. However, while ELB is stable and robust, Amazon EKS will use load balancer by default, so we keep using ELB.

In the following example, an ELB is created and associated with a public subnet host, `nginx` (`80/TCP`). Because ELB also needs a security group, create a new one for this, first: **// Create New Security Group for ELB**
$ aws ec2 create-security-group --vpc-id vpc-0ca37d4650963adbb --group-name elb --description "elb sg"
{
"GroupId": "sg-024f1c5315bac6b9e"
}

// ELB opens TCP port 80 for all IP addresses (0.0.0.0/0)
$ aws ec2 authorize-security-group-ingress --group-id sg-024f1c5315bac6b9e --protocol tcp --port 80 --cidr 0.0.0.0/0

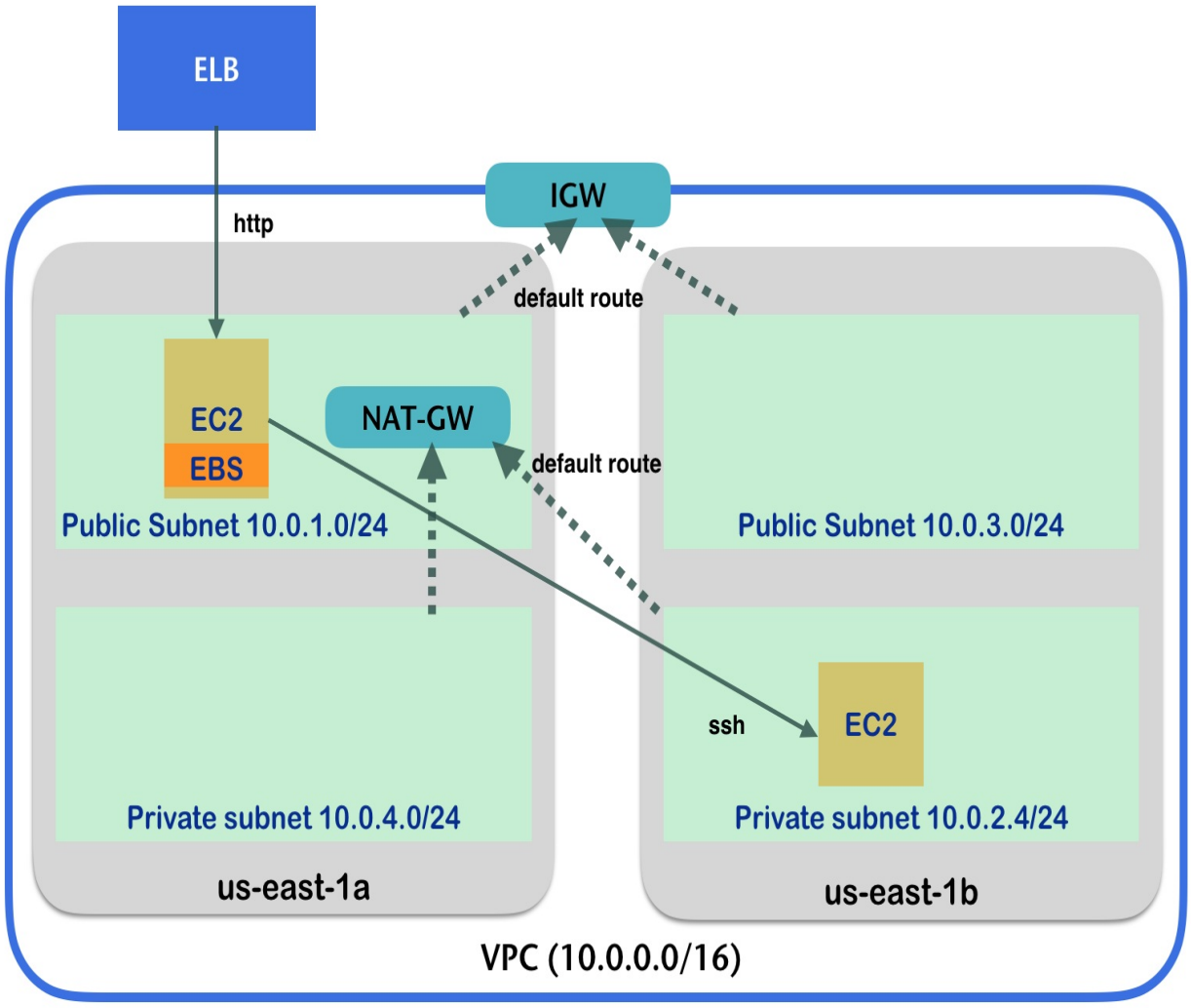
// create ELB on public subnets $ aws elb create-load-balancer --load-balancer-name public-elb --listeners Protocol=HTTP,LoadBalancerPort=80,InstanceProtocol=HTTP,InstancePor --subnets subnet-09f8f7f06c27cb0a0 subnet-026058e32f09c28af --security-group sg-024f1c5315bac6b9e
{
"DNSName": "public-elb-1952792388.us-east-1.elb.amazonaws.com"
}

// Register an EC2 instance which runs nginx $ aws elb register-instances-with-load-balancer --load-balancer-name public-elb --instances i-0f2750f65dd857e54

// You can access to ELB from your laptop
$ curl -I public-elb-1952792388.us-east-1.elb.amazonaws.com
HTTP/1.1 200 OK
Accept-Ranges: bytes
Content-Length: 3520
Content-Type: text/html
Date: Mon, 17 Dec 2018 06:05:45 GMT
ETag: "5bbfda61-dc0"
Last-Modified: Thu, 11 Oct 2018 23:18:57 GMT
Server: nginx/1.12.2
Connection: keep-alive ...

Overall, we've discussed how to configure AWS components. The following is a summary and diagram about major components and relationships:

- One VPC that has an **Internet Gateway** (**IGW**)
- Two subnets (public and private) on `us-east-1a`
- Two subnets (public and private) on `us-east-1b`
- One NAT-GW
- One public EC2 instance on public subnet with EBS
- One private EC2 instance on private subnet
- ELB that forwards the traffic to a public EC2 instance


ELB
IGW
http
default route
EC2
EBS
NAT-GW
default route
Public Subnet 10.0.1.0/24
Public Subnet 10.0.3.0/24
ssh
EC2
Private subnet 10.0.4.0/24
Private subnet 10.0.2.4/24
us-east-1a
us-east-1b
VPC (10.0.0.0/16)

Amazon EKS

We've discussed some AWS components that are quite easy for setting up networks, virtual machines, storage, and load balancers. Consequently, there are a variety of ways to set up Kubernetes on AWS such as kubeadm (https://github.com/kubernetes/kubeadm), kops (https://github.com/kubernetes/kops), and kubespray (https://github.com/kubernetes-sigs/kubespray).

In addition, since June 2018, AWS starts to provide a new Service, which is called Amazon Elastic Container Service for Kubernetes (https://aws.amazon.com/eks/), in short **EKS**. This is similar to Google Kubernetes Engine (https://cloud.google.com/kubernetes-engine/) and Azure Kubernetes Service (https://docs.microsoft.com/en-us/azure/aks/), the managed Kubernetes service.

*AWS also provides another container orchestration service that's called Amazon **Elastic Container Service (ECS)** https://aws.amazon.com/ecs/. AWS ECS isn't a Kubernetes service, but it's fully integrated into AWS components to launch your container application.*

AWS EKS uses AWS components such as VPC, security groups, EC2 instance, EBS, ELB, IAM, and so on, to set up a Kubernetes cluster. This also manages the Kubernetes cluster that patches and replaces the problematic component 24/7. As a result of the constant management, the user will offload the efforts of installation, configuration and monitoring to the Kubenetes cluster, while only needing to pay AWS on an hourly basis.

It's beneficial for the user that AWS provides a fully tested combination of AWS components and Kubernetes versions. This means that the user can start to use the production grade of Kubernetes on AWS within minutes.

Let's explore AWS EKS to learn how AWS integrates Kubernetes into AWS components.

Deep dive into AWS EKS

AWS EKS has two main components. These components are as follows:

- Control plane
- Worker nodes

Control Plane is the managed Kubernetes master by AWS, which includes an `etcd` database. AWS helps to deploy the Kubernetes master on multiple availability zones. A user can monitor and access the control plane via the AWS Web Console or AWS CLI. As well as this, a user can gain access to Kubernetes API server via Kubernetes clients such as the `kubectl` command.

As of December 2018, AWS only provides a custom **Amazon Machine Images** (**AMI**) for worker nodes. AWS provides neither Web Console nor AWS CLI to create and configure the worker nodes yet. Therefore, the user needs to use that AMI to launch EC2 instance(s) to configure worker nodes manually.

Amazon and Weaveworks made an open source project named `eksctl` (`https://eksctl.io/`). It's easier to deploy an EKS cluster than AWS CLI and some manual steps.
If you have difficulty understanding AWS basics and EKS provisioning, it's recommended to use `eksctl` instead.

Fortunately, AWS provides a CloudFormation template that's easy to use to launch and configure worker nodes, so let's extend the previous example of VPC to set up Amazon EKS. To do that, you need to prepare the following settings beforehand:

- Set up the IAM Service Role (defines which AWS user can create EKS resources) as follows:

```
$ aws iam create-role --role-name eksServiceRole --assume-role-policy-docume
$ aws iam attach-role-policy --role-name eksServiceRole --policy-arn arn:aws
$ aws iam attach-role-policy --role-name eksServiceRole --policy-arn arn:aws
```

- Set up the security group (assign to `Control Plane`, then worker nodes use this

security group to allow access from `Control Plane`):

```
$ aws ec2 create-security-group --vpc-id vpc-0ca37d4650963adbb --group-name
{
 "GroupId": "sg-0fbac0a39bf64ba10"
}
```

- Tag this to a private subnet (to tell Internal ELB that this is a private subnet):

```
$ aws ec2 create-tags --resources subnet-04b78ed9b5f96d76e --tags Key=kubern

$ aws ec2 create-tags --resources subnet-08e16157c15cefcbc --tags Key=kubern
```

Launching the EKS control plane

The EKS control plane is a managed Kubernetes master; you just need to use the AWS CLI to specify your IAM, subnets, and security group. This example also specifies the Kubernetes version as 1.10: **// Note: specify all 4 subnets**
$ aws eks create-cluster --name chap10 --role-arn arn:aws:iam::xxxxxxxxxxxx:role/eksServiceRole --resources-vpc-config subnetIds=subnet-09f8f7f06c27cb0a0,subnet-04b78ed9b5f96d76e,subnet-026058e32f09c28af,subnet-08e16157c15cefcbc,securityGroupIds=sg-0fbac0a39bf64ba10 --kubernetes-version 1.10

This takes around 10 minutes to complete. You can check the status by typing `aws eks describe-cluster --name chap10`. Once your control plane status is `ACTIVE`, you can start to set up `kubeconfig` to access your Kubernetes API server.

However, AWS integrates Kubernetes API access control with AWS IAM credentials. So, you need to use `aws-iam-authenticator` (https://github.com/kubernetes-sigs/aws-iam-authenticator) to generate a token when you run the `kubectl` command.

This simply downloads an `aws-iam-authenticator` binary and installs it to the default command search path (for example, `/usr/local/bin`), then verifies whether `aws-iam-authenticator` works or not, using the following command: **$ aws-iam-authenticator token -i chap10**

If you see the `authenticator` token, run the AWS CLI to generate `kubeconfig`, as follows: **$ aws eks update-kubeconfig --name chap10**

If you succeed in creating `kubeconfig`, you can check whether you can access the Kubernetes master using the `kubectl` command, as follows: **$ kubectl cluster-info**
$ kubectl get svc

At this moment, you don't see any Kubernetes nodes (`kubectl get nodes` returns empty). So, you need one more step to add worker nodes (Kubernetes nodes).

You must use the same IAM user that creates a control plane and access the API server with `aws-iam-authenticator`. For example, it won't work if you create an EKS control plane by the AWS root account and then access the API server through one of the IAM users. You can see an

error such as You must be logged in to the server (Unauthorized) when using kubectl.

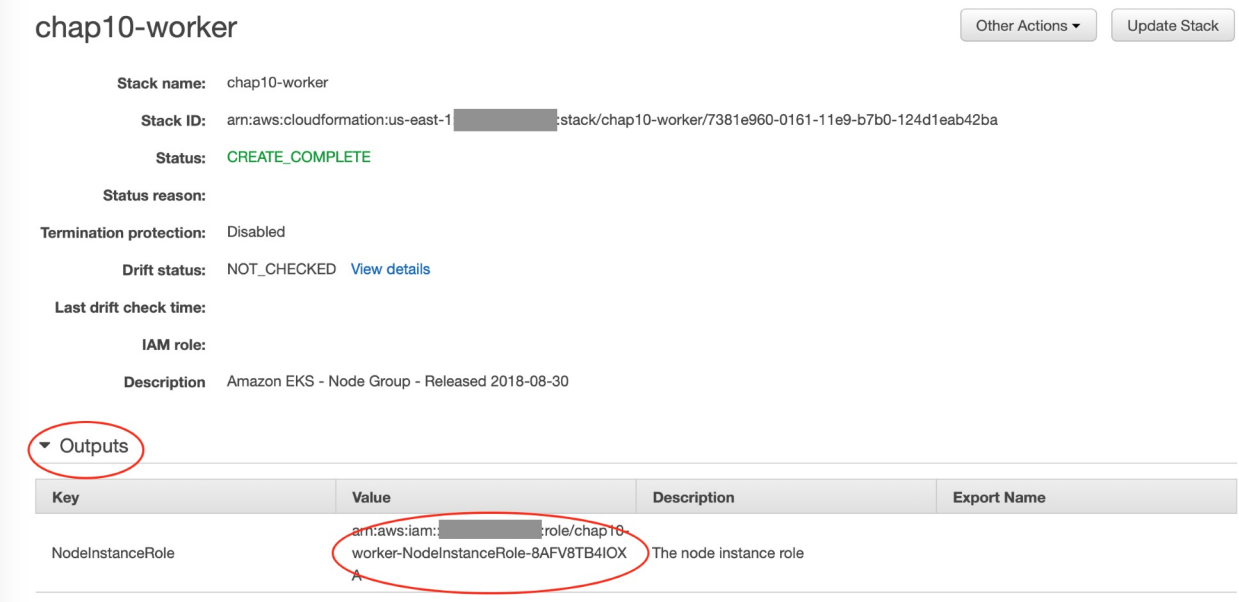
Adding worker nodes

As discussed, AWS doesn't allow the AWS CLI to set up EKS worker nodes. Instead, use CloudFormation. This creates the necessary AWS component for worker nodes, such as security groups, AutoScaling groups, and IAM Instance Roles. Furthermore, the Kubernetes master needs an IAM Instance Role when a worker node joins the Kubernetes cluster. It's highly recommended to use the CloudFormation template to launch worker nodes.

CloudFormation execution steps are simple and follow the AWS EKS documentation, `https://docs.aws.amazon.com/eks/latest/userguide/launch-workers.html`. Use the S3 template URL, `https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2018-12-10/amazon-eks-nodegroup.yaml`, and then specify the parameters as in the following example:

Parameter	Value
`Stack name`	`chap10-worker`
`ClusterName`	`chap10 (must be match to EKS control plane name)`
`ClusterControlPlaneSecurityGroup`	`sg-0fbac0a39bf64ba10 (eks-control-plane)`
`NodeGroupName`	`chap10 EKS worker node (any name)`
`NodeImageId`	`ami-027792c3cc6de7b5b (version 1.10.x)`
`KeyName`	`my-key`
`VpcId`	`vpc-0ca37d4650963adbb`
`Subnets`	• `subnet-04b78ed9b5f96d76e (10.0.2.0/24)` • `subnet-08e16157c15cefcbc (10.0.4.0/24)` **Note**: only private subnets

CloudFormation execution takes around five minutes to complete, and then you need to get the `NodeInstanceRole` value from Outputs, as follows:



Finally, you can add these nodes to your Kubernetes cluster by adding `ConfigMap`. You can download a `ConfigMap` template from `https://amazon-eks.s3-us-west-2.amazonaws.com/cloudformation/2018-12-10/aws-auth-cm.yaml` and then fill out the Instance Role ARN, as in this example: **$ cat aws-auth-cm.yaml**
apiVersion: v1
kind: ConfigMap
metadata:
name: aws-auth
namespace: kube-system
data:
mapRoles: |
- rolearn: arn:aws:iam::xxxxxxxxxxxx:role/chap10-worker-NodeInstanceRole-8AFV8TB4IOXA
username: system:node:{{EC2PrivateDNSName}}
groups:
- system:bootstrappers
- system:nodes

$ kubectl create -f aws-auth-cm.yaml
configmap "aws-auth" created

After a few minutes, the worker nodes will be registered to your Kubernetes

master, as follows: **$ kubectl get nodes**
NAME STATUS ROLES AGE VERSION
ip-10-0-2-218.ec2.internal Ready <none> 3m v1.10.3
ip-10-0-4-74.ec2.internal Ready <none> 3m v1.10.3
...

Now you can start to use your own Kubernetes cluster on AWS. Deploy your application to take a look at this. Note that, based on the preceding instruction, we deployed the worker nodes on a private subnet, so if you want to deploy an internet-facing Kubernetes Service, you need to use `type:LoadBalancer`. We'll explore this in the next section.

Cloud provider on EKS

AWS EKS integrates Kubernetes cloud provider into AWS components, for instance, elastic load balancer and Elastic Block Store. This section explores how EKS integrates into AWS components.

Storage class

As of December 2018, if you deploy Kubernetes version 1.10, EKS doesn't create storage classes by default. On the other hand, in version 1.11 or above, EKS creates default storage classes automatically. Using the following command, you can check whether a storage class exists or not: **$ kubectl get sc**
No resources found.

In this case, you need to create a storage class to make a storage class. Note that AWS EBS and EC2 are zone sensitive. Therefore, EBS and EC2 must be located on the same availability zone. Therefore, it's recommended to create `StorageClass` for each availability zone as follows: **//Storage Class for us-east-1a**
$ cat storage-class-us-east-1a.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: gp2-us-east-1a
provisioner: kubernetes.io/aws-ebs
parameters:
type: gp2
fsType: ext4
zone: us-east-1a

// Storage Class for us-east-1b
$ cat storage-class-us-east-1b.yaml
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
name: gp2-us-east-1b
provisioner: kubernetes.io/aws-ebs
parameters:
type: gp2
fsType: ext4
zone: us-east-1b # only change this

$ kubectl create -f storage-class-us-east-1a.yaml
storageclass.storage.k8s.io "gp2-us-east-1a" created

$ kubectl create -f storage-class-us-east-1b.yaml
storageclass.storage.k8s.io "gp2-us-east-1b" created

// there are 2 StorageClass
$ kubectl get sc
NAME PROVISIONER AGE
gp2-us-east-1a kubernetes.io/aws-ebs 6s
gp2-us-east-1b kubernetes.io/aws-ebs 3s

`PersistentVolumeClaim` can then specify either `gp2-us-east-1a` or `gp2-us-east-1b` storage class to provision the persistent volume: **$ cat pvc-a.yaml**
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: pvc-a
spec:
storageClassName: "gp2-us-east-1a"
accessModes:
- ReadWriteOnce
resources:
requests:
storage: 10Gi

$ cat pvc-b.yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
name: pvc-b
spec:
storageClassName: "gp2-us-east-1b" # only change this
accessModes:

- ReadWriteOnce
resources:
requests:
storage: 10Gi

$ kubectl create -f pvc-a.yaml
persistentvolumeclaim "pvc-a" created
$ kubectl create -f pvc-b.yaml
persistentvolumeclaim "pvc-b" created

$ kubectl get pv
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE
pvc-20508a71-0187-11e9-bf51-02a2ca4dacd8 10Gi RWO Delete Bound default/pvc-a gp2-us-east-1a 4m
pvc-2235f412-0187-11e9-bf51-02a2ca4dacd8 10Gi RWO Delete Bound 4m

// use AWS CLI to search EBS instances by following command
$ aws ec2 describe-volumes --query "Volumes[*].{Id:VolumeId,AZ:AvailabilityZone,Tags:Tags[?Key=='kubernetes.io/created-for/pv/name'].Value}" --filter "Name=tag-key,Values=kubernetes.io/cluster/chap10"
[
{
"Id": "vol-0fdec40626aac7cc4",
"AZ": "us-east-1a",
"Tags": [
"pvc-20508a71-0187-11e9-bf51-02a2ca4dacd8"
]
},
{
"Id": "vol-0d9ef53eedde70115",
"AZ": "us-east-1b",
"Tags": [

"pvc-2235f412-0187-11e9-bf51-02a2ca4dacd8"
]
}
]

Note that worker nodes have a label of `failure-domain.beta.kubernetes.io/zone`, so you can specify a `nodeSelector` to deploy the pod to the desired availability zone: **$ kubectl get nodes --show-labels**
NAME STATUS ROLES AGE VERSION LABELS
ip-10-0-2-218.ec2.internal Ready <none> 4h v1.10.3 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=t2.small,beta.kubernetes.io/os=linux,failure-domain.beta.kubernetes.io/region=us-east-1,failure-domain.beta.kubernetes.io/zone=us-east-1b,kubernetes.io/hostname=ip-10-0-2-218.ec2.internal
ip-10-0-4-74.ec2.internal Ready <none> 4h v1.10.3 beta.kubernetes.io/arch=amd64,beta.kubernetes.io/instance-type=t2.small,beta.kubernetes.io/os=linux,failure-domain.beta.kubernetes.io/region=us-east-1,failure-domain.beta.kubernetes.io/zone=us-east-1a,kubernetes.io/hostname=ip-10-0-4-74.ec2.internal

```
// nodeSelector specifies us-east-1a, also pvc-a
$ cat pod-us-east-1a.yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    project: devops-with-kubernetes
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    volumeMounts:
      - mountPath: /var/log/nginx
        name: nginx-log
  volumes:
  - name: nginx-log
    persistentVolumeClaim:
      claimName: "pvc-a"
  nodeSelector:
    failure-domain.beta.kubernetes.io/zone: us-east-1a

$ kubectl create -f pod-us-east-1a.yaml
pod "nginx" created

// deploy to 10.0.4.0/24 subnet (us-east-1a)
```

```
$ kubectl get pods -o wide
NAME READY  STATUS RESTARTS  AGE  IP NODE
nginx  1/1  Running  0 18s  10.0.4.33  ip-10-0-4-74.ec2.internal

// successfully to mount PVC-a
$ kubectl exec -it nginx /bin/bash
root@nginx:/# df -h /var/log/nginx
Filesystem Size Used Avail Use% Mounted on
/dev/xvdca 9.8G  37M 9.2G  1% /var/log/nginx
```

Load balancer

EKS also integrates Kubernetes Service into classic load balancer (also known as ELB). When you create a Kubernetes Service by specifying `type:LoadBalancer`, EKS creates the Classic ELB instance and security group, and then associates between ELB and worker nodes automatically.

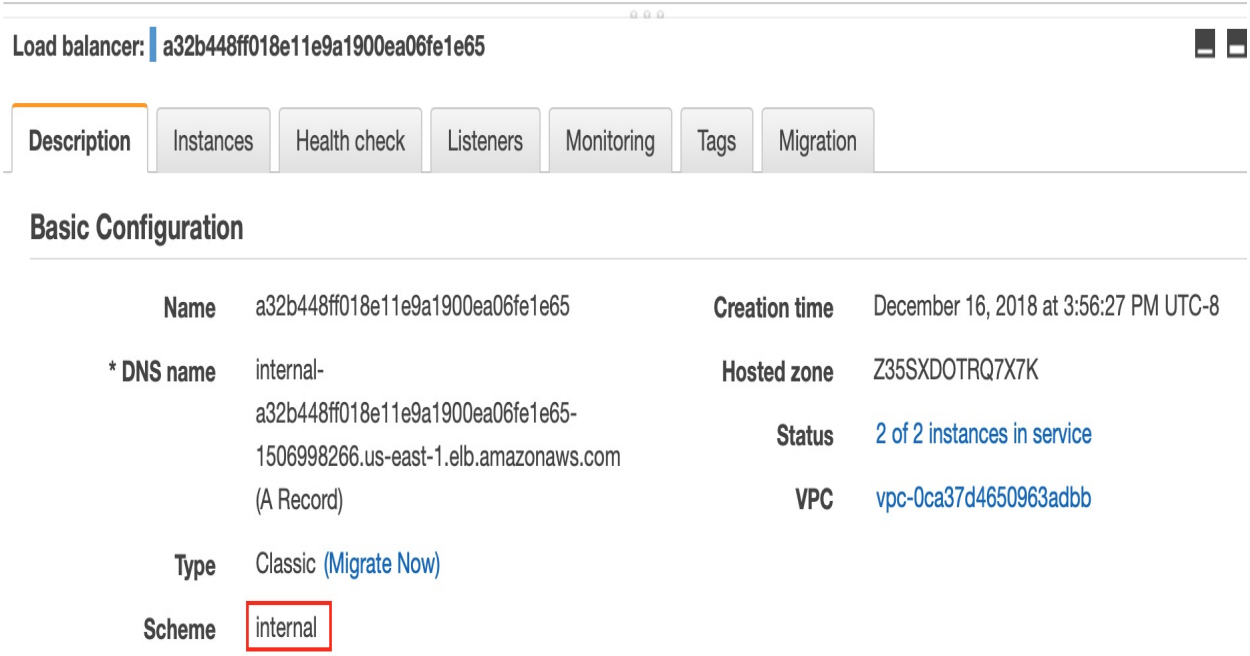
In addition, you can either create an internet-facing (on the public subnet) or internal (on the private subnet) load balancer. If you don't need to serve traffic to the external internet, you should use an internal load balancer for security reasons.

Internal load balancer

Let's create an internal load balancer for the previous `nginx` pod. In order to use an internal load balancer, you need to add an annotation (`service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0`), as follows:

```
$ cat internal-elb.yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx
 annotations:
 service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0
spec:
 ports:
 - protocol: TCP
 port: 80
 type: LoadBalancer
 selector:
 project: devops-with-kubernetes
 app: nginx

$ kubectl create -f internal-elb.yaml
service "nginx" created
```

Then, the EKS Cloud provider will create and configure a classic ELB as in the following screenshot:



Since it's an internal ELB, you can't gain access to the ELB from outside of the AWS network, for example, from your laptop. However, it's useful to expose your application to the outside of the Kubernetes cluster within VPC.

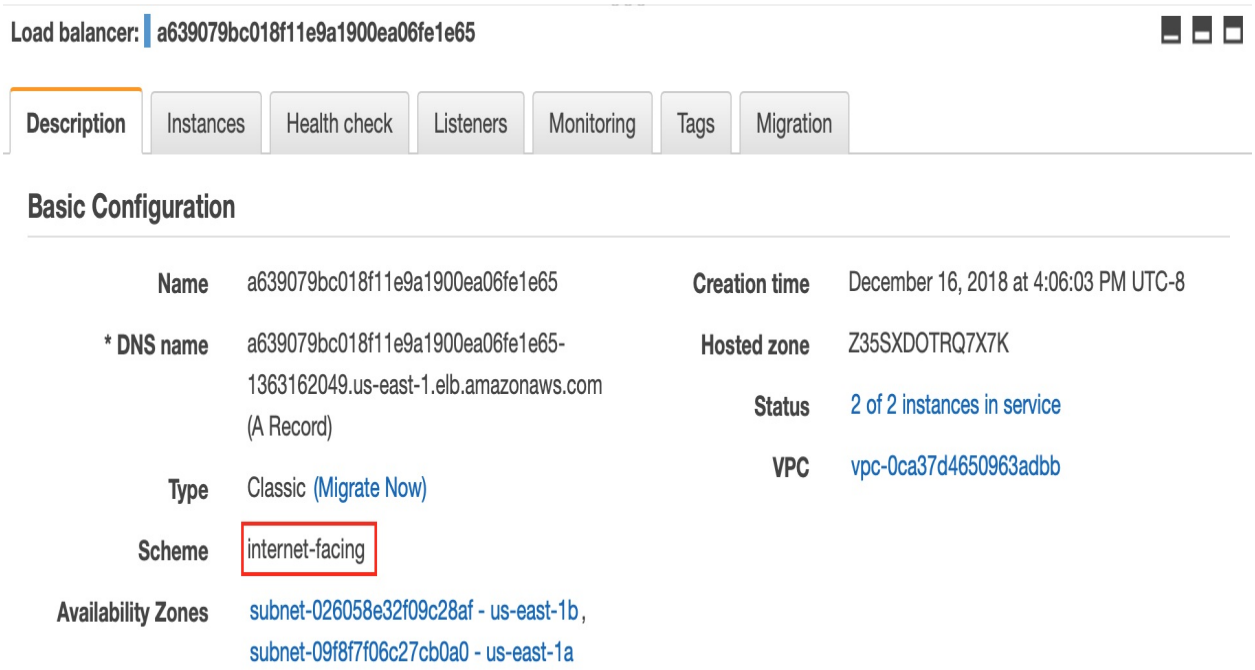
The AWS charges ELB per hour. If your Kubernetes Service serves within Kubernetes cluster pods, you may consider using `type:ClusterIP`*.*

Internet-facing load balancer

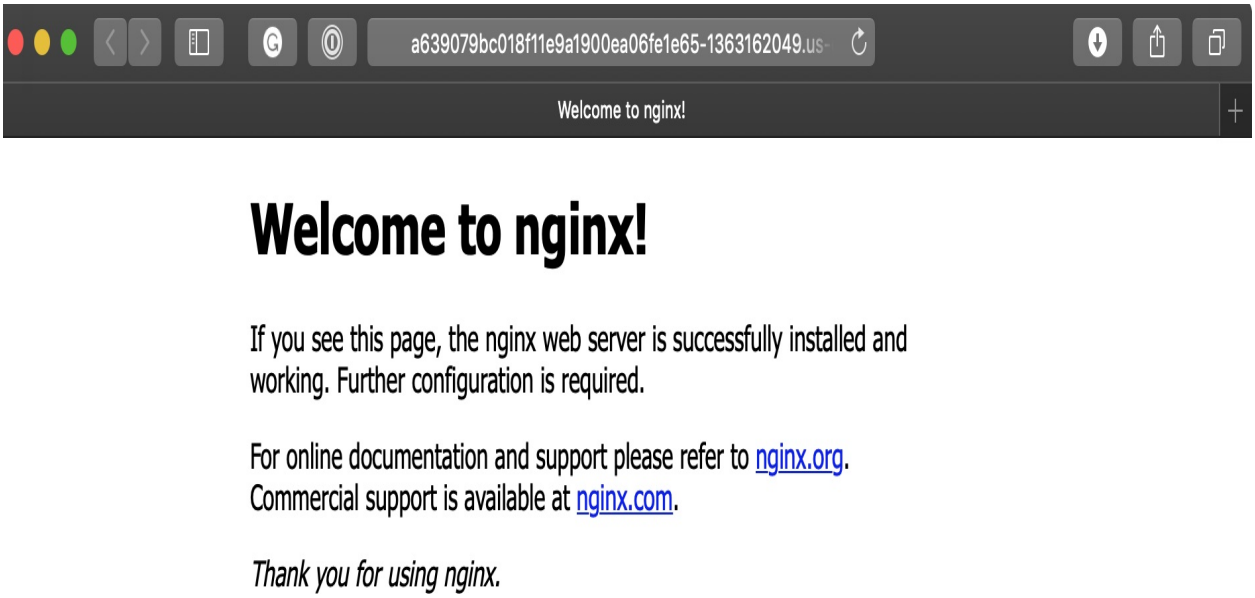
Creating an internet-facing load balancer consists of the same steps as an internal load balancer, but there's no need for an annotation (`service.beta.kubernetes.io/aws-load-balancer-internal: 0.0.0.0/0`):

```
$ cat external-elb.yaml
apiVersion: v1
kind: Service
metadata:
 name: nginx-external
spec:
 ports:
 - protocol: TCP
 port: 80
 type: LoadBalancer
 selector:
 project: devops-with-kubernetes
 app: nginx

$ kubectl create -f external-elb.yaml
service "nginx-external" created
```

When you check the AWS Web Console, you can see that Scheme is internet-facing as follows:



You can access the ELB from your laptop as well:



As you can see from the preceding screenshots, the EKS cloud provider is integrated into Kubernetes Service, which launches a classic ELB. This feature is very powerful for scaling out the traffic volume that dispatches to multiple pods.

*EKS has also already begun to support the use of **Network Load Balancer** (**NLB**), the new version of L4 load balancer in AWS.*

In order to use NLB, you need an additional annotation. This annotation is as follows:

```
metadata:
 name: nginx-external
  annotations:
    service.beta.kubernetes.io/aws-load-balancer-type: "nlb"
```

Updating the Kubernetes version on EKS

When Kubernetes releases a new version, EKS will follow and provide the latest version of Kubernetes for the user in a timely manner. In previous examples, we've used Kubernetes version 1.10. As of December 2018, EKS also supports version 1.11. Let's perform the upgrade to see how EKS handles cluster updates.

The typical upgrade steps are as follows:

1. Upgrade the Kubernetes master via the AWS CLI
2. Create a new version of worker nodes via CloudFormation
3. Add new worker nodes to the Kubernetes cluster (both old and new worker nodes co-exist at this time)
4. Migrate pods from the old worker node to the new one

Mostly, the worker nodes upgrade requires some manual steps. We'll explore this step by step.

Upgrading the Kubernetes master

Upgrading the Kubernetes master involves the simple step of specifying your EKS name and desired new version, as shown in the following. This takes around 30 minutes to complete, based on the condition. Meanwhile, accessing Kubernetes API server (via `kubectl`) might fail. Although pods and Services won't be affected, you need to leave enough time to perform this operation: **$ aws eks update-cluster-version --name chap10 --kubernetes-version 1.11**

```
{
"update": {
"status": "InProgress",
"errors": [],
"params": [
{
"type": "Version",
"value": "1.11"
},
{
"type": "PlatformVersion",
"value": "eks.1"
}
],
"type": "VersionUpdate",
"id": "09688495-4d12-4aa5-a2e8-dfafec1cee17",
"createdAt": 1545007011.285
}
}
```

As you can see in the preceding code, the `aws eks update-cluster-version` command returns update `id`. You can use this ID to check the upgrade status, as follows: **$ aws eks describe-update --name chap10 --update-id 09688495-4d12-4aa5-a2e8-dfafec1cee17**

```
{
"update": {
"status": "InProgress",
```

"errors": [],
...
...

Once the status changes from `InProgress` to `Successful`, you can see the newer version of the API server as follows: **$ kubectl version --short**
Client Version: v1.10.7
Server Version: v1.11.5-eks-6bad6d

Based on differences between older and newer versions of Kubernetes, there are some additional migration steps that we might need to follow. For example, change the DNS service from `kube-dns` to `core-dns`. You need to follow these steps if AWS EKS provides some instructions.

Upgrading worker nodes

After upgrading the Kubernetes master, you can start to upgrade the worker nodes. However, again, there's no AWS CLI support yet, so you need some manual steps to upgrade worker nodes:

1. Create new worker nodes using the same steps as earlier using CloudFormation. However, here, you'll specify the new version of AMI, such as `ami-0b4eb1d8782fc3aea`. You can get an AMI ID list from the AWS documentation via https://docs.aws.amazon.com/eks/latest/userguide/eks-optimized-ami.html.
2. Update a security group for both old and new worker nodes to allow network traffic between them. You can find a security group ID via the AWS CLI or AWS Web Console. For more details on this, please visit the AWS documentation: https://docs.aws.amazon.com/eks/latest/userguide/migrate-stack.html.
3. Update `ConfigMap` to add (not replace) new worker nodes Instance ARNs, as in the following example:

```
$ vi aws-auth-cm.yaml
apiVersion: v1
kind: ConfigMap
metadata:
 name: aws-auth
 namespace: kube-system
data:
 mapRoles: |
 #
 # new version of Worker Nodes
 #
 - rolearn: arn:aws:iam::xxxxxxxxxxx:role/chap10-v11-NodeInstanceRole-10YYF
 username: system:node:{{EC2PrivateDNSName}}
 groups:
 - system:bootstrappers
 - system:nodes
 #
 # old version of Worker Nodes
 #
 - rolearn: arn:aws:iam::xxxxxxxxxxx:role/chap10-worker-NodeInstanceRole-8A
 username: system:node:{{EC2PrivateDNSName}}
 groups:
 - system:bootstrappers
 - system:nodes

//apply command to update ConfigMap
```

```
$ kubectl apply -f aws-auth-cm.yaml

// you can see both 1.10 and 1.11 nodes
$ kubectl get nodes
NAME  STATUS ROLES  AGE  VERSION
ip-10-0-2-122.ec2.internal  Ready  <none> 1m v1.11.5
ip-10-0-2-218.ec2.internal  Ready  <none> 6h v1.10.3
...
```

4. Taint and drain the old nodes to move the pod to the new node:

```
// prevent to assign pod to older Nodes
$ kubectl taint nodes ip-10-0-2-218.ec2.internal key=value:NoSchedule
$ kubectl taint nodes ip-10-0-4-74.ec2.internal key=value:NoSchedule

// move Pod from older to newer Nodes
$ kubectl drain ip-10-0-2-218.ec2.internal --ignore-daemonsets --delete-loca
$ kubectl drain ip-10-0-4-74.ec2.internal --ignore-daemonsets --delete-local

// Old worker node became SchedulingDisabled
$ kubectl get nodes
NAME  STATUS  ROLES  AGE  VERSION
ip-10-0-2-122.ec2.internal  Ready <none> 7m v1.11.5
ip-10-0-2-218.ec2.internal  Ready,SchedulingDisabled  <none> 7h v1.10.3
ip-10-0-4-74.ec2.internal Ready,SchedulingDisabled  <none> 7h v1.10.3
```

5. Remove old nodes from the cluster and update `ConfigMap` again:

```
$ kubectl delete node ip-10-0-2-218.ec2.internal
node "ip-10-0-2-218.ec2.internal" deleted

$ kubectl delete node ip-10-0-4-74.ec2.internal
node "ip-10-0-4-74.ec2.internal" deleted

$ kubectl edit configmap aws-auth -n kube-system
configmap "aws-auth" edited

$ kubectl get nodes
NAME  STATUS ROLES  AGE  VERSION
ip-10-0-2-122.ec2.internal  Ready  <none> 15m  v1.11.5
```

Upgrading a Kubernetes version is an annoying topic for Kubernetes administrators. This is because of Kubernetes' release cycle (which usually occurs every three months) and the need to carry out enough compatibility testing.

The EKS upgrade procedure requires AWS knowledge and understanding. This consists of many steps and involves some technical difficulty, but it should not be too difficult. Because EKS is still a newer service in AWS, it'll keep improving and providing easier options to the user in the future.

Summary

In this chapter, we discussed the public cloud. AWS is the most popular public cloud service and it gives APIs the ability to control AWS infrastructure programmatically.

In addition, AWS EKS makes it easy to deploy Kubernetes on AWS. Furthermore, the control plane manages the master and `etcd` with high availability design that offloads huge management efforts.

On the other hand, you need to be aware of AWS basics such as availability zone awareness between pod (EC2) and persistent volume (EBS). In addition, you need intermediate AWS knowledge such as IAM credentials to gain access to an API server and use a worker node Instance Role ARN to register the cluster.

In addition, using ALB as ingress controller is available as of December 2018 (`https://aws.amazon.com/blogs/opensource/kubernetes-ingress-aws-alb-ingress-controller/`), but it also requires additional effort to configure this.

Although AWS keeps improving functionality, open source tools such as `eksctl` indicate EKS still requires more improvement for easier use of Kubernetes.

In `Chapter 11`, *Kubernetes on GCP*, we'll introduce Google Cloud Platform and Kubernetes Engine, which is a pioneer for making a hosted Kubernetes service on Cloud. This is more mature than AWS EKS.

Kubernetes on GCP

Google Cloud Platform (**GCP**) is becoming popular in the public cloud industry that's provided by Google. GCP has concepts that are similar to those provided by AWS, such as VPC, Compute Engine, Persistent Disk, Load Balancing, and several managed services. In this chapter, you'll learn about GCP and how to set up Kubernetes on GCP through the following topics:

- Understanding GCP
- Using and understanding GCP components
- Using **Google Kubernetes Engine** (**GKE**), the hosted Kubernetes service

Introduction to GCP

GCP was officially launched in 2011. Unlike AWS, GCP initially provided **PaaS** (**Platform as a Service**). Consequently, you can deploy your application directly instead of launching a VM. After that, GCP added some services and functionalities.

The most important service for Kubernetes users is GKE, which is a hosted Kubernetes service. So, you can get some relief from Kubernetes installation, upgrades, and management. This has a pay–as–you–go style approach to using the Kubernetes cluster. GKE is also a very active service that keeps providing new versions of Kubernetes in a timely manner and keeps coming up with new features and management tools for Kubernetes as well.

Let's take a look at what kind of foundation and services GCP provides and then we'll explore GKE.

$ gcloud init

VPC

VPC in GCP is quite a different policy compared with AWS. First of all, you don't need to set the CIDR prefix to VPC. In other words, you can't set CIDR to VPC. Instead, you just add one or some subnets to the VPC. Because you have to set certain CIDR blocks with a subnet, GCP VPC is therefore identified as a logical group of subnets, and subnets within VPC can communicate with each other.

Note that GCP VPC has two subnet modes, either `auto` or `custom`. If you choose `auto`, it will create some subnets on each region with predefined CIDR blocks. For example, type the following command: **$ gcloud compute networks create my-auto-network --subnet-mode auto**

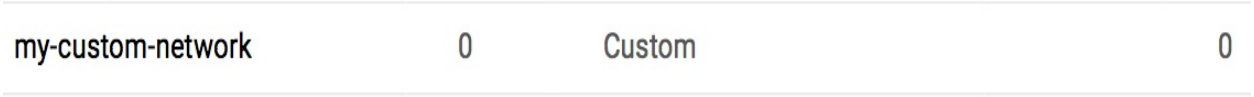
This will create 18 subnets as shown in the following screenshot (because, as of December 2018, GCP has 18 regions):

my-auto-network	18	Auto		
us-central1	my-auto-network		10.128.0.0/20	10.128.0.1
europe-west1	my-auto-network		10.132.0.0/20	10.132.0.1
us-west1	my-auto-network		10.138.0.0/20	10.138.0.1
asia-east1	my-auto-network		10.140.0.0/20	10.140.0.1
us-east1	my-auto-network		10.142.0.0/20	10.142.0.1
asia-northeast1	my-auto-network		10.146.0.0/20	10.146.0.1
asia-southeast1	my-auto-network		10.148.0.0/20	10.148.0.1
us-east4	my-auto-network		10.150.0.0/20	10.150.0.1
australia-southeast1	my-auto-network		10.152.0.0/20	10.152.0.1
europe-west2	my-auto-network		10.154.0.0/20	10.154.0.1
europe-west3	my-auto-network		10.156.0.0/20	10.156.0.1
southamerica-east1	my-auto-network		10.158.0.0/20	10.158.0.1
asia-south1	my-auto-network		10.160.0.0/20	10.160.0.1
northamerica-northeast1	my-auto-network		10.162.0.0/20	10.162.0.1
europe-west4	my-auto-network		10.164.0.0/20	10.164.0.1
europe-north1	my-auto-network		10.166.0.0/20	10.166.0.1
us-west2	my-auto-network		10.168.0.0/20	10.168.0.1
asia-east2	my-auto-network		10.170.0.0/20	10.170.0.1

Auto mode VPC is probably good to start with. However, in auto mode, you can't specify the CIDR prefix and 18 subnets from all regions might not fit your use case. For example, connect to your on–premise data center via VPN. Another example is creating subnets on a specific region only.

In this case, choose custom mode VPC, then you can create subnets with the desired CIDR prefix manually. Type the following command to create a custom mode VPC: **//create custom mode VPC which is named my-custom-network**
$ gcloud compute networks create my-custom-network --subnet-mode custom

Because custom mode VPC won't create any subnets, as shown in the following screenshot, let's add subnets onto this custom mode VPC:



Subnets

In GCP, subnets are always found across multiple zones (availability zones) within a region. In other words, you can't create subnets on a single zone like AWS. You always need to specify entire regions when creating a subnet.

In addition, unlike AWS, there are no significant concepts of public and private subnets (in AWS, a public subnet has a default route as IGW; on the other hand, a private subnet has a default route as the NAT gateway). This is because all subnets in GCP have a route to an internet gateway.

Instead of subnet-level access control, GCP uses host (instance)-level access control using **network tags** to ensure network security. This will be described in more detail in the following section.

It might make network administrators nervous; however, GCP best practice gives you a much more simplified and scalable VPC administration because you can add subnets at any time to expand entire network blocks.

Technically, you can launch a VM instance and set it up as a NAT gateway or HTTP proxy, and then create a custom priority route for the private subnet that points to the NAT/proxy instance to achieve an AWS–like private subnet.
Please refer to the following online document for details: `https://cloud.google.com/compute/docs/vpc/special-configurations`.

One more thing: an interesting and unique concept of GCP VPC is that you can add different CIDR prefix network blocks to a single VPC. For example, if you have custom mode VPC, then add the following three subnets:

- `subnet-a` (`10.0.1.0/24`) from `us-west1`
- `subnet-b` (`172.16.1.0/24`) from `us-east1`
- `subnet-c` (`192.168.1.0/24`) from `asia-northeast1`

The following commands will create three subnets from three different regions with different CIDR prefixes: **$ gcloud compute networks subnets create subnet-a --network=my-custom-network --range=10.0.1.0/24 --region=us-west1**
$ gcloud compute networks subnets create subnet-b --network=my-custom-

network --range=172.16.1.0/24 --region=us-east1
$ gcloud compute networks subnets create subnet-c --network=my-custom-network --range=192.168.1.0/24 --region=asia-northeast1

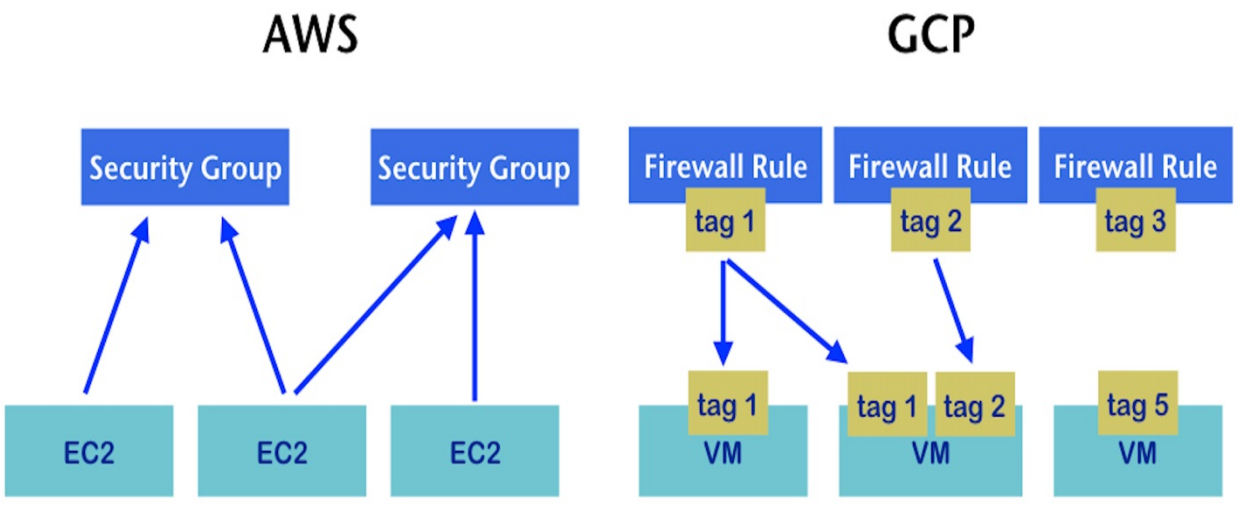
The result will be the following web console. If you're familiar with AWS VPC, you won't believe these combinations of CIDR prefixes are available within a single VPC! This means that, whenever you need to expand a network, you can assign another CIDR prefix to the VPC:

my-custom-network	3	Custom		0
us-west1	subnet-a		10.0.1.0/24	10.0.1.1
us-east1	subnet-b		172.16.1.0/24	172.16.1.1
asia-northeast1	subnet-c		192.168.1.0/24	192.168.1.1

Firewall rules

As previously mentioned, the GCP firewall rule is important for achieving network security. However, the GCP firewall is more simple and flexible than an AWS **Security Group** (**SG**). For example, in AWS, when you launch an EC2 instance, you have to assign at least one SG that is tightly coupled with EC2 and SG. On the other hand, in GCP, you can't assign any firewall rules directly. Instead, firewall rule and VM instance are loosely coupled via a **network tag**. Consequently, there's no direct association between the firewall rule and VM instance.

The following diagram is a comparison between AWS security groups and GCP firewall rules. EC2 requires a security group, while the GCP VM instance just sets a tag. This is irrespective of whether the corresponding firewall has the same tag or not:



For example, create a firewall rule for a public host (use the `public` network tag) and a private host (use the `private` network tag), as given in the following command:

```
//create ssh access for public host
$ gcloud compute firewall-rules create public-ssh --network=my-custom-network --allow="t

//create http access (80/tcp for public host)
$ gcloud compute firewall-rules create public-http --network=my-custom-network --allow="
```

```
//create ssh access for private host (allow from host which has "public" tag)
$ gcloud compute firewall-rules create private-ssh --network=my-custom-network --allow="

//create icmp access for internal each other (allow from host which has either "public"
$ gcloud compute firewall-rules create internal-icmp --network=my-custom-network --allow
```

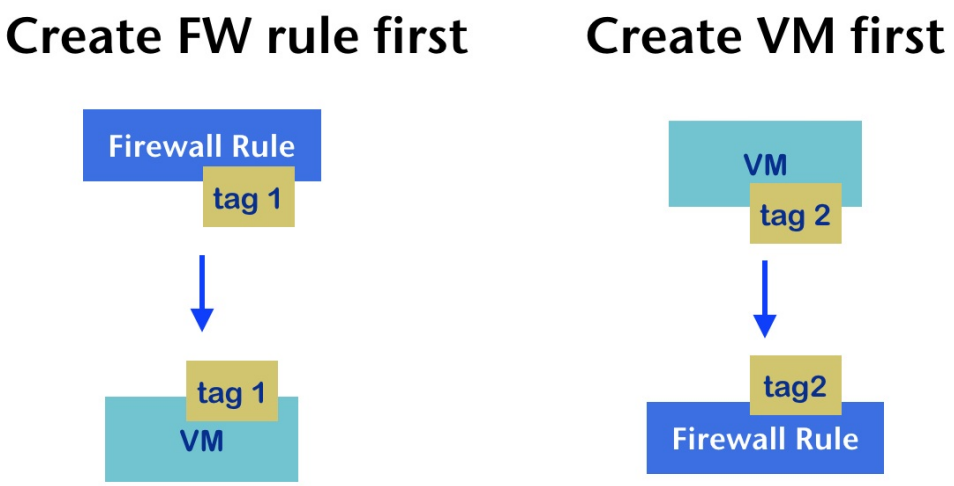
This creates four firewall rules as shown in the following screenshot. Let's create VM instances to use either the `public` or `private` network tag to see how it works:

Name	Targets	Source filters	Protocols / ports	Action	Priority	Network
internal-icmp	public, 1 more	Tags: public, 1 more	icmp	Allow	1000	my-custom-network
private-ssh	private	Tags: public	tcp:22	Allow	1000	my-custom-network
public-http	public	IP ranges: 0.0.0.0/0	tcp:80	Allow	1000	my-custom-network
public-ssh	public	IP ranges: 0.0.0.0/0	tcp:22	Allow	1000	my-custom-network

VM instances

In GCP, a VM instance is quite similar to AWS EC2. You can choose from a variety of machine (instance) types that have different hardware configurations; you can also choose a Linux-or Windows-based OS or your customized OS.

As mentioned when talking about firewall rules, you can specify any number of network tags. A tag doesn't necessarily need to be created beforehand. This means you can launch VM instances with network tags first, even though a firewall rule isn't created. It's still valid, but no firewall rule is applied in this case. Then you can create a firewall rule with a network tag. Eventually a firewall rule will be applied to the VM instances afterward. This is why VM instances and firewall rules are loosely coupled, which provides flexibility to the user:



Before launching a VM instance, you need to create an `ssh` public key first in the same way as AWS EC2. The easiest way to do this is to run the following command to create and register a new key: **//this command create new ssh key pair**
$ gcloud compute config-ssh

//key will be stored as ~/.ssh/google_compute_engine(.pub)
$ cd ~/.ssh
$ ls -l google_compute_engine*

-rw------- 1 saito admin 1766 Aug 23 22:58 google_compute_engine
-rw-r--r-- 1 saito admin 417 Aug 23 22:58 google_compute_engine.pub

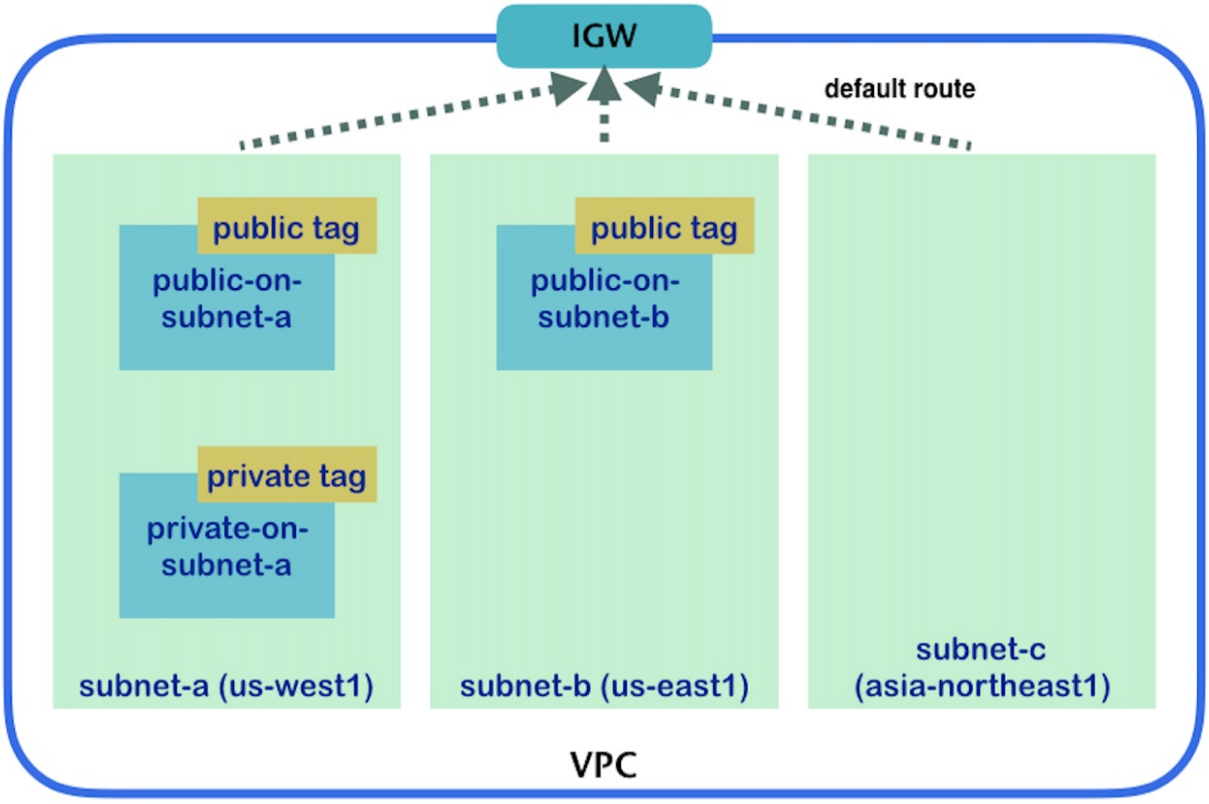
Now let's get started by launching a VM instance on GCP.

Deploy two instances on both `subnet-a` and `subnet-b` as public instances (use the `public` network tag) and then launch another instance on the `subnet-a` as a private instance (with a `private` network tag): **//create public instance ("public" tag) on subnet-a**
$ gcloud compute instances create public-on-subnet-a --machine-type=f1-micro --network=my-custom-network --subnet=subnet-a --zone=us-west1-a --tags=public

//create public instance ("public" tag) on subnet-b
$ gcloud compute instances create public-on-subnet-b --machine-type=f1-micro --network=my-custom-network --subnet=subnet-b --zone=us-east1-c --tags=public

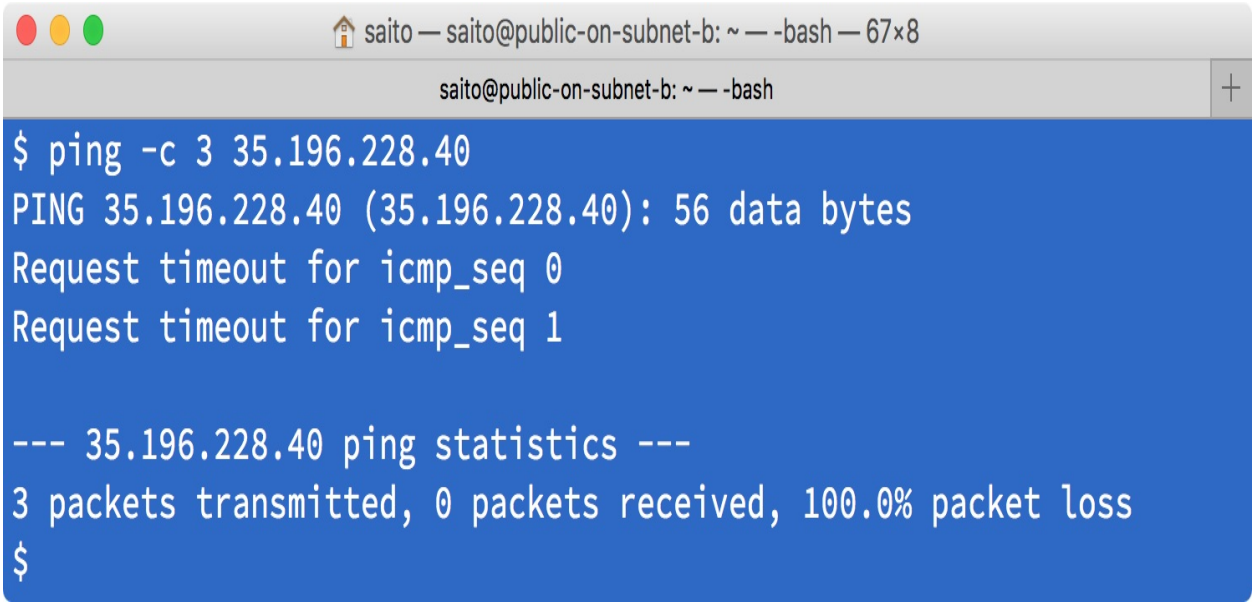
//create private instance ("private" tag) on subnet-a with larger size (g1-small)
$ gcloud compute instances create private-on-subnet-a --machine-type=g1-small --network=my-custom-network --subnet=subnet-a --zone=us-west1-a --tags=private

//Overall, there are 3 VM instances has been created in this example as below
$ gcloud compute instances list
NAME ZONE MACHINE_TYPE PREEMPTIBLE INTERNAL_IP EXTERNAL_IP STATUS
public-on-subnet-b us-east1-c f1-micro 172.16.1.2 35.196.228.40 RUNNING
private-on-subnet-a us-west1-a g1-small 10.0.1.2 104.199.121.234 RUNNING
public-on-subnet-a us-west1-a f1-micro 10.0.1.3 35.199.171.31 RUNNING

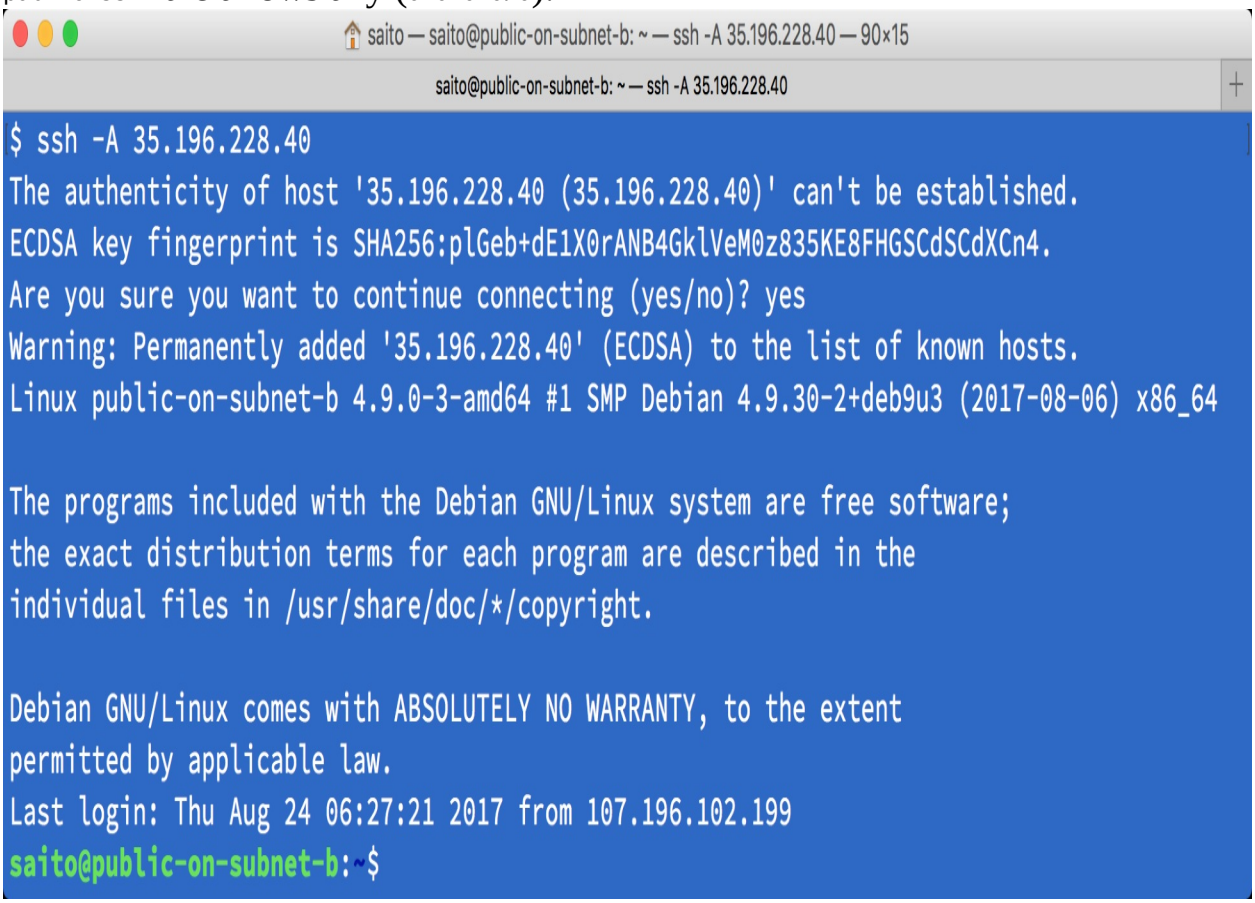


You can log in to those machines to check whether a firewall rule works as expected. First of all, you need to add an `ssh` key to `ssh-agent` on your machine: **$ ssh-add ~/.ssh/google_compute_engine**
Enter passphrase for /Users/saito/.ssh/google_compute_engine:
Identity added: /Users/saito/.ssh/google_compute_engine (/Users/saito/.ssh/google_compute_engine)

Then, check whether an ICMP firewall rule can reject traffic from external because ICMP only allows public or private tagged hosts, so the ping packet from your machine won't reach the public instance; this is shown in the following screenshot:



On the other hand, the public `host` allows `ssh` from your machine, because the `public-ssh` rule allows any (`0.0.0.0/0`):



Of course, this host can ping and `ssh` to private hosts on `subnet-a` (`10.0.1.2`) through

a private IP address, because of the `internal-icmp` and `private-ssh` rules.

Let's `ssh` to a private host and then install `tomcat8` and the `tomcat8-examples` package (this will install the `/examples/` application for Tomcat):

Remember that `subnet-a` is a `10.0.1.0/24` CIDR prefix, but `subnet-b` is a `172.16.1.0/24` CIDR prefix. However, within the same VPC, there's connectivity with each other. This is the great benefit of using GCP as you can expand a network address block whenever this is required.

Now install `nginx` to public hosts (`public-on-subnet-a` and `public-on-subnet-b`): **//logout from VM instance, then back to your machine**

$ exit

//install nginx from your machine via ssh
$ ssh 35.196.228.40 "sudo apt-get -y install nginx"
$ ssh 35.199.171.31 "sudo apt-get -y install nginx"

//check whether firewall rule (public-http) work or not
$ curl -I http://35.196.228.40/
HTTP/1.1 200 OK
Server: nginx/1.10.3
Date: Sun, 27 Aug 2017 07:07:01 GMT
Content-Type: text/html
Content-Length: 612
Last-Modified: Fri, 25 Aug 2017 05:48:28 GMT
Connection: keep-alive
ETag: "599fba2c-264"
Accept-Ranges: bytes

However, at this moment, you can't access Tomcat on a private host even if it has a public IP address. This is because a private host doesn't yet have any firewall rule that allows `8080/tcp`: **$ curl http://104.199.121.234:8080/examples/**
curl: (7) Failed to connect to 104.199.121.234 port 8080: Operation timed out

Rather than simply creating a firewall rule for Tomcat, we'll set up a LoadBalancer to be configured for both nginx and Tomcat in the next section.

Load balancing

GCP provides several types of load balancer as follows:

- Layer 4 TCP LoadBalancer
- Layer 4 UDP LoadBalancer
- Layer 7 HTTP(S) LoadBalancer

The Layer 4 LoadBalancers (both TCP and UDP) are similar to AWS Classic ELB. On the other hand, the Layer 7 HTTP(S) LoadBalancer has content (context)-based routing. For example, the URL/image will forward to `instance-a`; everything else will forward to `instance-b`. So, it's more like an application-level LoadBalancer.

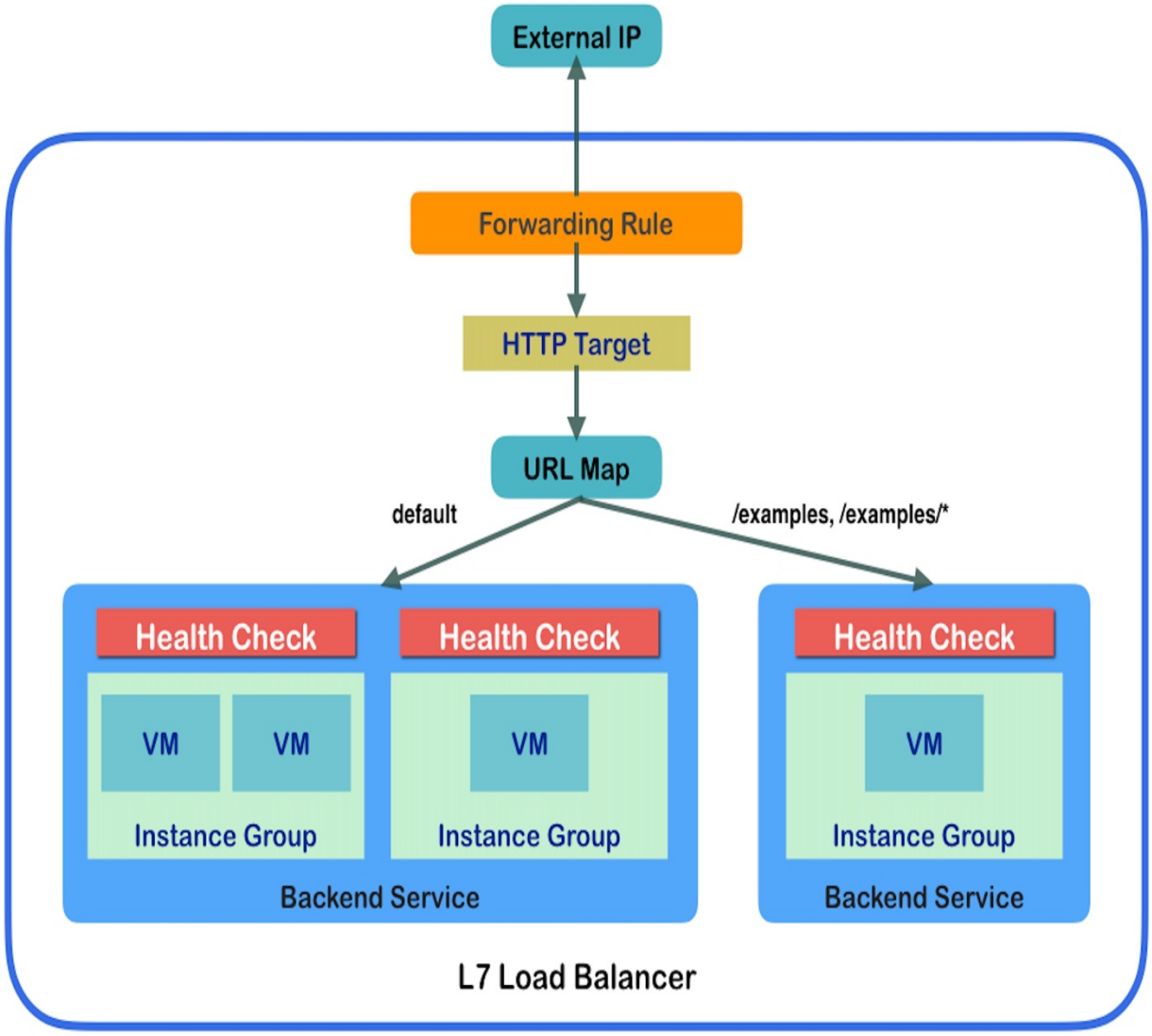
*AWS also provides **Application Load Balancer** (**ALB** or **ELBv2**), which is quite similar to the GCP Layer 7 HTTP(S) LoadBalancer. For details, please visit* https://aws.amazon.com/blogs/aws/new-aws-application-load-balancer/.

In order to set up LoadBalancer, and unlike AWS ELB, there are several steps you'll need to follow in order to configure some items beforehand:

Configuration item	**Purpose**
Instance group	Determine a group of VM instances or a VM template (OS image).
Health check	Set health threshold (interval, timeout, and so on) to determine instance group health status.
Backend service	Set load threshold (maximum CPU or requests per second) and session affinity (sticky session) to the instance group

	and associate it to health check.
`url-maps` (LoadBalancer)	This is an actual place-holder to represent an L7 LoadBalancer that associates backend services and targets the HTTP(S) proxy.
Target HTTP(S) proxy	This is a connector that makes relationships between frontend forwarding rules to LoadBalancer.
Frontend forwarding rule	Associate between the Target HTTP proxy and IP address (ephemeral or static) and port number.
External IP (static)	(Optional) allocate a static external IP address for LoadBalancer.

The following diagram is for all of the preceding components' association that constructs the L7 LoadBalancer:



Let's set up an instance group first. In this example, there are three instance groups to create: one for the private hosted Tomcat instance (`8080/tcp`) and another two instance groups for public HTTP instances for each zone.

To do that, execute the following commands to group three of them:

```
//create instance groups for HTTP instances and tomcat instance
$ gcloud compute instance-groups unmanaged create http-ig-us-west --zone us-west1-a
$ gcloud compute instance-groups unmanaged create http-ig-us-east --zone us-east1-c
$ gcloud compute instance-groups unmanaged create tomcat-ig-us-west --zone us-west1-a

//because tomcat uses 8080/tcp, create a new named port as tomcat:8080
$ gcloud compute instance-groups unmanaged set-named-ports tomcat-ig-us-west --zone us-w

//register an existing VM instance to correspond instance group
$ gcloud compute instance-groups unmanaged add-instances http-ig-us-west --instances pub
```

```
$ gcloud compute instance-groups unmanaged add-instances http-ig-us-east --instances pub
$ gcloud compute instance-groups unmanaged add-instances tomcat-ig-us-west --instances p
```

//create health check for http (80/tcp) for "/"
$ gcloud compute health-checks create http my-http-health-check --check-interval 5 --healthy-threshold 2 --unhealthy-threshold 3 --timeout 5 --port 80 --request-path /

 //create health check for Tomcat (8080/tcp) for "/examples/"
$ gcloud compute health-checks create http my-tomcat-health-check --check-interval 5 --healthy-threshold 2 --unhealthy-threshold 3 --timeout 5 --port 8080 --request-path /examples/

//create backend service for http (default) and named port tomcat (8080/tcp)
$ gcloud compute backend-services create my-http-backend-service --health-checks my-http-health-check --protocol HTTP --global
$ gcloud compute backend-services create my-tomcat-backend-service --health-checks my-tomcat-health-check --protocol HTTP --port-name tomcat --global

//add http instance groups (both us-west1 and us-east1) to http backend service
$ gcloud compute backend-services add-backend my-http-backend-service --instance-group http-ig-us-west --instance-group-zone us-west1-a --balancing-mode UTILIZATION --max-utilization 0.8 --capacity-scaler 1 --global
$ gcloud compute backend-services add-backend my-http-backend-service --instance-group http-ig-us-east --instance-group-zone us-east1-c --balancing-mode UTILIZATION --max-utilization 0.8 --capacity-scaler 1 --global

//also add tomcat instance group to tomcat backend service
$ gcloud compute backend-services add-backend my-tomcat-backend-service --instance-group tomcat-ig-us-west --instance-group-zone us-west1-a --balancing-mode UTILIZATION --max-utilization 0.8 --capacity-scaler 1 --global

Creating a LoadBalancer

The LoadBalancer needs to bind both `my-http-backend-service` and `my-tomcat-backend-service`. In this scenario, only `/examples` and `/examples/*` will be traffic forwarded to `my-tomcat-backend-service`. Other than that, every URI forwards traffic to `my-http-backend-service`: **//create load balancer(url-map) to associate my-http-backend-service as default**
$ gcloud compute url-maps create my-loadbalancer --default-service my-http-backend-service

//add /examples and /examples/* mapping to my-tomcat-backend-service
$ gcloud compute url-maps add-path-matcher my-loadbalancer --default-service my-http-backend-service --path-matcher-name tomcat-map --path-rules /examples=my-tomcat-backend-service,/examples/*=my-tomcat-backend-service

//create target-http-proxy that associate to load balancer(url-map) $ gcloud compute target-http-proxies create my-target-http-proxy --url-map=my-loadbalancer

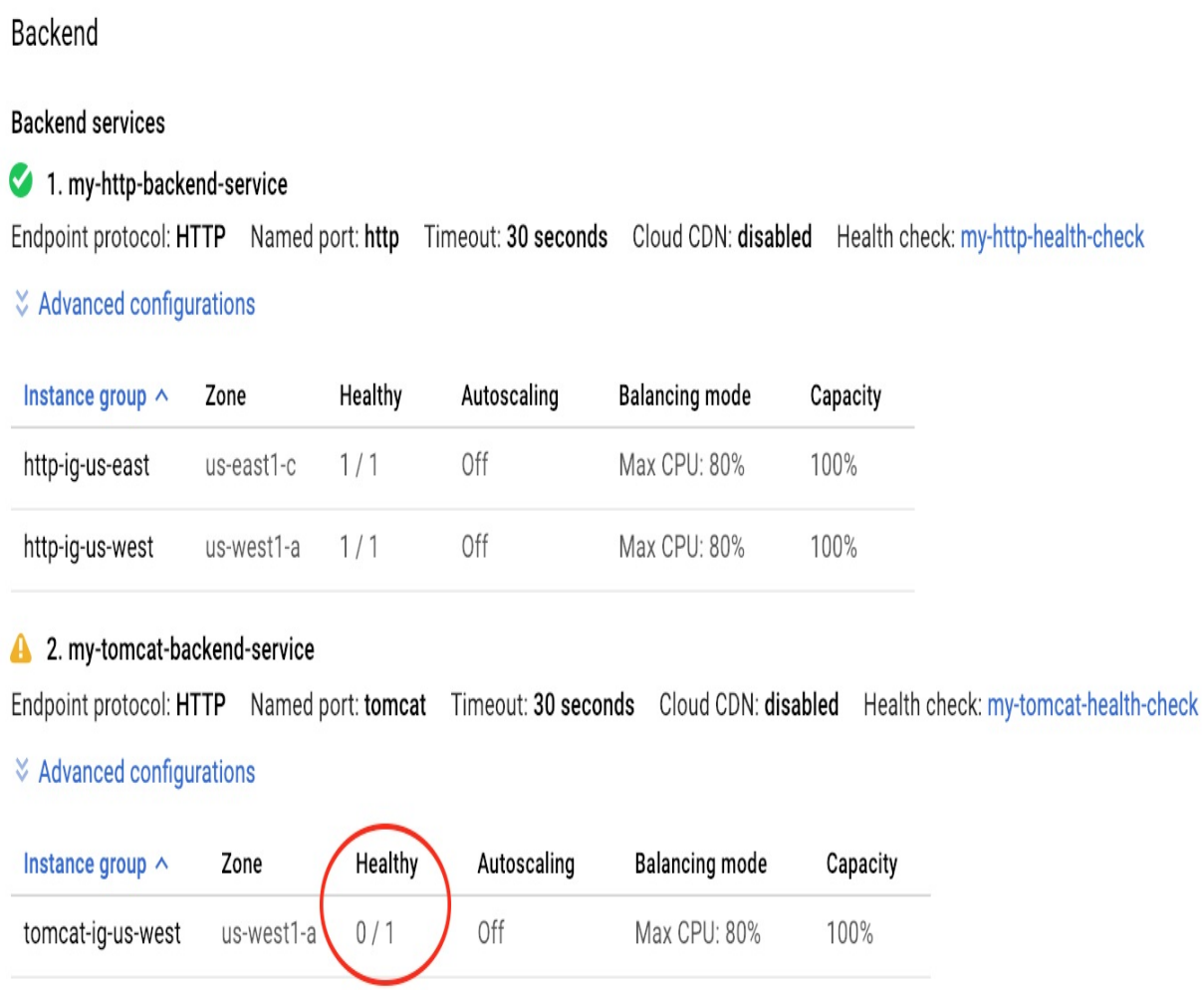
//allocate static global ip address and check assigned address
$ gcloud compute addresses create my-loadbalancer-ip --global
$ gcloud compute addresses describe my-loadbalancer-ip --global
address: 35.186.192.6

creationTimestamp: '2018-12-08T13:40:16.661-08:00'

...
...
//create forwarding rule that associate static IP to target-http-proxy
$ gcloud compute forwarding-rules create my-frontend-rule --global --target-http-proxy my-target-http-proxy --address 35.186.192.6 --ports 80

If you don't specify an `--address` option, ephemeral external IP address will be created and assigned.

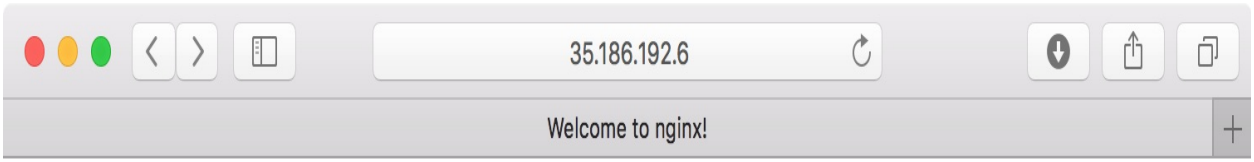
Finally, the LoadBalancer has been created. However, one missing configuration remains. Private hosts don't have any firewall rules to allow Tomcat traffic (`8080/tcp`). This is why, when you see the LoadBalancer status, a healthy status of `my-tomcat-backend-service` is kept down (`0`):



In this case, you need to add one more firewall rule that allows connection from LoadBalancer to a private subnet (use the `private` network tag for this). According to the GCP documentation (`https://cloud.google.com/compute/docs/load-balancing/health-checks#https_ssl_proxy_tcp_proxy_and_internal_load_balancing`), the health check heart-beat will come from the address range `35.191.0.0/16` to `130.211.0.0/22`:

```
//add one more Firewall Rule that allow Load Balancer to Tomcat (8080/tcp)

$ gcloud compute firewall-rules create private-tomcat --network=my-custom-network --sour
```

After a few minutes, the `my-tomcat-backend-service` healthy status will be up to (`1`); now you can access the LoadBalancer from a web browser. When accessing `/`, it should route to `my-http-backend-service`, which has the `nginx` application on public hosts:



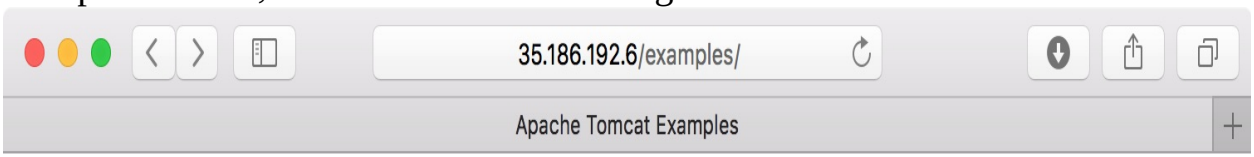
Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

On the other hand, if you access the `/examples/` URL with the same LoadBalancer IP address, it will route to `my-tomcat-backend-service`, which is a Tomcat application on a private host, as shown in the following screenshot:



Apache Tomcat Examples

- Servlets examples
- JSP Examples
- WebSocket Examples

Overall, there are some steps that need to be performed in order to set up a LoadBalancer, but it's useful to integrate different HTTP applications onto a single LoadBalancer to deliver your service efficiently with minimal resources.

Persistent Disk

GCE also has a storage service called **Persistent Disk** (**PD**) that's quite similar to AWS EBS. You can allocate the desired size and types (either standard or SSD) on each zone and attach/detach VM instances anytime.

Let's create one PD and then attach it to the VM instance. Note that, when attaching a PD to the VM instance, both must be in the same zones. This limitation is the same as AWS EBS. So, before creating PD, check the VM instance location once again: **$ gcloud compute instances list**
NAME ZONE MACHINE_TYPE PREEMPTIBLE INTERNAL_IP EXTERNAL_IP STATUS
public-on-subnet-b us-east1-c f1-micro 172.16.1.2 35.196.228.40 RUNNING
private-on-subnet-a us-west1-a g1-small 10.0.1.2 104.199.121.234 RUNNING
public-on-subnet-a us-west1-a f1-micro 10.0.1.3 35.199.171.31 RUNNING

Let's choose `us-west1-a` and then attach it to `public-on-subnet-a`: **//create 20GB PD on us-west1-a with standard type**
$ gcloud compute disks create my-disk-us-west1-a --zone us-west1-a --type pd-standard --size 20

//after a few seconds, check status, you can see existing boot disks as well
$ gcloud compute disks list
NAME ZONE SIZE_GB TYPE STATUS
public-on-subnet-b us-east1-c 10 pd-standard READY
my-disk-us-west1-a us-west1-a 20 pd-standard READY
private-on-subnet-a us-west1-a 10 pd-standard READY
public-on-subnet-a us-west1-a 10 pd-standard READY

//attach PD(my-disk-us-west1-a) to the VM instance(public-on-subnet-a)
$ gcloud compute instances attach-disk public-on-subnet-a --disk my-disk-us-west1-a --zone us-west1-a

//login to public-on-subnet-a to see the status
$ ssh 35.199.171.31
Linux public-on-subnet-a 4.9.0-3-amd64 #1 SMP Debian 4.9.30-2+deb9u3 (2017-08-06) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the individual files in /usr/share/doc/*/copyright.
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Fri Aug 25 03:53:24 2017 from 107.196.102.199
saito@public-on-subnet-a:~$ sudo su
root@public-on-subnet-a:/home/saito# dmesg | tail
[7377.421190] systemd[1]: apt-daily-upgrade.timer: Adding 25min 4.773609s random time.
[7379.202172] systemd[1]: apt-daily-upgrade.timer: Adding 6min 37.770637s random time.
[243070.866384] scsi 0:0:2:0: Direct-Access Google PersistentDisk 1 PQ: 0 ANSI: 6
[243070.875665] sd 0:0:2:0: [sdb] 41943040 512-byte logical blocks: (21.5 GB/20.0 GiB)
[243070.883461] sd 0:0:2:0: [sdb] 4096-byte physical blocks
[243070.889914] sd 0:0:2:0: Attached scsi generic sg1 type 0
[243070.900603] sd 0:0:2:0: [sdb] Write Protect is off
[243070.905834] sd 0:0:2:0: [sdb] Mode Sense: 1f 00 00 08
[243070.905938] sd 0:0:2:0: [sdb] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
[243070.925713] sd 0:0:2:0: [sdb] Attached SCSI disk

You may see that the PD has been attached at `/dev/sdb`. Similar to AWS EBS, you have to format this disk. Because this is a Linux OS operation, the steps are exactly the same as described in `Chapter 10`, *Kubernetes on AWS*.

Google Kubernetes Engine (GKE)

Overall, some GCP have introduced in previous sections. Now you can start to set up Kubernetes on GCP VM instances using those components. You can even use open-source Kubernetes provisioning tools such as `kops` and `kubespray` too.

Google Cloud provides GKE On-Prem (`https://cloud.google.com/gke-on-prem/`), which allows the user to set up GKE on their own data center resources. As of January 2019, this is an alpha version and not open to everyone yet.

However, GCP has a managed Kubernetes service called GKE. Under the hood, this uses some GCP components such as VPC, VM instances, PD, firewall rules, and LoadBalancers.

Of course, as usual you can use the `kubectl` command to control your Kubernetes cluster on GKE, which includes the Cloud SDK. If you haven't installed the `kubectl` command on your machine yet, type the following command to install it via the Cloud SDK: **//install kubectl command**
$ gcloud components install kubectl

Setting up your first Kubernetes cluster on GKE

You can set up a Kubernetes cluster on GKE using the `gcloud` command. This needs to specify several parameters to determine some configurations. One important parameter is the network. Here, you have to specify which VPC and subnet you'll deploy. Although GKE supports multiple zones to deploy, you need to specify at least one zone for the Kubernetes master node. This time, it uses the following parameters to launch a GKE cluster:

Parameter	Description	Value
`--machine-type`	VM instance type for Kubernetes Node	`f1-micro`
`--num-nodes`	Initial number of Kubernetes nodes	`3`
`--network`	Specify GCP VPC	`my-custom-network`
`--subnetwork`	Specify GCP Subnet if VPC is a custom mode	`subnet-c`
`--zone`	Specify a single zone	`asia-northeast1-a`
`--tags`	Network tags that will be assigned to Kubernetes nodes	`private`

In this scenario, you need to type the following commands to launch a Kubernetes cluster on GCP. It may take a few minutes to complete because, behind the scenes, it'll launch several VM instances and set up the Kubernetes master and nodes. Note that the Kubernetes master and etcd will be fully managed by GCP. This means that the master node and etcd don't consume your VM instances: **$ gcloud container clusters create my-k8s-cluster --machine-type f1-micro --num-nodes 3 --network my-custom-network --subnetwork subnet-c --zone asia-northeast1-a --tags private**

//after a few minutes, check node status
NAME STATUS ROLES AGE VERSION
gke-my-k8s-cluster-default-pool-bcae4a66-mlhw Ready <none> 2m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-tn74 Ready <none> 2m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-w5l6 Ready <none> 2m v1.10.9-gke.5

Note that we specify the `--tags private` option so that a Kubernetes node VM instance has a network tag of `private`. Therefore, it behaves the same as other regular VM instances that have `private` tags. Consequently, you can't SSH from the public internet and you can't HTTP from the internet either. However, you can ping and SSH from another VM instance that has a `public` network tag.

Node pool

When launching the Kubernetes cluster, you can specify the number of nodes using the `--num-nodes` option. GKE manages a Kubernetes node as a node pool. This means you can manage one or more node pools that are attached to your Kubernetes cluster.

What if you need to add or delete nodes? GKE let's you resize the node pool by performing the following command to change Kubernetes node from 3 to 5:
//run resize command to change number of nodes to 5
$ gcloud container clusters resize my-k8s-cluster --size 5 --zone asia-northeast1-a

//after a few minutes later, you may see additional nodes $ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-my-k8s-cluster-default-pool-bcae4a66-j8zz Ready <none> 32s v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-jnnw Ready <none> 32s v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-mlhw Ready <none> 4m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-tn74 Ready <none> 4m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-w5l6 Ready <none> 4m v1.10.9-gke.5

Increasing the number of nodes will help if you need to scale out your node capacity. However, in this scenario, it still uses the smallest instance type (`f1-micro`, which has only 0.6 GB memory). It might not help if a single container needs more than 0.6 GB of memory. In this case you need to scale up, which means you need to add a larger VM instance type.

In this case, you have to add another set of node pools onto your cluster. This is because, within the same node pool, all VM instances are configured the same. Consequently, you can't change the instance type in the same node pool.

Add a new node pool that has two new sets of `g1-small` (1.7 GB memory) VM instance types to the cluster. Then you can expand Kubernetes nodes with a different hardware configuration.

By default, there are some quotas that can create a number limit for VM instances within one region (for example, up to eight CPU cores on `us-west1`). If you wish to increase this quota, you must change your account to a paid one and then request a quota change to GCP. For more details, please read the online documentation from `https://cloud.google.com/compute/quotas` and `https://cloud.google.com/free/docs/frequently-asked-questions#how-to-upgrade`.

Run the following command to add an additional node pool that has two instances of a `g1-small` instance: **//create and add node pool which is named "large-mem-pool"**
$ gcloud container node-pools create large-mem-pool --cluster my-k8s-cluster --machine-type g1-small --num-nodes 2 --tags private --zone asia-northeast1-a
//after a few minustes, large-mem-pool instances has been added
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-my-k8s-cluster-default-pool-bcae4a66-j8zz Ready <none> 5m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-jnnw Ready <none> 5m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-mlhw Ready <none> 9m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-tn74 Ready <none> 9m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-bcae4a66-w5l6 Ready <none> 9m v1.10.9-gke.5
gke-my-k8s-cluster-large-mem-pool-66e3a44a-jtdn Ready <none> 46s v1.10.9-gke.5
gke-my-k8s-cluster-large-mem-pool-66e3a44a-qpbr Ready <none> 44s v1.10.9-gke.5

Now you have a total of seven CPU cores and 6.4 GB memory in your cluster, which has more capacity. However, due to larger hardware types, the Kubernetes scheduler will probably assign a pod to `large-mem-pool` first because it has enough memory capacity.

However, you may want to preserve the `large-mem-pool` node in case a big application needs a large memory size (for example, a Java application). Therefore, you may want to differentiate `default-pool` and `large-mem-pool`.

In this case, the Kubernetes label, `beta.kubernetes.io/instance-type`, helps to distinguish an instance type of node. Therefore, use `nodeSelector` to specify a desired node for the pod. For example, the following `nodeSelector` parameter will force the use of the `f1-micro` node for the `nginx` application: **//nodeSelector specifies f1-micro**
$ cat nginx-pod-selector.yml
apiVersion: v1
kind: Pod
metadata:
name: nginx
spec:
containers:
- name: nginx
image: nginx
nodeSelector:
beta.kubernetes.io/instance-type: f1-micro

//deploy pod
$ kubectl create -f nginx-pod-selector.yml
pod "nginx" created

//it uses default pool
NAME READY STATUS RESTARTS AGE IP NODE
nginx 0/1 ContainerCreating 0 10s <none> gke-my-k8s-cluster-default-pool-bcae4a66-jnnw

If you want to specify a particular label instead of `beta.kubernetes.io/instance-type`, use the `--node-labels` option to create a node pool. That assigns your desired label to the node pool. For more details, please read the following online document: https://cloud.google.com/sdk/gcloud/reference/container/node-pools/create.

Of course, you can feel free to remove a node pool if you no longer need it. To do that, run the following command to delete `default-pool` (`f1-micro` x 5 instances). This operation will involve pod migration (terminate the pod on `default-pool` and relaunch it on `large-mem-pool`) automatically if there are some pods running at

`default-pool`: **//list Node Pool**
$ gcloud container node-pools list --cluster my-k8s-cluster --zone asia-northeast1-a
NAME MACHINE_TYPE DISK_SIZE_GB NODE_VERSION
default-pool f1-micro 100 1.10.9-gke.5
large-mem-pool g1-small 100 1.10.9-gke.5

//delete default-pool
$ gcloud container node-pools delete default-pool --cluster my-k8s-cluster --zone asia-northeast1-a

//after a few minutes, default-pool nodes x 5 has been deleted
$ kubectl get nodes
NAME STATUS ROLES AGE VERSION
gke-my-k8s-cluster-large-mem-pool-66e3a44a-jtdn Ready <none> 9m v1.10.9-gke.5
gke-my-k8s-cluster-large-mem-pool-66e3a44a-qpbr Ready <none> 9m v1.10.9-gke.5

You may have noticed that all of the preceding operations happened in a single zone (`asia-northeast1-a`). Therefore, if the `asia-northeast1-a` zone gets an outage, your cluster will be down. In order to avoid zone failure, you may consider setting up a multi-zone cluster.

Multi-zone clusters

GKE supports multi-zone clusters, which allows you to launch Kubernetes nodes on multiple zones, but within the same region. In previous examples, Kubernetes nodes have been provisioned at `asia-northeast1-a` only, so let's re-provision a cluster that has `asia-northeast1-a`, `asia-northeast1-b`, and `asia-northeast1-c` in a total of three zones.

It's very simple; you just use the `--node-locations` parameter to specify three zones when creating a new cluster.

Let's delete the previous cluster and create a new one with the `--node-locations` option: **//delete cluster first $ gcloud container clusters delete my-k8s-cluster --zone asia-northeast1-a**
The following clusters will be deleted.
- [my-k8s-cluster] in [asia-northeast1-a]
Do you want to continue (Y/n)? y
...
...

//create a new cluster with --node-locations option with 2 nodes per zones $ gcloud container clusters create my-k8s-cluster --machine-type f1-micro --num-nodes 2 --network my-custom-network --subnetwork subnet-c --tags private --zone asia-northeast1-a --node-locations asia-northeast1-a,asia-northeast1-b,asia-northeast1-c

This example will create two nodes per zone (`asia-northeast1-a`, `b`, and `c`). Consequently, a total of six nodes will be added: **$ kubectl get nodes**
NAME STATUS ROLES AGE VERSION
gke-my-k8s-cluster-default-pool-58e4e9a4-74hc Ready <none> 43s v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-58e4e9a4-8jft Ready <none> 1m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-f7baf2e1-lk7j Ready <none> 1m v1.10.9-gke.5

gke-my-k8s-cluster-default-pool-f7baf2e1-zktg Ready <none> 1m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-fa5a04a5-bbj5 Ready <none> 1m v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-fa5a04a5-d35z Ready <none> 1m v1.10.9-gke.5

You may also distinguish node zones with the `failure-domain.beta.kubernetes.io/zone` *Kubernetes label so that you can specify the desired zones to deploy a pod.*

Cluster upgrade

Once you start to manage Kubernetes, you may encounter difficulties when upgrading Kubernetes clusters. Due to the fact that the Kubernetes project is very aggressive, there's a new release around every three months, such as version 1.11.0 (released on June 27th 2018), 1.12.0 (released on September 27th 2018), and 1.13.0 (released on December 3rd 2018).

GKE also keeps adding new version support in a timely manner. This allows us to upgrade both master and nodes via the `gcloud` command. You can run the following command to see which Kubernetes version is supported by GKE: **$ gcloud container get-server-config**
Fetching server config for us-west1-b

defaultClusterVersion: 1.10.9-gke.5
defaultImageType: COS
validImageTypes:
- COS_CONTAINERD
- COS
- UBUNTU
validMasterVersions:
- 1.11.4-gke.8
- 1.11.3-gke.18
- 1.11.2-gke.20
- 1.11.2-gke.18
- 1.10.9-gke.7
- 1.10.9-gke.5
- 1.10.7-gke.13
- 1.10.7-gke.11
- 1.10.6-gke.13
- 1.10.6-gke.11
- 1.9.7-gke.11
validNodeVersions:
- 1.11.4-gke.8
- 1.11.3-gke.18

- 1.11.2-gke.20
- 1.11.2-gke.18
- 1.11.2-gke.15
- 1.11.2-gke.9
- 1.10.9-gke.7
- 1.10.9-gke.5
- 1.10.9-gke.3
- 1.10.9-gke.0
- 1.10.7-gke.13
- 1.10.7-gke.11
...

So, you can see that the latest supported Kubernetes version is `1.11.4-gke.8` on both master and node at the time of writing. Since the previous example installed is version `1.10.9-gke.5`, let's update this to `1.11.4-gke.8`. First of all, you need to upgrade `master` first: **//upgrade master using --master option**
$ gcloud container clusters upgrade my-k8s-cluster --zone asia-northeast1-a --cluster-version 1.11.4-gke.8 --master

Master of cluster [my-k8s-cluster] will be upgraded from version [1.10.9-gke.5] to version [1.11.4-gke.8]. This operation is long-running and will block other operations on the cluster (including delete) until it has run to completion.

Do you want to continue (Y/n)? y

Upgrading my-k8s-cluster...done.
Updated [https://container.googleapis.com/v1/projects/devops-with-kubernetes/zones/asia-northeast1-a/clusters/my-k8s-cluster].

This takes around 10 minutes depending on the environment. After that, you can verify `MASTER_VERSION` via the following command: **//master upgrade has been successfully to done**
$ gcloud container clusters list --zone asia-northeast1-a
NAME LOCATION MASTER_VERSION MASTER_IP MACHINE_TYPE NODE_VERSION NUM_NODES STATUS

my-k8s-cluster asia-northeast1-a 1.11.4-gke.8 35.243.78.166 f1-micro 1.10.9-gke.5 * 6 RUNNING

Now you can start to upgrade all nodes to version `1.11.4-gke.8`. Because GKE tries to perform a rolling upgrade, it'll perform the following steps on each node, one by one:

1. De-register a target node from the cluster
2. Delete old VM instances
3. Provision a new VM instance

4. Set up the node with the `1.11.4-gke.8` version
5. Register to `master`

Therefore, a node upgrade takes much longer than a master upgrade: **//node upgrade (not specify --master)**
$ gcloud container clusters upgrade my-k8s-cluster --zone asia-northeast1-a --cluster-version 1.11.4-gke.8
All nodes (6 nodes) of cluster [my-k8s-cluster] will be upgraded from version [1.10.9-gke.5] to version [1.11.4-gke.8]. This operation is long-running and will block other operations on the cluster (including delete) until it has run to completion.

Do you want to continue (Y/n)? y

During the rolling upgrade, you can see the node status as follows; this example shows the halfway point of a rolling update. Here, two nodes have upgraded to `1.11.4-gke.8`, one node is about to upgrade, and the remaining three nodes are pending:

```
$ kubectl get nodes
NAME STATUS  ROLES  AGE  VERSION
gke-my-k8s-cluster-default-pool-58e4e9a4-74hc  Ready <none> 18m  v1.11.4-gke.8
gke-my-k8s-cluster-default-pool-58e4e9a4-8jft  Ready <none> 19m  v1.11.4-gke.8
gke-my-k8s-cluster-default-pool-f7baf2e1-lk7j  Ready <none> 19m  v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-f7baf2e1-zktg  Ready <none> 19m  v1.10.9-gke.5
gke-my-k8s-cluster-default-pool-fa5a04a5-bbj5  Ready,SchedulingDisabled  <none> 19m  v1.
gke-my-k8s-cluster-default-pool-fa5a04a5-d35z  Ready <none> 19m  v1.10.9-gke.5
```

Kubernetes cloud provider

GKE also integrates out-of-the-box Kubernetes cloud providers, which deeply integrate with the GCP infrastructure, for example, the overlay network by VPC route, `StorageClass` by Persistent Disk, and Service by L4 LoadBalancer. One of the best integrations is provided by the ingress controller by L7 LoadBalancer. Let's take a look at how this works.

$ kubectl get storageclass
NAME TYPE
standard (default) kubernetes.io/gce-pd

$ kubectl describe storageclass standard

Name: standard
IsDefaultClass: Yes
Annotations: storageclass.beta.kubernetes.io/is-default-class=true
Provisioner: kubernetes.io/gce-pd
Parameters: type=pd-standard
AllowVolumeExpansion: <unset>
MountOptions: <none>
ReclaimPolicy:

Delete
VolumeBindingMode: Immediate
Events: <none>

$ cat pvc-gke.yml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: pvc-gke-1
spec:
 storageClassName: "standard"
 accessModes:
 -ReadWriteOnce
 resources:
 requests:
 <span

xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.50.1">storage: 10Gi

  //create Persistent Volume Claim $ kubectl create -f pvc-gke.yml persistentvolumeclaim "pvc-gke-1" created
 //check Persistent Volume $ kubectl get pv

NAME CAPACITY ACCESSMODES RECLAIMPOLICY STATUS CLAIM STORAGECLASS REASON AGE

pvc-bc04e717-8c82-11e7-968d-42010a920fc3 10Gi RWO Delete Bound default/pvc-gke-1 standard 2s
 //check via gcloud command $ gcloud compute disks list

NAME ZONE SIZE_GB TYPE STATUS

gke-my-k8s-cluster-d2e-pvc-bc04e717-8c82-11e7-968d-42010a920fc3 asia-northeast1-a 10 pd-standard READY

$ cat grafana.yml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: grafana
spec:
 replicas: 1
 selector:
 matchLabels:
 project: devops-with-kubernetes
 app: grafana
 template:
 metadata:
 labels:
 <span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.22.1">project: devops-with-kubernetes
 app: grafana
 spec:
 containers:
 - image: grafana/grafana
 name: grafana
 ports:
 - containerPort: 3000

```
---

apiVersion: v1

kind: Service

metadata:

  name: grafana

spec:

  ports:

  - port: 80

  targetPort: 3000
```

type: LoadBalancer

selector:
 project: devops-with-kubernetes
 app: grafana

//deploy grafana with Load Balancer service

$ kubectl create -f grafana.yml

deployment.apps "grafana" created
service "grafana" created

//check L4 Load balancer IP address

$ kubectl get svc grafana
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S)AGE
grafana LoadBalancer

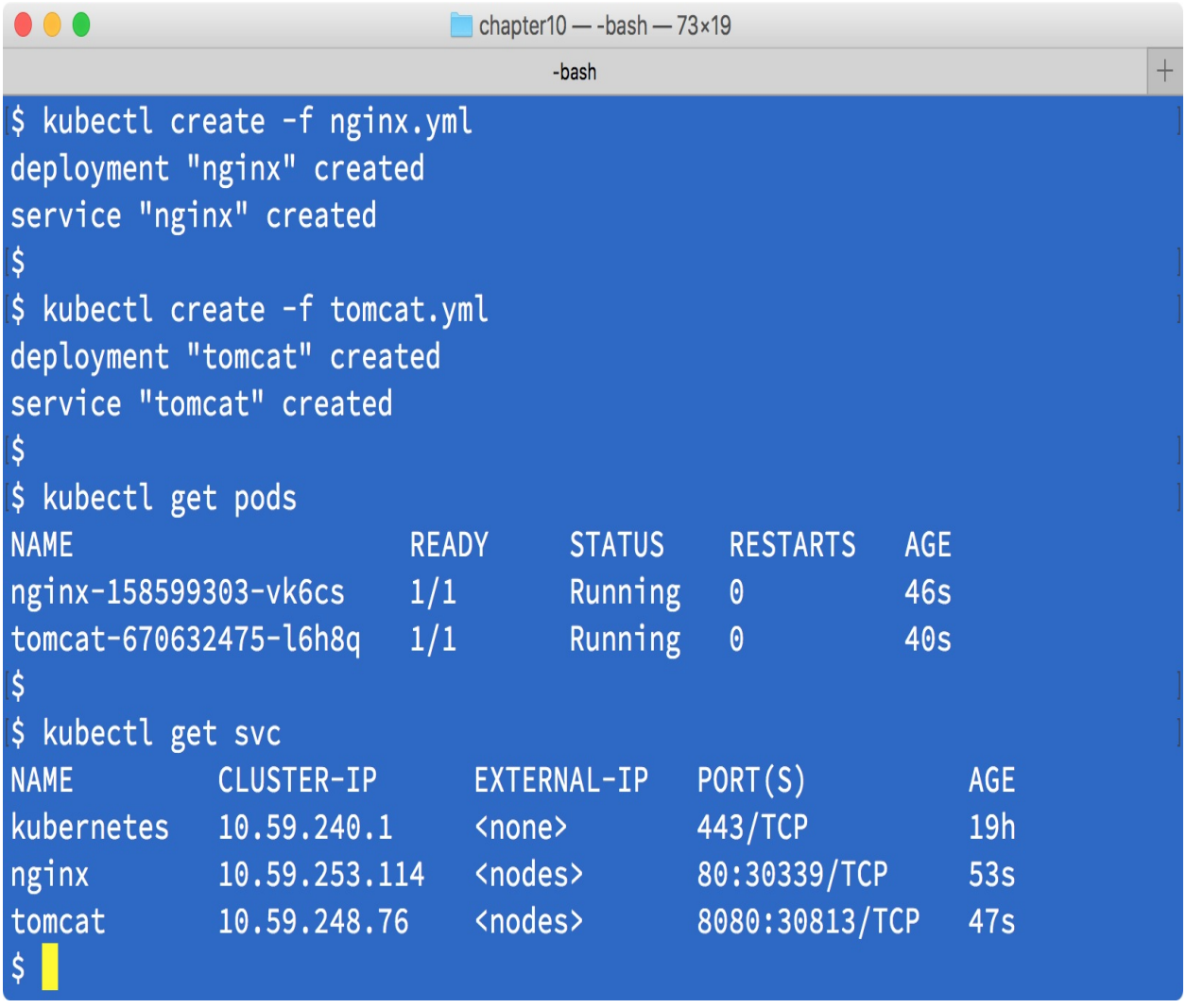
10.59.244.97
35.243.118.88
80:31213/TCP
1m

//can reach via GCP L4 Load Balancer
$ curl -I 35.243.118.88
HTTP/1.1 302 Found
Content-Type: text/html; charset=utf-8
Location: /login
Set-Cookie: grafana_sess=186d81bd66d150d5; Path=/; HttpOnly
Set-Cookie: redirect_to=%252F; Path=/; HttpOnly
Date: Sun, 09 Dec 2018 02:25:48 GMT

L7 LoadBalancer (ingress)

GKE also supports Kubernetes ingress, which can set up the GCP L7 LoadBalancer to dispatch HTTP requests to the target service based on the URL. You just need to set up one or more NodePort services and then create ingress rules to point to the services. Behind the scenes, Kubernetes automatically creates and configures the following firewall rules; `health check`, `backend` service, `forwarding rule`, and `url-maps`.

Let's create same examples that use nginx and Tomcat to deploy to the Kubernetes cluster first. These use Kubernetes Services that bind to NodePort instead of LoadBalancer:



At present, you can't access Kubernetes Service because there are as yet no firewall rules that allow access to it from the internet. Consequently, let's create Kubernetes ingress to point to these services.

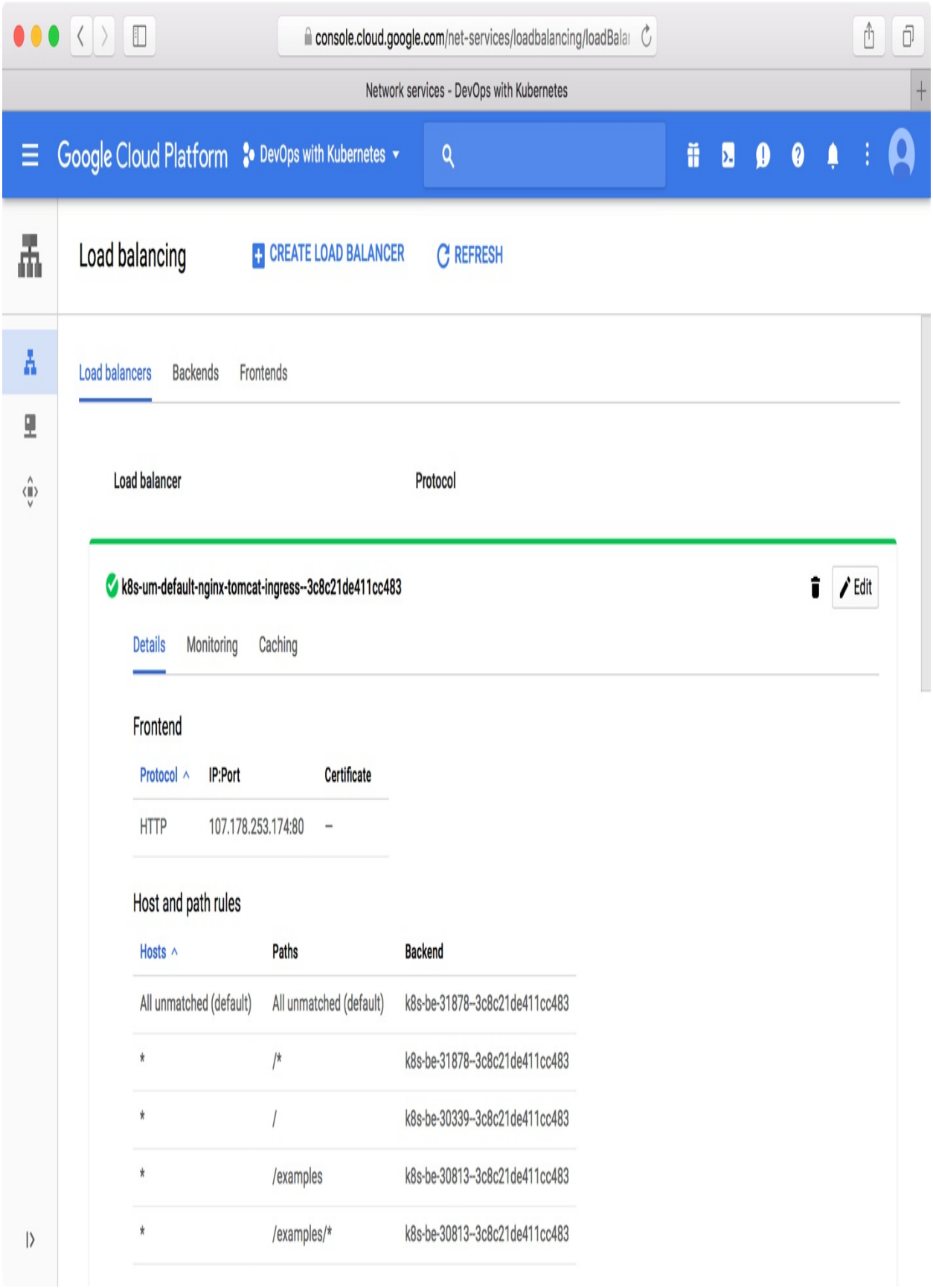
You can use `kubectl port-forward <pod name> <your machine available port><: service port number>` *to access the pod via the Kubernetes API server. For the preceding case, use* `kubectl port-forward tomcat-670632475-l6h8q 10080:8080`*. After that, open your web browser to* `http://localhost:10080/` *and then you can directly access the Tomcat pod.*

Kubernetes ingress definition is quite similar to GCP backend service definition as it needs to specify a combination of URL path, Kubernetes service name, and service port number. In this scenario, the `/` and `/*` URLs point to the `nginx` service, while the `/examples` and `/examples/*` URLs also point to the Tomcat service, as follows: **$ cat nginx-tomcat-ingress.yaml apiVersion: extensions/v1beta1 kind: Ingress metadata:**
name: nginx-tomcat-ingress
spec: rules: - http: paths: - path: / backend: serviceName: nginx servicePort: 80 - path: /examples backend: serviceName: tomcat servicePort: 8080 - path: /examples/* backend: serviceName: tomcat servicePort: 8080

$ kubectl create -f nginx-tomcat-ingress.yaml ingress "nginx-tomcat-ingress" created

It takes around 10 to 15 minutes to fully configure GCP components such as `health check`, `forwarding rule`, `backend` services, and `url-maps`: **$ kubectl get ing NAME HOSTS ADDRESS PORTS AGE nginx-tomcat-ingress * 107.178.253.174 80 1m**

You can also check the status on the web console, as follows:


console.cloud.google.com/net-services/loadbalancing/loadBal
Network services - DevOps with Kubernetes
Google Cloud Platform
DevOps with Kubernetes
Load balancing
CREATE LOAD BALANCER
REFRESH
Load balancers
Backends
Frontends
Load balancer
Protocol
k8s-um-default-nginx-tomcat-ingress--3c8c21de411cc483
Edit
Details
Monitoring
Caching
Frontend
Protocol
IP:Port
Certificate
HTTP
107.178.253.174:80
–
Host and path rules
Hosts
Paths
Backend
All unmatched (default)
All unmatched (default)
k8s-be-31878--3c8c21de411cc483
*
/*
k8s-be-31878--3c8c21de411cc483
*
/
k8s-be-30339--3c8c21de411cc483
*
/examples
k8s-be-30813--3c8c21de411cc483
*
/examples/*
k8s-be-30813--3c8c21de411cc483

Once you've completed setting up of the L7 LoadBalancer, you can access the public IP LoadBalancer address (`http://107.178.253.174/`) to see the nginx page. As well as accessing `http://107.178.253.174/examples/`, you can see the `tomcat example` page.

GKE returns 404 Not found until GKE is fully complete in order to configure the LoadBalancer.

In the preceding steps, we created and assigned an ephemeral IP address for the L7 LoadBalancer. However, the best practice when using L7 LoadBalancer is to assign a static IP address instead, because you can also associate DNS (FQDN) to the static IP address.

To do that, update the ingress setting to add an annotation named `kubernetes.io/ingress.global-static-ip-name` to associate a GCP static IP address name, as follows: **//allocate static IP as my-nginx-tomcat $ gcloud compute addresses create my-nginx-tomcat --global
//check assigned IP address $ gcloud compute addresses list NAME REGION ADDRESS STATUS my-nginx-tomcat 35.186.227.252 IN_USE
//add annotations definition $ cat nginx-tomcat-static-ip-ingress.yaml apiVersion: extensions/v1beta1 kind: Ingress metadata: name: nginx-tomcat-ingress annotations: kubernetes.io/ingress.global-static-ip-name: my-nginx-
tomcat spec: rules: - http: paths: - path: / backend: serviceName: nginx servicePort: 80 - path: /examples backend: serviceName: tomcat servicePort: 8080 - path: /examples/* backend: serviceName: tomcat servicePort: 8080
//apply command to update Ingress $ kubectl apply -f nginx-tomcat-static-ip-ingress.yaml
//check Ingress address that associate to static IP $ kubectl get ing NAME HOSTS ADDRESS PORTS AGE nginx-tomcat-ingress * 35.186.227.252 80 48m**

So, now you can access the ingress via a static IP address such as `http://35.186.227.252/` (`nginx`) and `http://35.186.227.252/examples/` (Tomcat) instead of an ephemeral IP address. This benefits to the user and preserves the static IP address. For example, when you recreate an ingress, the IP address won't be

changed.

Summary

In this chapter, we discussed Google Cloud Platform. Its basic concept is similar to AWS, but some policies and concepts are different. This is particularly the case for Google Container Engine, as this is a very powerful service for using Kubernetes as production grade. Kubernetes cluster and node management is quite easy to install and upgrade. The cloud provider is also fully integrated with GCP (especially ingress, as this can configure the L7 LoadBalancer with one command). Consequently, it's highly recommended you try GKE if you plan to use Kubernetes on the public cloud.

Chapter 12, *Kubernetes on Azure*, will explore one more public cloud service named Microsoft Azure, which also provides a managed Kubernetes service.

Kubernetes on Azure

Just like AWS and GCP, Microsoft Azure's public cloud also has a hosted offering, which is Kubernetes. The **Azure Kubernetes Service** (**AKS**) was introduced in 2017. Users of Azure can manage, deploy, and scale their containerized applications on AKS without worrying about the underlying infrastructure.

In this chapter, we'll start by giving an introduction to Azure and then go through the major services that AKS uses. We'll then learn how to launch an AKS cluster and play with it:

- Introduction to Azure
- Fundamental services in Azure
- Setting up AKS
- Azure cloud providers

Introduction to Azure

Like GCP, Microsoft Azure provides **Platform as a Service** (**PaaS**). Users can deploy their applications to the Azure app service without having to know about detailed settings and VM management. Since 2010, Azure has been serving Microsoft software and third-party software to many users. Each Azure service provides different pricing tiers. In Azure, these pricing tiers are also called *SKUs* (https://en.wikipedia.org/wiki/Stock_keeping_unit).

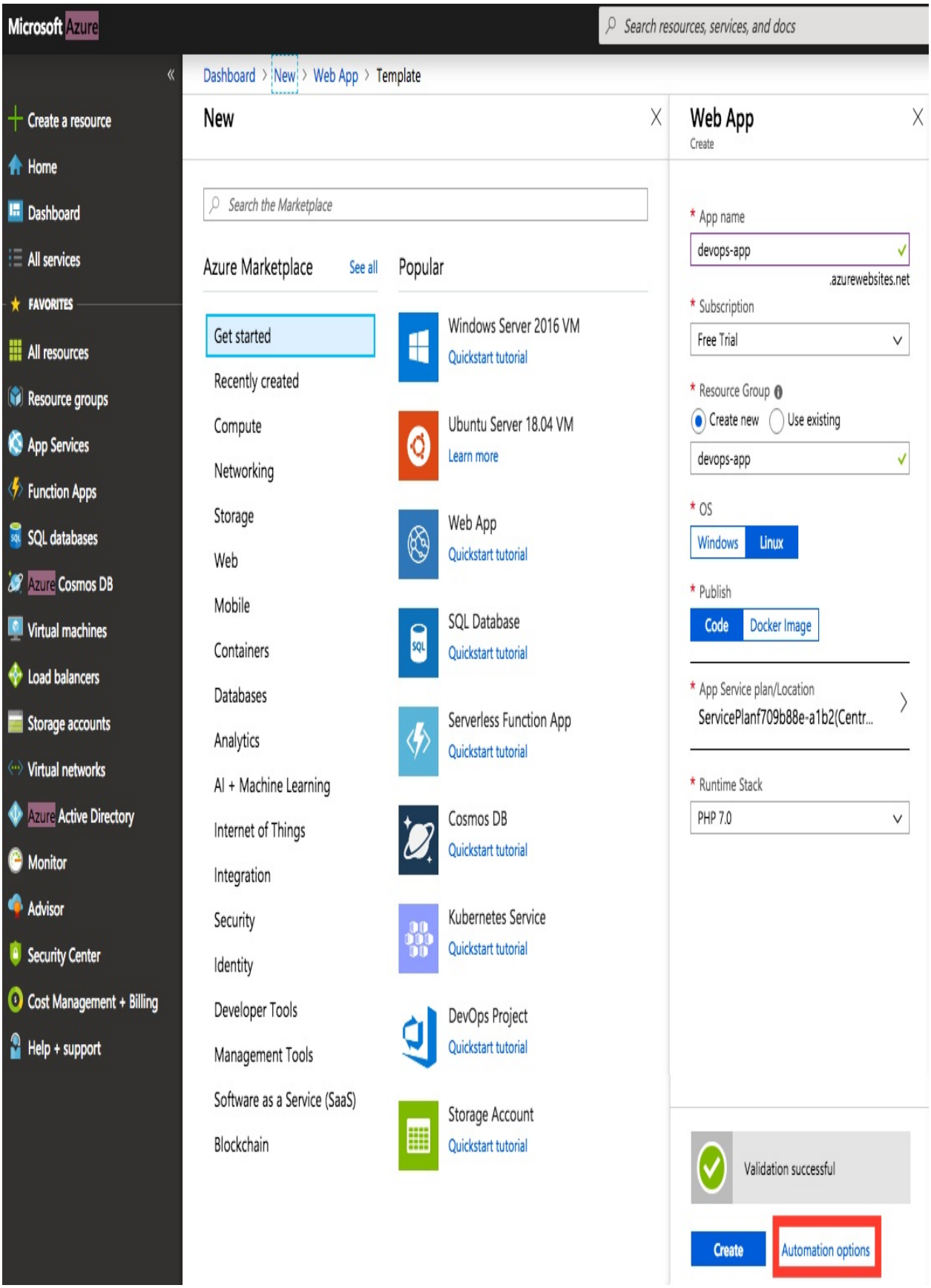
The **Azure Kubernetes Service** (**AKS**) was announced in 2017 as the new support for their original container orchestrator solution, **Azure Container Service** (**ACS**). Since then, container solutions in Azure focused more on Kubernetes support rather than other container orchestrators, such as Docker Enterprise and Mesosphere DC/OS. As a Kubernetes cloud provider, AKS provides some native support, such as Azure active directory for RBAC, Azure disks for storage class, Azure load balancers for services, and HTTP application routing for ingress.

Resource groups

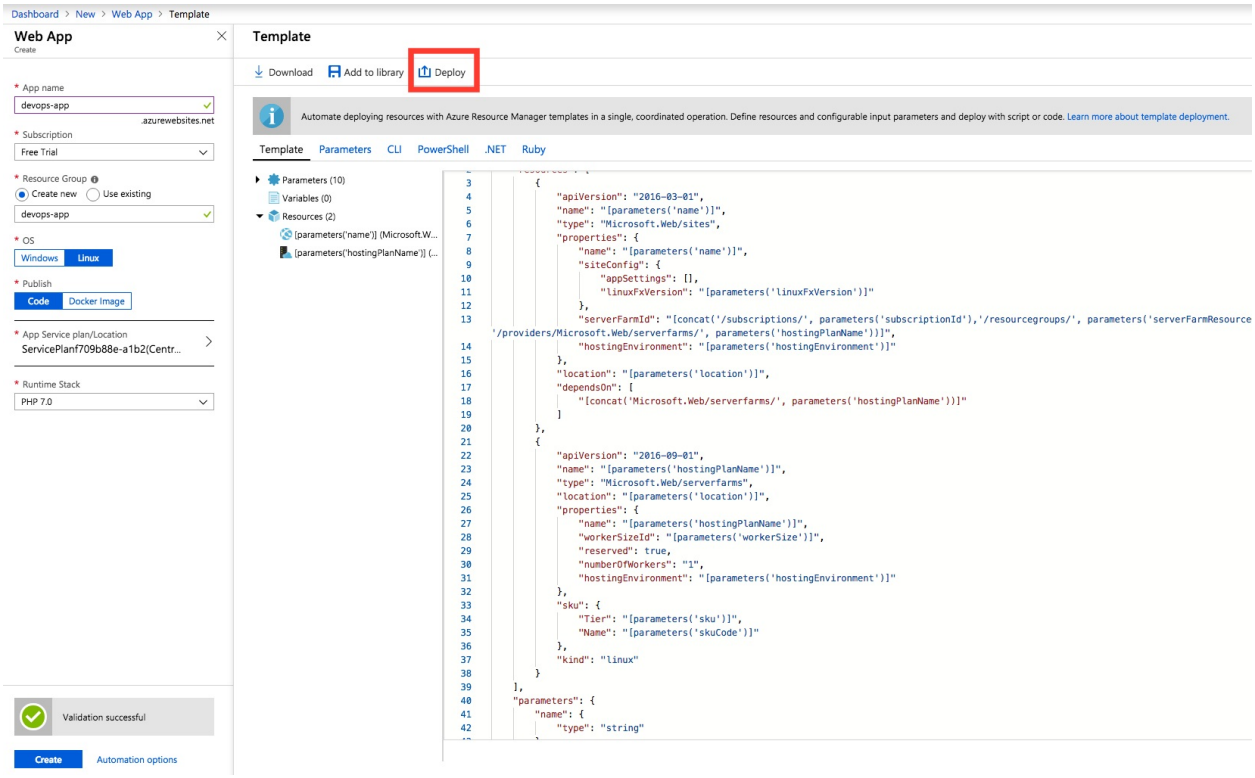
A r**esource group** in Azure is a set of resources that represent a logical group. You can deploy and delete all the resources inside a group at once. **Azure resource manager** is a tool that's used to help you manage your resource groups. In line with the spirit of infrastructure as code (https://en.wikipedia.org/wiki/Infrastructure_as_code), Azure provides a **resource manager template**, which is a file in JSON format that defines the configuration and the dependencies of the desired resources. Users can deploy the template to multiple resource groups for different environments repeatedly and consistently.

Let's see how these things look in the Azure portal. First, you'll need to have an Azure account. If you don't have one, go to https://azure.microsoft.com/features/azure-portal/ and sign up to get a free account. The Azure free account offers you 12 months of popular free services and $200 credit for 30 days. Credit card information is needed for account registration, but you won't be charged unless you upgrade your account type.

After logging in, click on Create a resource on the sidebar and go to Get started. We'll see a web app there; click on it and input the app name. For resource creation, you'll need to specify the Resource Group. We can either use an existing one or create a new one. Let's create a new one for now, as we don't have any resource groups yet. Change the Runtime Stack to your application runtime if needed. The screenshot for this is as follows:


Microsoft Azure
Search resources, services, and docs
Dashboard > New > Web App > Template
Create a resource
Home
Dashboard
All services
FAVORITES
All resources
Resource groups
App Services
Function Apps
SQL databases
Azure Cosmos DB
Virtual machines
Load balancers
Storage accounts
Virtual networks
Azure Active Directory
Monitor
Advisor
Security Center
Cost Management + Billing
Help + support
New
Search the Marketplace
Azure Marketplace
See all
Get started
Recently created
Compute
Networking
Storage
Web
Mobile
Containers
Databases
Analytics
AI + Machine Learning
Internet of Things
Integration
Security
Identity
Developer Tools
Management Tools
Software as a Service (SaaS)
Blockchain
Popular
Windows Server 2016 VM
Quickstart tutorial
Ubuntu Server 18.04 VM
Learn more
Web App
Quickstart tutorial
SQL Database
Quickstart tutorial
Serverless Function App
Quickstart tutorial
Cosmos DB
Quickstart tutorial
Kubernetes Service
Quickstart tutorial
DevOps Project
Quickstart tutorial
Storage Account
Quickstart tutorial
Web App
Create
* App name
devops-app
.azurewebsites.net
* Subscription
Free Trial
* Resource Group
Create new
Use existing
devops-app
* OS
Windows
Linux
* Publish
Code
Docker Image
* App Service plan/Location
ServicePlanf709b88e-a1b2(Centr...
* Runtime Stack
PHP 7.0
Validation successful
Create
Automation options

At the bottom of the page, beside the Create button, there is an Automation options button. If we click that, we'll see that a resource template is created automatically. If we click Deploy, the custom parameters defined by the template will be shown. For now, we will just click on Create directly. Here is a screenshot of the resource template:



After clicking Create, the console will bring us to the following view for us to explore. Let's go to our newly created resource group, `devops-app`, under the Recent resources tab:

Azure services

See all (+100) 〉

Virtual machines | Storage accounts | App Services | SQL databases | Azure Database for PostgreSQL | Azure Cosmos DB | Kubernetes services | Function Apps | Azure Databricks | Cognitive Services

Make the most out of Azure

Explore Azure with free online courses by Microsoft

Microsoft Learn

Monitor your apps and infrastructure

Azure Monitor 〉

Secure your apps and infrastructure

Security Center 〉

Optimize performance, reliability, security, and costs

Azure Advisor 〉

Connect to Azure via an authenticated browser-based shell

Cloud Shell 〉

Recent resources

See all your recent resources 〉 See all your resources 〉

NAME	TYPE	LAST VIEWED
devops-app	Resource group	Tue 3:04 PM

Useful links

Get started or go deep with technical docs

Our articles include everything from quickstarts, samples, and tutorials to help you get started, to SDKs and architecture guides for designing applications.

Discover Azure products

Explore Azure offers that help turn ideas into solutions, and get info on support, training, and pricing.

Keep current with Azure updates

Learn more and what's on the roadmap and subscribe to notifications to stay informed. Azure.Source wraps up all the news from last week in Azure.

News from the Azure team

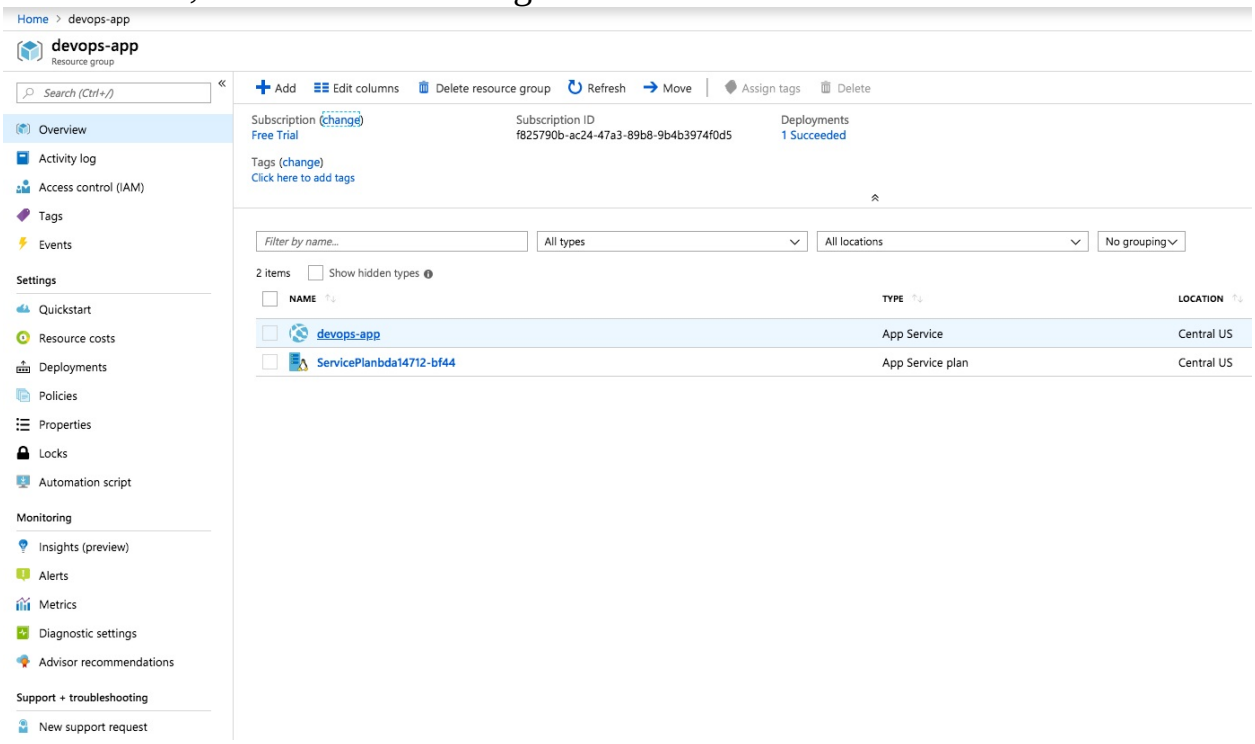
Hear right from the team developing features that help you solve problems in the Azure blog.

Azure mobile app

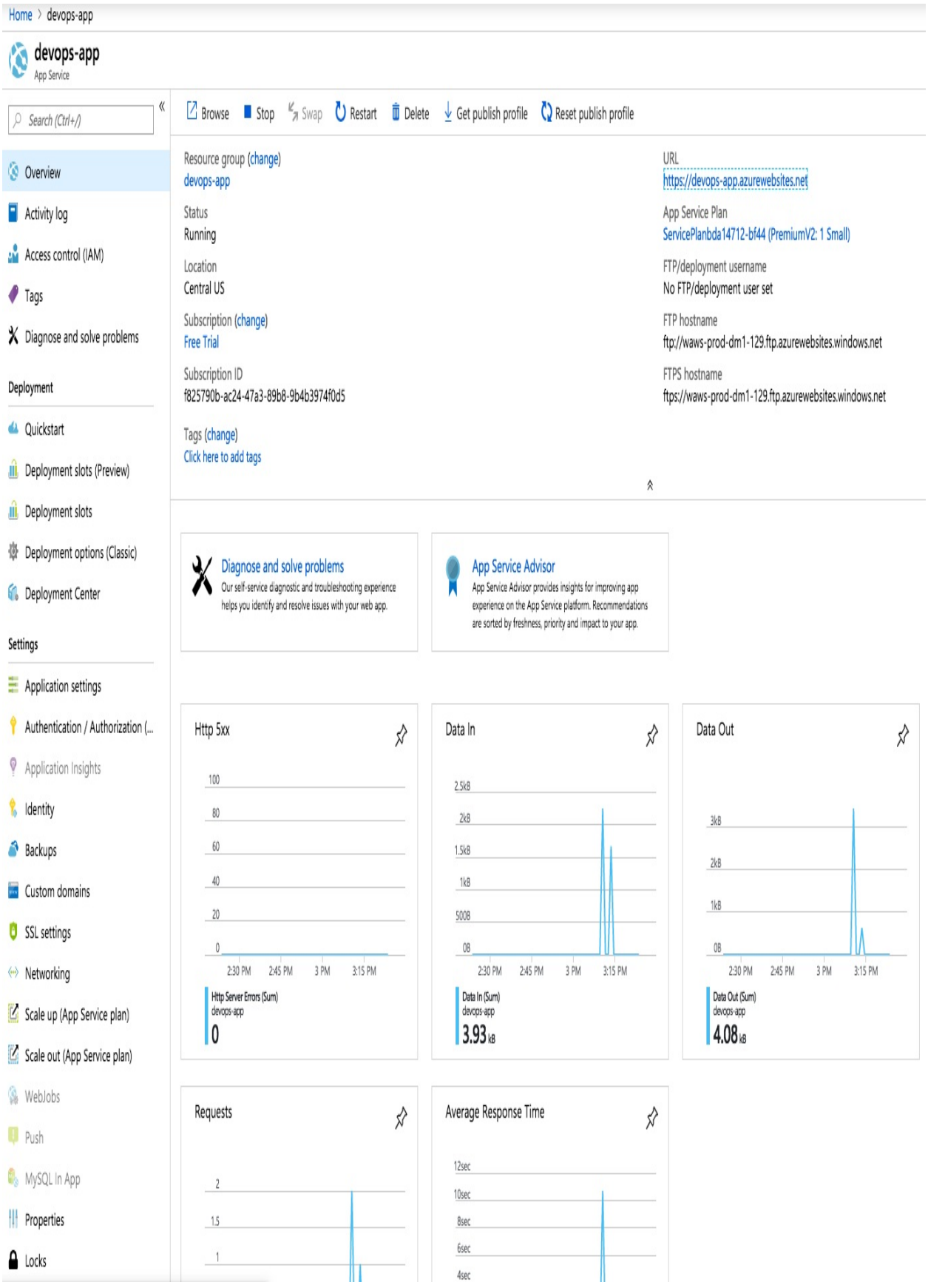
Get anytime, anywhere access to your Azure resources.

Learn more about staying connected on the go

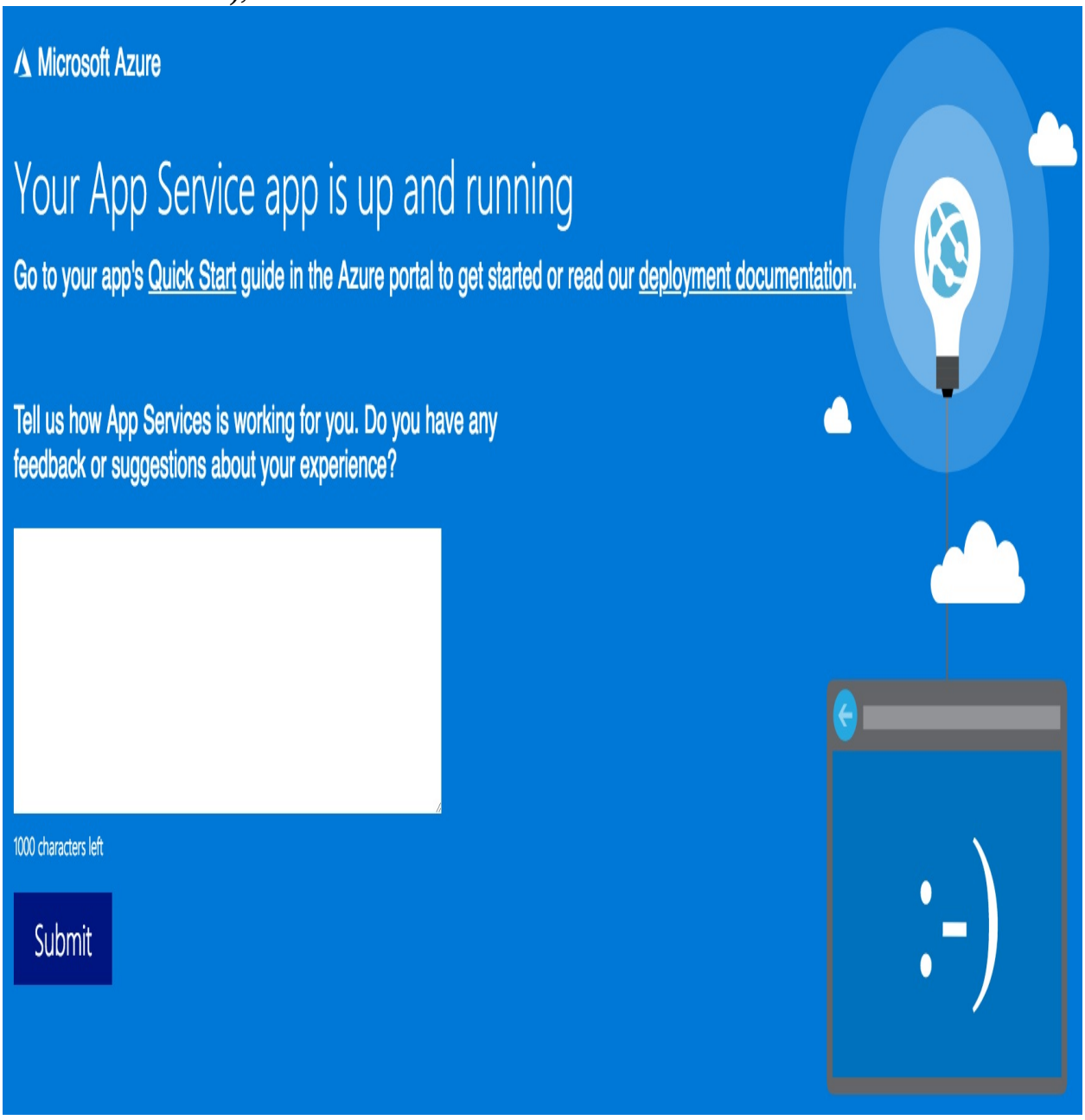
In this resource group, we can see that there's one application running in the App Services and one service plan. We can also see lots of functionalities in the sidebar. The resource group aims to give users a comprehensive view of a group of resources, so we don't need to go to a different console to find them:



If we click on the `devops-app` Resource group, it'll bring us to the app service console, which is the PaaS offering in Azure:

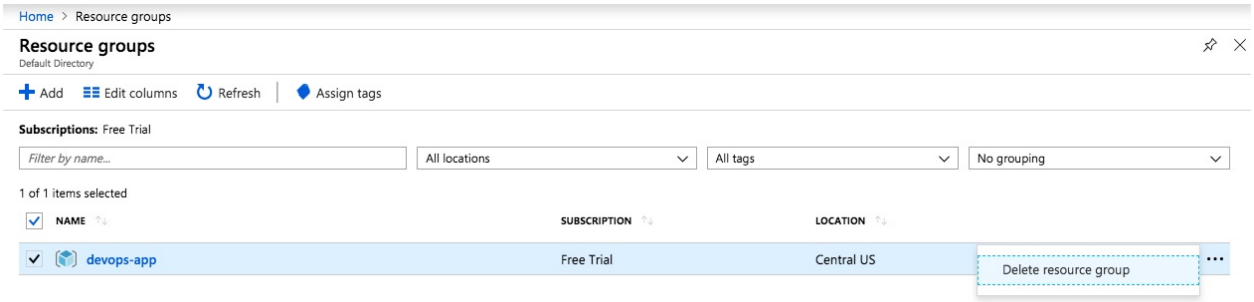

Home › devops-app
devops-app
App Service
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Deployment
Quickstart
Deployment slots (Preview)
Deployment slots
Deployment options (Classic)
Deployment Center
Settings
Application settings
Authentication / Authorization (...
Application Insights
Identity
Backups
Custom domains
SSL settings
Networking
Scale up (App Service plan)
Scale out (App Service plan)
WebJobs
Push
MySQL In App
Properties
Locks
Browse
Stop
Swap
Restart
Delete
Get publish profile
Reset publish profile
Resource group (change)
devops-app
Status
Running
Location
Central US
Subscription (change)
Free Trial
Subscription ID
f825790b-ac24-47a3-89b8-9b4b3974f0d5
Tags (change)
Click here to add tags
URL
https://devops-app.azurewebsites.net
App Service Plan
ServicePlanbda14712-bf44 (PremiumV2: 1 Small)
FTP/deployment username
No FTP/deployment user set
FTP hostname
ftp://waws-prod-dm1-129.ftp.azurewebsites.windows.net
FTPS hostname
ftps://waws-prod-dm1-129.ftp.azurewebsites.windows.net
Diagnose and solve problems
Our self-service diagnostic and troubleshooting experience helps you identify and resolve issues with your web app.
App Service Advisor
App Service Advisor provides insights for improving app experience on the App Service platform. Recommendations are sorted by freshness, priority and impact to your app.
Http 5xx
100
80
60
40
20
0
2:30 PM
2:45 PM
3 PM
3:15 PM
Http Server Errors (Sum)
devops-app
0
Data In
2.5kB
2kB
1.5kB
1kB
500B
0B
Data In (Sum)
devops-app
3.93 kB
Data Out
3kB
2kB
1kB
0B
Data Out (Sum)
devops-app
4.08 kB
Requests
2
1.5
1
Average Response Time
12sec
10sec
8sec
6sec
4sec

At this point, the sample web app has been deployed to the Azure app service. If we visit the endpoint specified in the URL (which in this case is `https://devops-app.azurewebsites.net/`), we can find it:

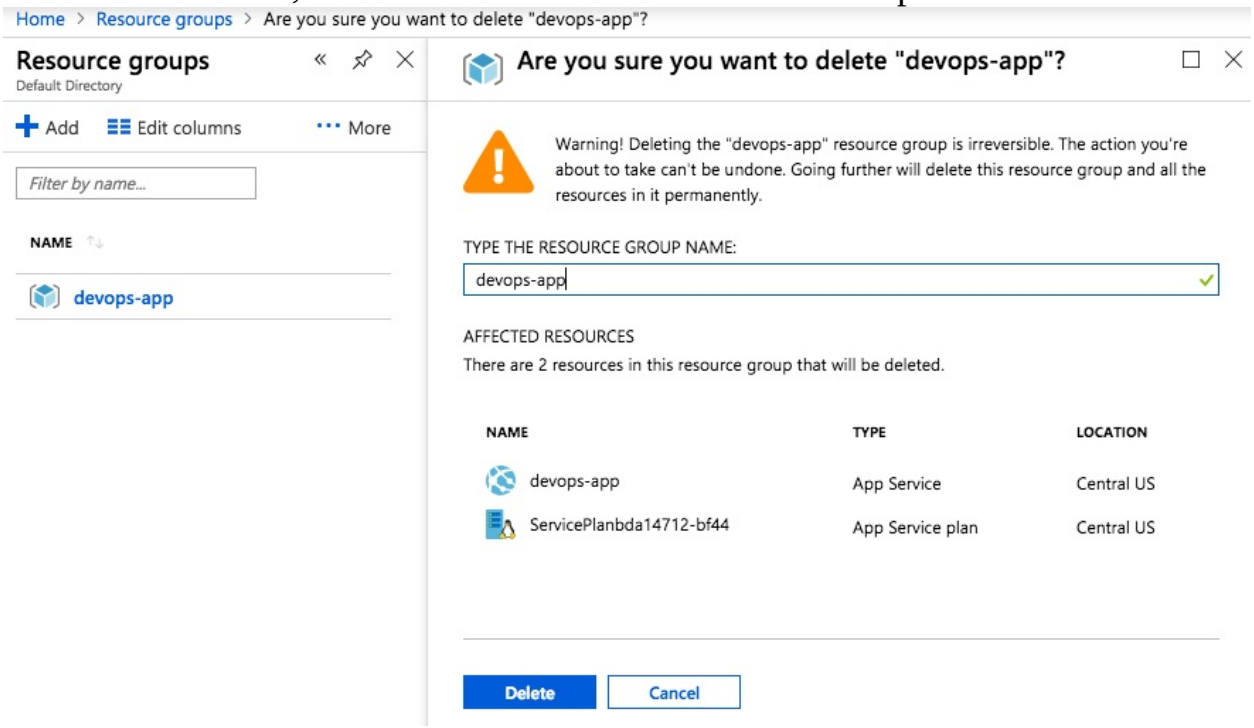


We could also upload our own web app, or integrate with our version control software, such as GitHub or Bitbucket, and build our whole pipeline with Azure. For more information, please visit the deployment center page (`https://docs.microsoft.com/en-us/azure/app-service/deploy-continuous-deployment`).

We can also easily delete resource groups in the Resource groups console:



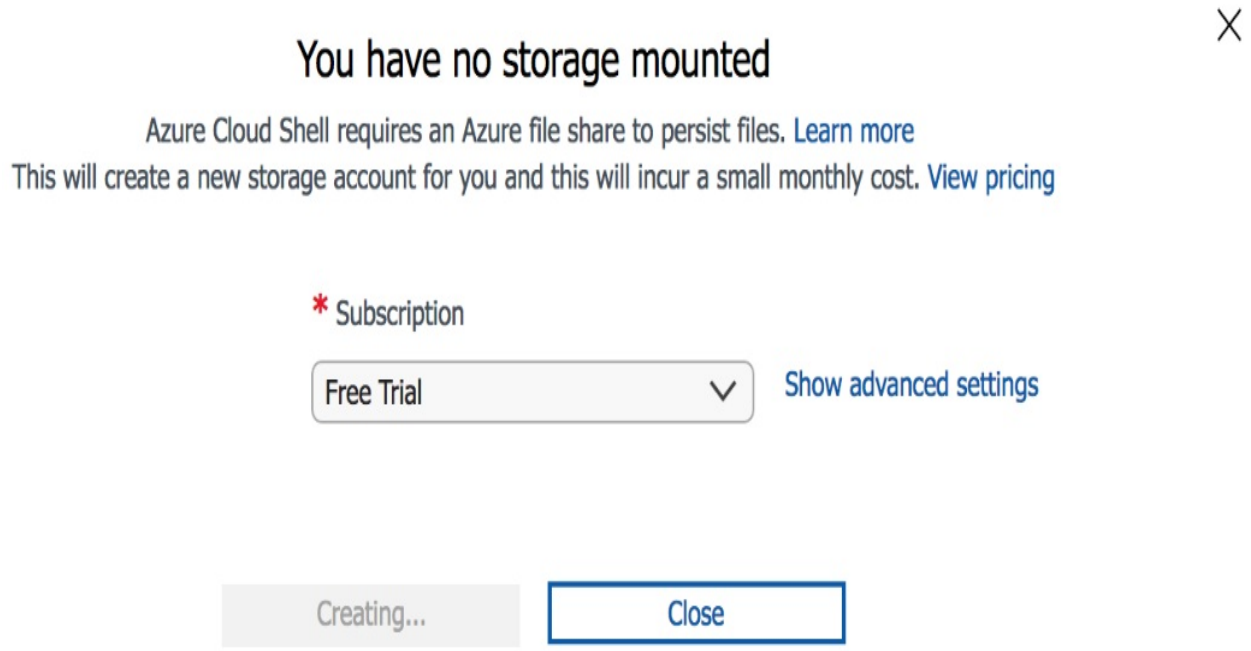
After confirmation, the related resources will be cleaned up:



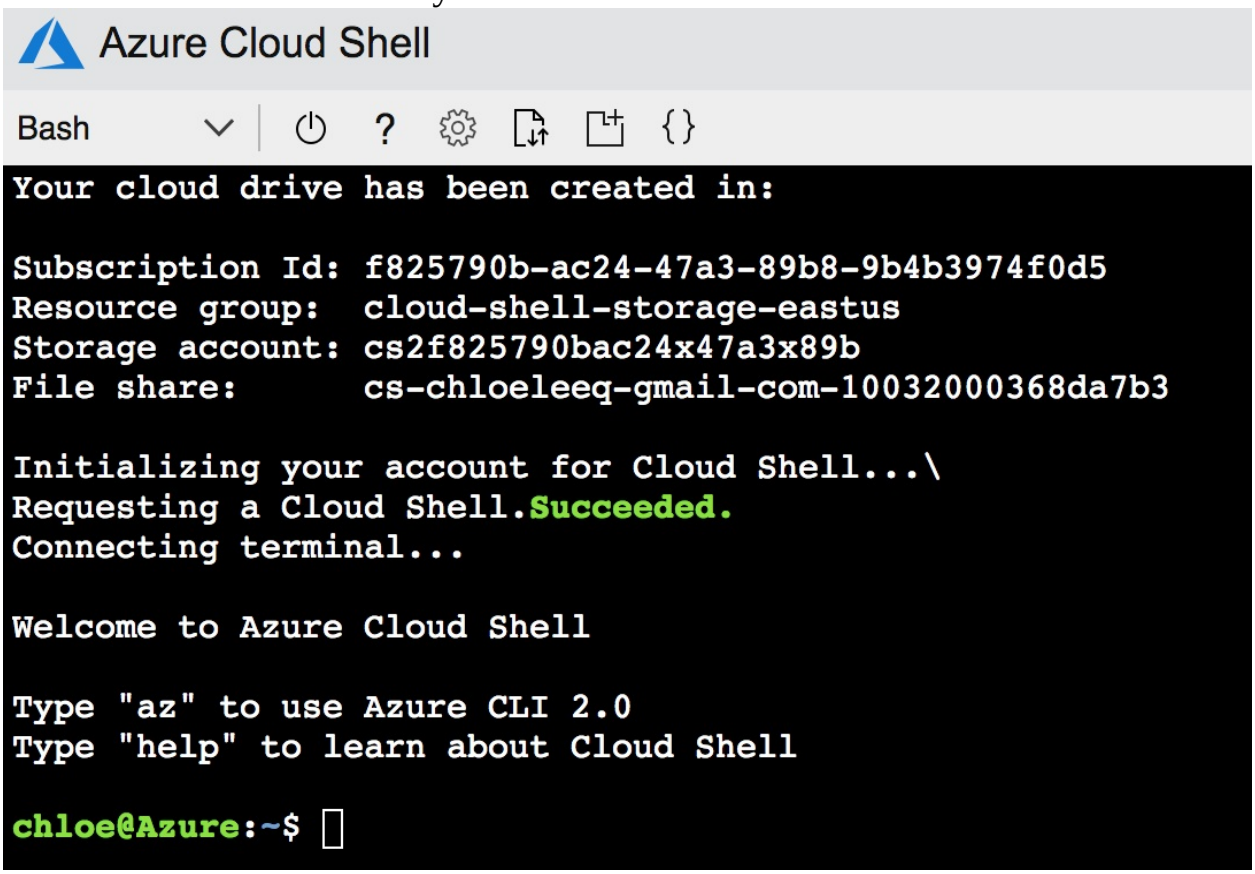
Azure virtual network

The Azure **Virtual Network** (**VNet**) creates an isolated private network segment in Azure. This concept is similar to VPC in AWS and GCP. Users specify a range of contiguous IPs (that is, CIDR: https://en.wikipedia.org/wiki/Classless_Inter-Domain_Routing) and locations (otherwise known as regions in AWS). We can find a full list of locations at https://azure.microsoft.com/global-infrastructure/locations/. We can also create multiple subnets inside a virtual network, or enable an Azure firewall upon creation. The Azure firewall is a network security service with high availability and scalability. It can control and filter traffic with user-specified rules. It also provides inbound DNAT and outbound SNAT support. Depending on the platform you're using, you can install the Azure CLI (the documentation for which can be found here: https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest) via the instructions at the following link: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest. Alternatively, you can use cloud shell (https://shell.azure.com/) directly. Azure cloud shell is a cloud-based admin shell that you can use to manage your cloud resources, which already has the Azure CLI installed.

In the following example, we'll demonstrate how to use the Azure CLI to create an Azure virtual network via an Azure cloud shell. Simply log in to your account and attach cloud storage to persist the data. Then, we're good to go:



After clicking the Create button and waiting for a few seconds, a cloud shell console will be launched in your browser:



The Azure CLI commands start with `az` as the group name. You could type `az --help` to see a list of subgroups or use `az $subgroup_name --help` any time to find more information about a subcommand for a subgroup. A subgroup might contain multiple subgroups. At the end of the command is the operation that you want to carry out with the resource and a set of parameters about the configuration. This looks as follows: **# az $subgroup1 [$subgroup2 ...] $commands [$parameters]**

In the following example, we'll create a virtual network named `devops-vnet`. First, we'll have to create a new resource group, since we deleted the only one we had in the previous section. Now, let's create a resource group called `devops` in the central US location:

```
# az group create --name devops --location centralus
{
  "id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops",
  "location": "centralus",
  "managedBy": null,
  "name": "devops",
  "properties": {
    "provisioningState": "Succeeded"
  },
  "tags": null
}
```

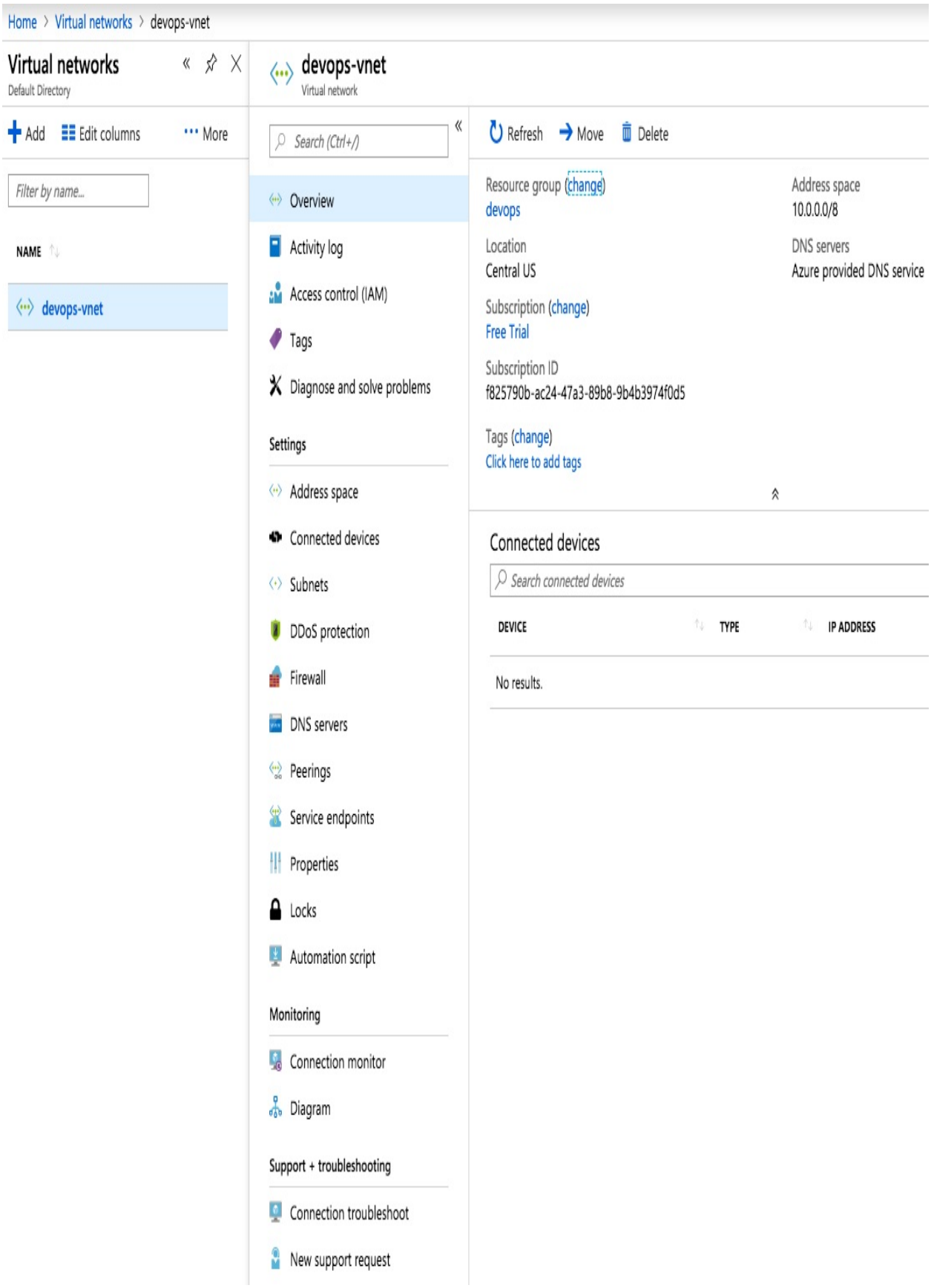
In the preceding command, the subgroup name is `group` and the operation command is `create`. Next, we'll use `network.vnet` subgroups to create our virtual network resources with the CIDR `10.0.0.0/8`, and leave the rest of the settings as their default values: **# az network vnet create --name devops-vnet --resource-group devops --subnet-name default --address-prefixes 10.0.0.0/8**

{
"newVNet": {
"addressSpace": {
"addressPrefixes": [
"10.0.0.0/8"
]
},
"ddosProtectionPlan": null,
"dhcpOptions": {
"dnsServers": []
},
"enableDdosProtection": false,

"enableVmProtection": false,
"etag": "W/\"a93c56be-6eab-4391-8fca-25e11625c6e5\"",
"id": "/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/virtualN
vnet",
"location": "centralus",
"name": "devops-vnet",
"provisioningState": "Succeeded",
"resourceGroup": "devops",
"resourceGuid": "f5b9de39-197c-440f-a43f-51964ee9e252",
"subnets": [
{
"addressPrefix": "10.0.0.0/24",
"addressPrefixes": null,
"delegations": [],
"etag": "W/\"a93c56be-6eab-4391-8fca-25e11625c6e5\"",
"id": "/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/virtualN
vnet/subnets/default",
"interfaceEndpoints": null,
"ipConfigurationProfiles": null,
"ipConfigurations": null,
"name": "default",
"networkSecurityGroup": null,
"provisioningState": "Succeeded",
"purpose": null,
"resourceGroup": "devops",
"resourceNavigationLinks": null,
"routeTable": null,
"serviceAssociationLinks": null,
"serviceEndpointPolicies": null,
"serviceEndpoints": null,
"type": "Microsoft.Network/virtualNetworks/subnets"
}
],
"tags": {},
"type": "Microsoft.Network/virtualNetworks",

"virtualNetworkPeerings": []
}
}

We could always view a list of our settings using `az` with the `list` command, such as `az network vnet list`, or go to the Azure portal to check it out:


Home > Virtual networks > devops-vnet
Virtual networks
Default Directory
Add
Edit columns
More
Filter by name...
NAME
devops-vnet
devops-vnet
Virtual network
Search (Ctrl+/)
Overview
Activity log
Access control (IAM)
Tags
Diagnose and solve problems
Settings
Address space
Connected devices
Subnets
DDoS protection
Firewall
DNS servers
Peerings
Service endpoints
Properties
Locks
Automation script
Monitoring
Connection monitor
Diagram
Support + troubleshooting
Connection troubleshoot
New support request
Refresh
Move
Delete
Resource group (change)
devops
Address space
10.0.0.0/8
Location
Central US
DNS servers
Azure provided DNS service
Subscription (change)
Free Trial
Subscription ID
f825790b-ac24-47a3-89b8-9b4b3974f0d5
Tags (change)
Click here to add tags
Connected devices
Search connected devices
DEVICE
TYPE
IP ADDRESS
No results.

# az network nsg create --name test-nsg --resource-group devops --location centralus
{
 "NewNSG": {
 "defaultSecurityRules": [
{
 "access": "Allow",
 "description": "Allow inbound traffic from all VMs in VNET",
 "destinationAddressPrefix": "VirtualNetwork",
 "name": "AllowVnetInBound",
 "priority": 65000,
 "sourceAddressPrefix": "VirtualNetwork",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 },
{
 "access": "Allow",
 "description": "Allow inbound traffic from azure load balancer",
 "destinationAddressPrefix": "*",
 "name": "AllowAzureLoadBalancerInBound",

 "priority": 65001,
 "sourceAddressPrefix": "AzureLoadBalancer",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 },
 {
 "access": "Deny",
 "description": "Deny all inbound traffic",
 "destinationAddressPrefix": "*",
 "name": "DenyAllInBound",
 "priority": 65500,
 "sourceAddressPrefix": "*",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 },
 {
 "access": "Allow",
 "description": "Allow outbound traffic from all VMs to all VMs in VNET",
 "destinationAddressPrefix": "VirtualNetwork",
 "name": "AllowVnetOutBound",

 "priority": 65000,
 "sourceAddressPrefix": "VirtualNetwork",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 },
 {
 "access": "Allow",
  "description": "Allow outbound traffic from all VMs to Internet",
 "destinationAddressPrefix": "Internet",
 "name": "AllowInternetOutBound",
 "priority": 65001,
 "sourceAddressPrefix": "*",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 },
 {
 "access": "Deny",
  "description": "Deny all outbound traffic",
 "destinationAddressPrefix": "*",
  "name": "DenyAllOutBound",

 "priority": 65500,
 "sourceAddressPrefix": "*",
 ...
 "type": "Microsoft.Network/networkSecurityGroups/defaultSecurityRules"
 }
 ],
 "id": "...test-nsg",
  "location": "centralus",
 "name": "test-nsg",
 "networkInterfaces": null,
 "provisioningState": "Succeeded",
 "resourceGroup": "devops",
 "resourceGuid": "9e0e3d0f-e99f-407d-96a6-8e96cf99ecfc",
 "securityRules": [],
 "subnets": null,
 "tags": null,
 "type": "Microsoft.Network/networkSecurityGroups"
  }
}

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.182.1"># az network nsg rule create --name test-nsg --priority 100 --
resource-group devops --nsg-name test-nsg
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.183.1">{
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.184.1">
"access": "Allow",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.185.1">
"description": null,
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.186.1">
"destinationAddressPrefix": "*",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.187.1">
"destinationAddressPrefixes": [],
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.188.1">
"destinationApplicationSecurityGroups": null,
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.189.1"> "destinationPortRange": "80",
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.190.1"> "destinationPortRanges": [],
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.191.1"> "direction": "Inbound",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.192.1">
"etag": "W/\"65e33e31-ec3c-4eea-8262-1cf5eed371b1\"",

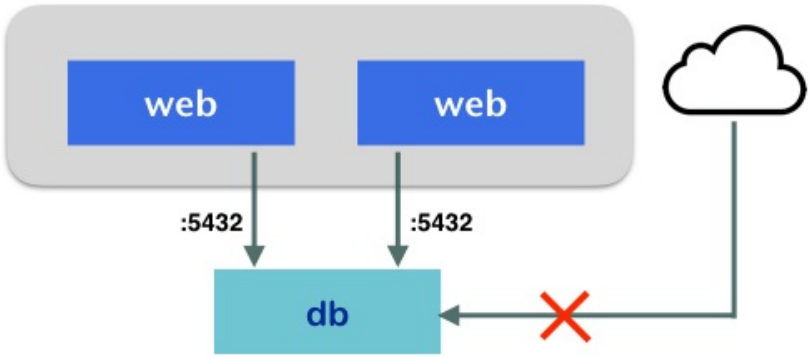
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.193.1"> "id":
"/subscriptions/.../resourceGroups/devops/providers/Microsoft.Network/networkS
nsg/securityRules/test-nsg",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.194.1">
"name": "test-nsg",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.195.1">
"priority": 100,
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.196.1">
"protocol": "*",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.197.1">
"provisioningState": "Succeeded",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.198.1">
"resourceGroup": "devops",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.199.1">
"sourceAddressPrefix": "*",
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.200.1">

"sourceAddressPrefixes": [],
"sourceApplicationSecurityGroups": null,
"sourcePortRange": "*",
"sourcePortRanges": [],
"type": "Microsoft.Network/networkSecurityGroups/securityRules"

}

Application security groups

Application security groups are a logical collection of VM NICs, which can be a source of destinations in the network security group rules. They make network security groups even more flexible. For example, let's assume that we have two VMs that will access the PostgreSQL database via `5432` ports. We want to make sure that only those VMs have access to the database:



We can create two application security groups named `web` and `db`. Then, we join the VMs to the web group and the database to the db group, and create the following network security group rules:

Direction	Priority	Source	Source ports	Destination	Dest ports	Protocol
Inbound	120	*	*	`db`	`0-65535`	All
Inbound	110	`web`	*	`db`	`5432`	TCP

According to this table, the priority of the second rule is higher than the first one.

Only the web group has access to the db group with port `5432`. All other inbound traffic will be denied.

Subnets

A subnet can be associated to a network security group. A subnet can also be associated with a route table so that it has specific routes.

Just like AWS, Azure also provides route table resources for route management. By default, Azure already provides default routing for virtual networks and subnets. We don't need to worry about the routes when we use the AKS service.

When creating a virtual network, a `default` subnet will be created by default:

```
# az network vnet subnet list --vnet-name devops-vnet --resource-group devops
[
  {
    "addressPrefix": "10.0.0.0/24",
    "addressPrefixes": null,
    "delegations": [],
    "etag": "W/\"a93c56be-6eab-4391-8fca-25e11625c6e5\"",
    "id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/prc
    "interfaceEndpoints": null,
    "ipConfigurationProfiles": null,
    "ipConfigurations": null,
    "name": "default",
    "networkSecurityGroup": null,
    "provisioningState": "Succeeded",
    "purpose": null,
    "resourceGroup": "devops",
    "resourceNavigationLinks": null,
    "routeTable": null,
    "serviceAssociationLinks": null,
    "serviceEndpointPolicies": null,
    "serviceEndpoints": null,
    "type": "Microsoft.Network/virtualNetworks/subnets"
  }
]
```

Other than the default subnet, let's create one more subnet with the prefix `10.0.1.0/24`. Note that the CIDR of the subnet needs to be in the same CIDR prefix network block as the VNet in which the subnet is located:

```
# az network vnet subnet create --address-prefixes 10.0.1.0/24 --name test --vnet-name d
{
  "addressPrefix": "10.0.1.0/24",
  "addressPrefixes": null,
  "delegations": [],
  "etag": "W/\"9f7e284f-fd31-4bd3-ad09-f7dc75bb7c68\"",
  "id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/provi
  "interfaceEndpoints": null,
  "ipConfigurationProfiles": null,
  "ipConfigurations": null,
  "name": "test",
```

```
  "networkSecurityGroup": null,
  "provisioningState": "Succeeded",
  "purpose": null,
  "resourceGroup": "devops",
  "resourceNavigationLinks": null,
  "routeTable": null,
  "serviceAssociationLinks": null,
  "serviceEndpointPolicies": null,
  "serviceEndpoints": null,
  "type": "Microsoft.Network/virtualNetworks/subnets"
}
```

We can now list the subnets in this VNet: **# az network vnet subnet list --vnet-name devops-vnet --resource-group devops | jq .[].name**
"default"
"test"

`jq` (https://stedolan.github.io/jq/):

`jq` is a JSON command-line processor that is installed in the cloud shell by default. It's a very convenient tool to list the desired fields inside a JSON output. If you're not familiar with `jq`, take a look at the manual at the following link: https://stedolan.github.io/jq/manual/.

Azure virtual machines

Virtual Machines (**VM**) in Azure are like Amazon EC2s. To launch an instance, we have to know which VM image we want to launch. We can use the `az vm image list` command to list a set of images that we can use. In the following example, we'll use a CentOS image: **# az vm image list --output table**
You are viewing an offline list of images, use --all to retrieve an up-to-date list
Offer Publisher Sku Urn UrnAlias Version
------------ -------------------- ----------------- --
CentOS OpenLogic 7.5 OpenLogic:CentOS:7.5:latest CentOS latest
CoreOS CoreOS Stable CoreOS:CoreOS:Stable:latest CoreOS latest
Debian credativ 8 credativ:Debian:8:latest Debian latest
openSUSE-Leap SUSE 42.3 SUSE:openSUSE-Leap:42.3:latest openSUSE-Leap latest
RHEL RedHat 7-RAW RedHat:RHEL:7-RAW:latest RHEL latest
SLES SUSE 12-SP2 SUSE:SLES:12-SP2:latest SLES latest
UbuntuServer Canonical 16.04-LTS Canonical:UbuntuServer:16.04-LTS:latest UbuntuLTS latest
WindowsServer MicrosoftWindowsServer 2019-Datacenter MicrosoftWindowsServer:WindowsServer:2019-Datacenter:latest Win2019Datacenter latest
WindowsServer MicrosoftWindowsServer 2016-Datacenter MicrosoftWindowsServer:WindowsServer:2016-Datacenter:latest Win2016Datacenter latest
WindowsServer MicrosoftWindowsServer 2012-R2-Datacenter MicrosoftWindowsServer:WindowsServer:2012-R2-Datacenter:latest Win2012R2Datacenter latest
WindowsServer MicrosoftWindowsServer 2012-Datacenter MicrosoftWindowsServer:WindowsServer:2012-Datacenter:latest Win2012Datacenter latest
WindowsServer MicrosoftWindowsServer 2008-R2-SP1 MicrosoftWindowsServer:WindowsServer:2008-R2-SP1:latest Win2008R2SP1 latest

Then, we could use `az vm create` to launch our VM. Specifying `--generate-ssh-keys` will create an ssh for you to access: **# az vm create --resource-group devops --name newVM --image CentOS --admin-username centos-user --generate-ssh-keys**
SSH key files '/home/chloe/.ssh/id_rsa' and '/home/chloe/.ssh/id_rsa.pub' have been generated under ~/.ssh to allow SSH access to the VM. If using machines without permanent storage, back up your keys to a safelocation.
- Running ..
{
"fqdns": "",
"id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Compute/virtual
"location": "centralus",
"macAddress": "00-0D-3A-96-6B-A2",
"powerState": "VM running",
"privateIpAddress": "10.0.0.4",
"publicIpAddress": "40.77.97.79",
"resourceGroup": "devops",
"zones": ""
}

We can see that the `publicIpAddress` of the newly created VM is `40.77.97.79`. Let's connect to it with the username we specified earlier: **# ssh centos-user@40.77.97.79**
The authenticity of host '40.77.97.79 (40.77.97.79)' can't be established.
ECDSA key fingerprint is SHA256:LAvnQH94bY7NaIoNgDLM5iMHT1LMRseFwu2HPqicTuo.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '40.77.97.79' (ECDSA) to the list of known hosts.
[centos-user@newVM ~]$

It isn't appropriate to allow SSH into the instance all the time. Let's take a look at how to fix this. First, we'll have to know the network interfaces that attach to this VM: **# az vm get-instance-view --name newVM --resource-group devops**
{
...

"networkProfile": {
"networkInterfaces": [
{
"id": "/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networ
"primary": null,
"resourceGroup": "devops"
}
]
},
...
}

After we find the `id` of the NIC, we can find the associated network security group based on the NIC ID: **# az network nic show --ids**
/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networ
| jq .networkSecurityGroup
{
"defaultSecurityRules": null,
"etag": null,
"id": "/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networ
"location": null,
"name": null,
"networkInterfaces": null,
"provisioningState": null,
"resourceGroup": "devops",
"resourceGuid": null,
"securityRules": null,
"subnets": null,
"tags": null,
"type": null
}

Here, we can see that the name of the NSG is `newVMNSG`. Let's list the rules that

attach to this NSG: **# az network nsg rule list --resource-group devops --nsg-name newVMNSG**

[
{
"access": "Allow",
"description": null,
"destinationAddressPrefix": "*",
"destinationAddressPrefixes": [],
"destinationApplicationSecurityGroups": null,
"destinationPortRange": "22",
"destinationPortRanges": [],
"direction": "Inbound",
"etag": "W/\"9ab6b2d7-c915-4abd-9c02-a65049a62f02\"",
"id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networkSecurityGroups/newVMNSG/securityRules/default-allow-ssh",
"name": "default-allow-ssh",
"priority": 1000,
"protocol": "Tcp",
"provisioningState": "Succeeded",
"resourceGroup": "devops",
"sourceAddressPrefix": "*",
"sourceAddressPrefixes": [],
"sourceApplicationSecurityGroups": null,
"sourcePortRange": "*",
"sourcePortRanges": [],
"type": "Microsoft.Network/networkSecurityGroups/securityRules"
}
]

There is a `default-allow-ssh` rule with the ID `/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networkSecurityGroups/newVMN` `allow-ssh` attached to the NSG. Let's delete it: **# az network nsg rule delete --ids "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/networkSecurityGroups/newVMNSG/securityRules/default-allow-ssh"**

As soon as we delete the rule, we can no longer access the VM via SSH: **# ssh**

centos-user@40.77.97.79

...

Storage account

A storage account is a home for the storage objects provided by Azure storage solutions, such as files, tables, and disks. Users can create one or multiple storage accounts based on their usage.

There are three types of storage accounts:

- General-purpose v2 accounts
- General-purpose v1 accounts
- Blob storage accounts

v1 is the legacy type of storage account and blob storage accounts only allow us to use blob storage. v2 accounts are the most recommended account type in Azure right now.

Load balancers

The functionality of load balancers in Azure is similar to other public cloud offerings, which are used to route traffic to the backend. They also provide a health check to the endpoint and drain the connection if they find any unhealthy backends. The main difference between Azure load balancers to other load balancers is that Azure load balancers can have multiple IP addresses and multiple ports. This means that AKS doesn't need to create a new load balancer when a new LoadBalancer Service is created. Instead, it creates a new frontend IP inside the load balancer.

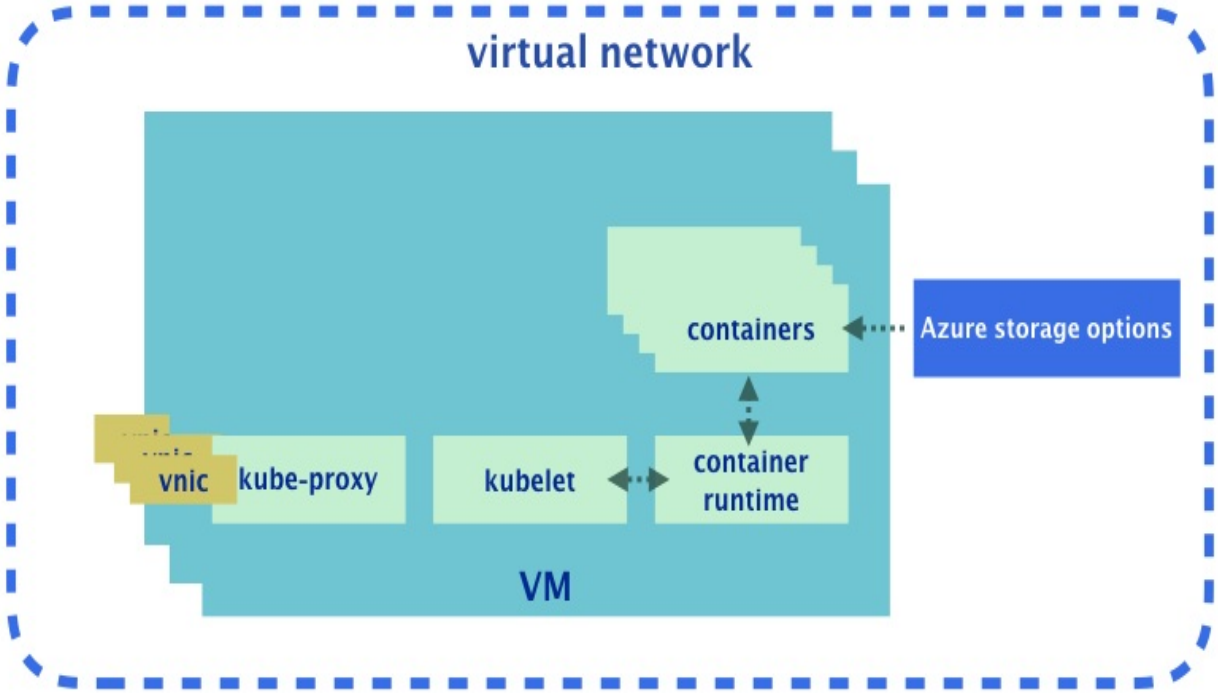
Azure disks

There are two types of Azure disks: managed disks and unmanaged disks. Before Azure managed disks, users had to create storage accounts to use unmanaged disks. A storage account might exceed the scalability target (https://docs.microsoft.com/en-us/azure/storage/common/storage-scalability-targets) and impact the performance of I/O. With managed disks, we don't need to create storage accounts by ourselves; Azure manages the accounts behind the scene. There are different types of performance tiers: standard HDD disks, standard SSD disks, and Premium SSD disks. HDD disks (https://en.wikipedia.org/wiki/Hard_disk_drive) are a cost-effective option, while SSDs (https://en.wikipedia.org/wiki/Solid-state_drive) have better performance. For an I/O-intensive workload, the Premium tier is the best option. With premium SSD disks attaching to the VM, the VM can reach up to 80,000 IOPS and 2,000 MB/s for disk throughput. Azure disks also offer four types of replication:

- **Locally-Redundant Storage (LRS)**: Data is only persisted in one single zone
- **Zone-Redundant Storage (ZRS)**: Data is persisted across three zones
- **Geo-Redundant Storage (GRS)**: Cross-regional replication
- **Read-Access Geo-Redundant Storage (RA-GRS)**: Cross-regional replication with read replica

Azure Kubernetes service

Azure Kubernetes service is a hosted Kubernetes service in Azure. A cluster contains a set of nodes (such as Azure VMs). Just like a normal Kubernetes node, kube-proxy and kubelet are installed on the node. kube-proxy, which communicates with the Azure virtual NIC, manages the route in and out for services and pods. kubelet receives the request from the master, schedules the pods, and reports the metrics. In Azure, we could mount various Azure storage options such as Azure disk and Azure files as the **Persistent Volume** (**PV**) for persisting the data for containers. An illustration of this is shown here:



Want to build a cluster from scratch?
If you would prefer to build a cluster on your own, be sure to check out the AKS-engine project (https://github.com/Azure/aks-engine), which builds Kubernetes infrastructure in Azure using the Azure resource manager.

Setting up your first Kubernetes cluster on AKS

An AKS cluster can be launched in its own VPC (basic networking configuration) or in an existing VPC (advanced networking configuration); both can be launched via Azure CLI. There are a set of arguments that we can specify during cluster creation, which include the following:

Arguments	Description
`--name`	The cluster name.
`--enable-addons`	Enables the Kubernetes addons module in a comma-separated list.
`--generate-ssh-keys`	Generates SSH key files if they do not already exist.
`--node-count`	The number of nodes. The default value is three.
`--network-policy`	(Preview) Enables or disables the network policy. The default is that it is disabled.
`--vnet-subnet-id`	The subnet ID in a VNet to deploy the cluster.
`--node-vm-size`	The size of the VMs. The default is `Standard_DS2_v2`.
`--service-cidr`	A CIDR notation IP range from which to assign service cluster IPs.
`--max-pods`	The default is `110` or `30` for an advanced network configuration (using an existing VNet).

In the following example, we'll first create a cluster with two nodes and enable `addons` for monitoring to enable Azure monitor for the cluster, and `http_application_routing` to enable HTTP application routing for ingress: **// create an AKS cluster with two nodes**
 # az aks create --resource-group devops --name myAKS --node-count 2 --enable-addons monitoring,http_application_routing --generate-ssh-keys

Running...
 # az aks list --resource-group devops
[
{
"aadProfile":null,
"addonProfiles":{
"httpApplicationRouting":{
"config":{
"HTTPApplicationRoutingZoneName":"cef42743fd964970b357.centralus.al
},
"enabled":true
},
"omsagent":{
"config":{
"logAnalyticsWorkspaceResourceID":...
},
"enabled":true
}
},
"agentPoolProfiles":[
{
"count":2,
"maxPods":110,
"name":"nodepool1",
"osDiskSizeGb":30,
"osType":"Linux",
"storageProfile":"ManagedDisks",
"vmSize":"Standard_DS2_v2"
}
],
"dnsPrefix":"myAKS-devops-f82579",
"enableRbac":true,
"fqdn":"myaks-devops-f82579-077667ba.hcp.centralus.azmk8s.io",
"id":"/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourcegroups/devops/providers/Microsoft.ContainerService
"kubernetesVersion":"1.9.11",
"linuxProfile":{

"adminUsername":"azureuser",
"ssh":{
"publicKeys":[
{
"keyData":"ssh-rsa xxx"
}
]
}
},
"location":"centralus",
"name":"myAKS",
"networkProfile":{
"dnsServiceIp":"10.0.0.10",
"dockerBridgeCidr":"172.17.0.1/16",
"networkPlugin":"kubenet",
"networkPolicy":null,
"podCidr":"10.244.0.0/16",
"serviceCidr":"10.0.0.0/16"
},
"nodeResourceGroup":"MC_devops_myAKS_centralus",
"provisioningState":"Succeeded",
"resourceGroup":"devops",
"servicePrincipalProfile":{
"clientId":"db016e5d-b3e5-4e22-a844-1dad5d16fec1"
},
"type":"Microsoft.ContainerService/ManagedClusters"
}
]

After the cluster is launched, we can use the get-credentials subcommand to configure our `kubeconfig`. The context name will be the cluster name, which in this case is `myAKS`: **// configure local kubeconfig to access the cluster**
 # az aks get-credentials --resource-group devops --name myAKS
Merged "myAKS" as current context in /home/chloe/.kube/config

// check current context
 # kubectl config current-context

myAKS

Let's see whether the nodes have joined the cluster. Make sure that all of the nodes are in the `Ready` state: **# kubectl get nodes**
NAME STATUS ROLES AGE VERSION
aks-nodepool1-37748545-0 Ready agent 12m v1.9.11
aks-nodepool1-37748545-1 Ready agent 12m v1.9.11

Let's try to deploy a `ReplicaSet` via the example we used in `Chapter3`: **# kubectl create -f chapter3/3-2-3_Service/3-2-3_rs1.yaml**
replicaset.apps/nginx-1.12 created

Create a service to access the `ReplicaSet`. We will use the `chapter3/3-2-3_Service/3-2-3_service.yaml` file and add the `type: LoadBalancer` line in the `spec`: **kind: Service**
apiVersion: v1
metadata:
name: nginx-service
spec:
selector:
project: chapter3
service: web
type: LoadBalancer
ports:
- protocol: TCP
port: 80
targetPort: 80
name: http

We can then watch the service until the `EXTERNAL-IP` changes to an external IP address. Here, we get `40.122.78.184`: **// watch the external ip from <pending> to IP**
 # kubectl get svc --watch
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
kubernetes ClusterIP 10.0.0.1 <none> 443/TCP 6h
nginx-service LoadBalancer 10.0.139.13 40.122.78.184 80:31011/TCP 1m

Let's visit the site!

In the preceding example, we have demonstrated how to deploy an nginx service to AKS and use its load balancer. How about if we already have a set of resources in an existing VNet, and we want to launch an AKS cluster inside the VNet to communicate with existing resources? Then, we need to use advanced networking in AKS. The fundamental difference between basic and advanced networking is that basic networking uses kubenet (`https://github.com/vplauzon/aks/tree/master/aks-kubenet`) as a network plugin, while advanced networking uses the Azure CNI plugin (`https://github.com/Azure/azure-container-networking/tree/master/cni`). Compared to basic networking, advanced networking has more limitations. For example, the default maximum number of pods on a node is 30, instead of 110. This is because only 30 additional IP addresses are configured by Azure CNI for the NIC on a node. The pod IPs are the secondary IPs on the NICs, so the private IPs are assigned the pods that are accessible inside the virtual network. When using kubenet, the cluster IPs are assigned to the pods, which don't belong to the virtual network but are instead managed by AKS. The cluster IPs won't be accessible outside of the cluster. Unless you have special requirements, such as if you want to gain access to the pods from outside the cluster or you want connectivity between existing resources and the cluster, the basic networking configuration should be able to fulfill most scenarios.

Let's see how we can create an AKS cluster with advanced networking. We need to specify the existing subnet for the cluster. First, let's list the subnet IDs in the VNet `devops-vnet` file that we created earlier: **# az network vnet subnet list --vnet-name devops-vnet --resource-group devops | jq .[].id "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/virtual**

vnet/subnets/default"
"/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/virtualN
vnet/subnets/test"

Let's use the default subnet. Note that one subnet should only locate one AKS cluster in advanced networking. Also, the service CIDR we specified, which is used to assign cluster IPs, cannot overlap the subnet CIDR in which the cluster is located: **# az aks create --resource-group devops --name myAdvAKS --network-plugin azure --vnet-subnet-id /subscriptions/f825790b-ac24-47a3-89b8-**
9b4b3974f0d5/resourceGroups/devops/providers/Microsoft.Network/virtualN
vnet/subnets/default --service-cidr 10.2.0.0/24 --node-count 1 --enable-addons monitoring,http_application_routing --generate-ssh-keys
- Running ..

{
"aadProfile": null,
"addonProfiles": {
"httpApplicationRouting": {
"config": {
"HTTPApplicationRoutingZoneName":
"cef42743fd964970b357.centralus.aksapp.io"
},
"enabled": true
},
"omsagent": {
"config": {
"logAnalyticsWorkspaceResourceID":
},
"enabled": true
}
},
"agentPoolProfiles": [
{
"count": 1,
"maxPods": 30,
"name": "nodepool1",

```
...
"vmSize": "Standard_DS2_v2"
}
],
"dnsPrefix": "myAdvAKS-devops-f82579",
"enableRbac": true,
"fqdn": "myadveks-devops-f82579-059d5b24.hcp.centralus.azmk8s.io",
"id": "/subscriptions/f825790b-ac24-47a3-89b8-
9b4b3974f0d5/resourcegroups/devops/providers/Microsoft.ContainerService
"kubernetesVersion": "1.9.11",
"linuxProfile": {
"adminUsername": "azureuser",
"ssh": {
"publicKeys": [
...
]
}
},
"location": "centralus",
"name": "myAdvAKS",
"networkProfile": {
"dnsServiceIp": "10.0.0.10",
"dockerBridgeCidr": "172.17.0.1/16",
"networkPlugin": "azure",
"networkPolicy": null,
"podCidr": null,
"serviceCidr": "10.2.0.0/24"
},
"nodeResourceGroup": "MC_devops_myAdvAKS_centralus",
"provisioningState": "Succeeded",
"resourceGroup": "devops",
"servicePrincipalProfile": {
"clientId": "db016e5d-b3e5-4e22-a844-1dad5d16fec1",
"secret": null
},
"tags": null,
"type": "Microsoft.ContainerService/ManagedClusters"
```

}

Remember to configure `kubeconfig`: **# az aks get-credentials --resource-group devops --name myAdvAKS**
Merged "myAdvAKS" as current context in /home/chloe/.kube/config

If we repeat the preceding code for `chapter3/3-2-3_Service/3-2-3_rs1.yaml` and `chapter3/3-2-3_Service/3-2-3_service.yaml`, we should be able to achieve the same result.

Node pools

Just like GKE, a node pool is a group of VMs of the same size. At the time of writing this book, multiple node pool support is on its way. Follow the discussion at https://github.com/Azure/AKS/issues/759, or watch for the official announcement.

Azure offers virtual machine scale sets as its autoscaling group solution. In Kubernetes 1.12, VMSS support became generally available. With VMSS, the VMs can be scaled out by VM metrics. For more information, check out the official documentation: *https://kubernetes.io/blog/2018/10/08/support-for-azure-vmss-cluster-autoscaler-and-user-assigned-identity/.*

Cluster upgrade

Before you upgrade the cluster, make sure your subscription has enough resources, since nodes will be replaced by rolling deployments. The additional node will be added to the cluster. To check the quota limit, use the `az vm list-usage --location $location` command.

Let's see which Kubernetes version we're using: **# az aks show --resource-group devops --name myAKS | jq .kubernetesVersion**
"1.9.11"

The Azure CLI provides the `get-upgrades` subcommand to check which version the cluster can upgrade to: **# az aks get-upgrades --name myAKS --resource-group devops**

{
"agentPoolProfiles": [
{
"kubernetesVersion": "1.9.11",
"name": null,
"osType": "Linux",
"upgrades": [
"1.10.8",
"1.10.9"
]
}
],
"controlPlaneProfile": {
"kubernetesVersion": "1.9.11",
"name": null,
"osType": "Linux",
"upgrades": [
"1.10.8",
"1.10.9"
]
},

"id": "/subscriptions/f825790b-ac24-47a3-89b8-9b4b3974f0d5/resourcegroups/devops/providers/Microsoft.ContainerService
"name": "default",
"resourceGroup": "devops",
"type": "Microsoft.ContainerService/managedClusters/upgradeprofiles"
}

This shows that we can upgrade to versions 1.10.8 and 1.10.9. In Azure, minor versions cannot be skipped, meaning we can't upgrade from 1.9.11 to 1.11.x. We have to upgrade the cluster to 1.10 first, and then upgrade to 1.11. Upgrading from AKS is extremely easy, just like from GKE. Let's say that we want to upgrade to 1.10.9: **# az aks upgrade --name myAKS --resource-group devops --kubernetes-version 1.10.9**

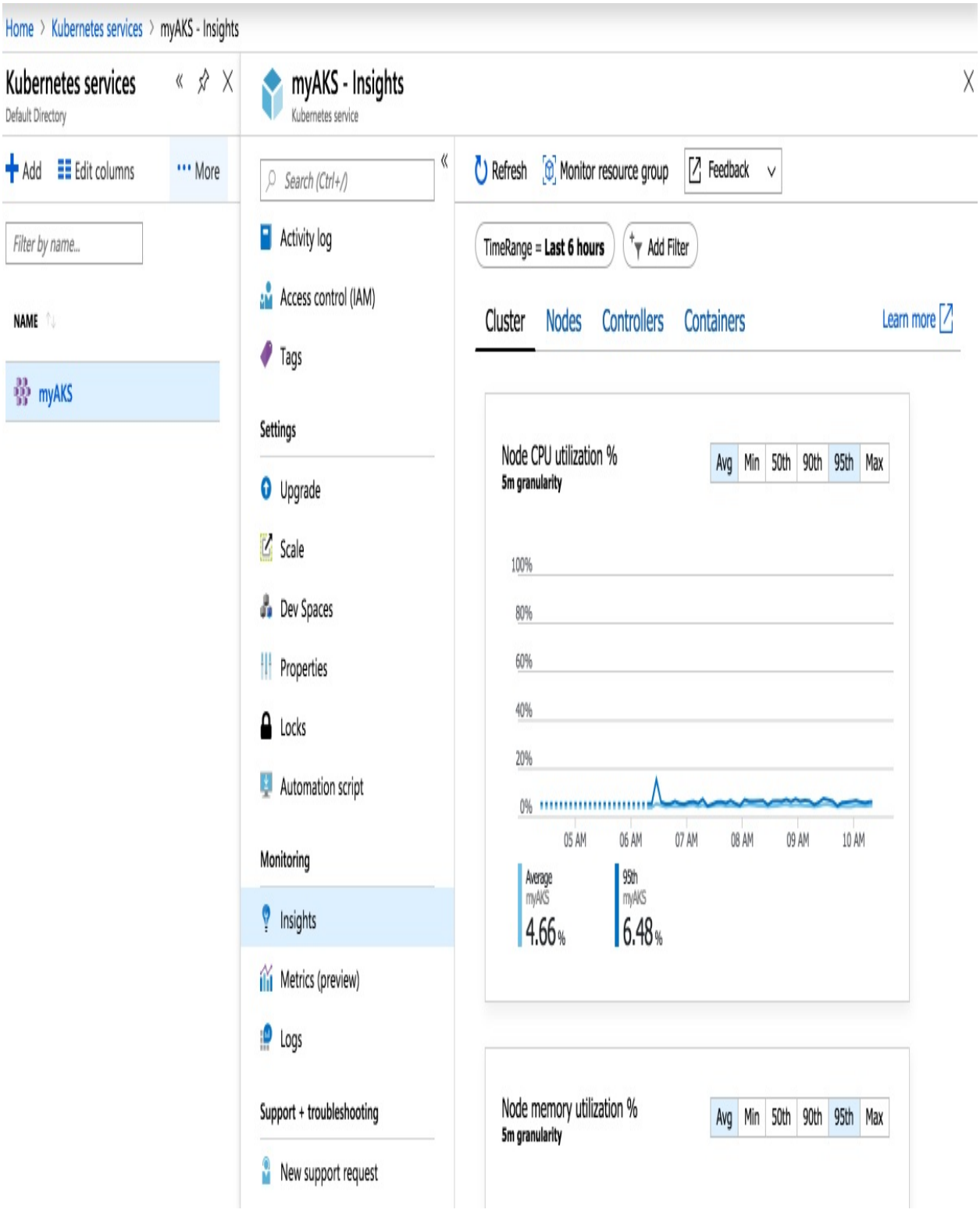
After the operation is done, we can check the current version of the cluster. The cluster has now been upgraded to the desired version: **# az aks show --resource-group devops --name myAKS --output table**
Name Location ResourceGroup KubernetesVersion ProvisioningState Fqdn
------ ---------- -------------- ------------------ ------------------ ----------------------------
myAKS centralus devops 1.10.9 Succeeded myaks-devops-f82579-077667ba.hcp.centralus.azmk8s.io

The nodes should be upgraded to 1.10.9 as well: **# kubectl get nodes**
NAME STATUS ROLES AGE VERSION
aks-nodepool1-37748545-2 Ready agent 6m v1.10.9
aks-nodepool1-37748545-3 Ready agent 6m v1.10.9

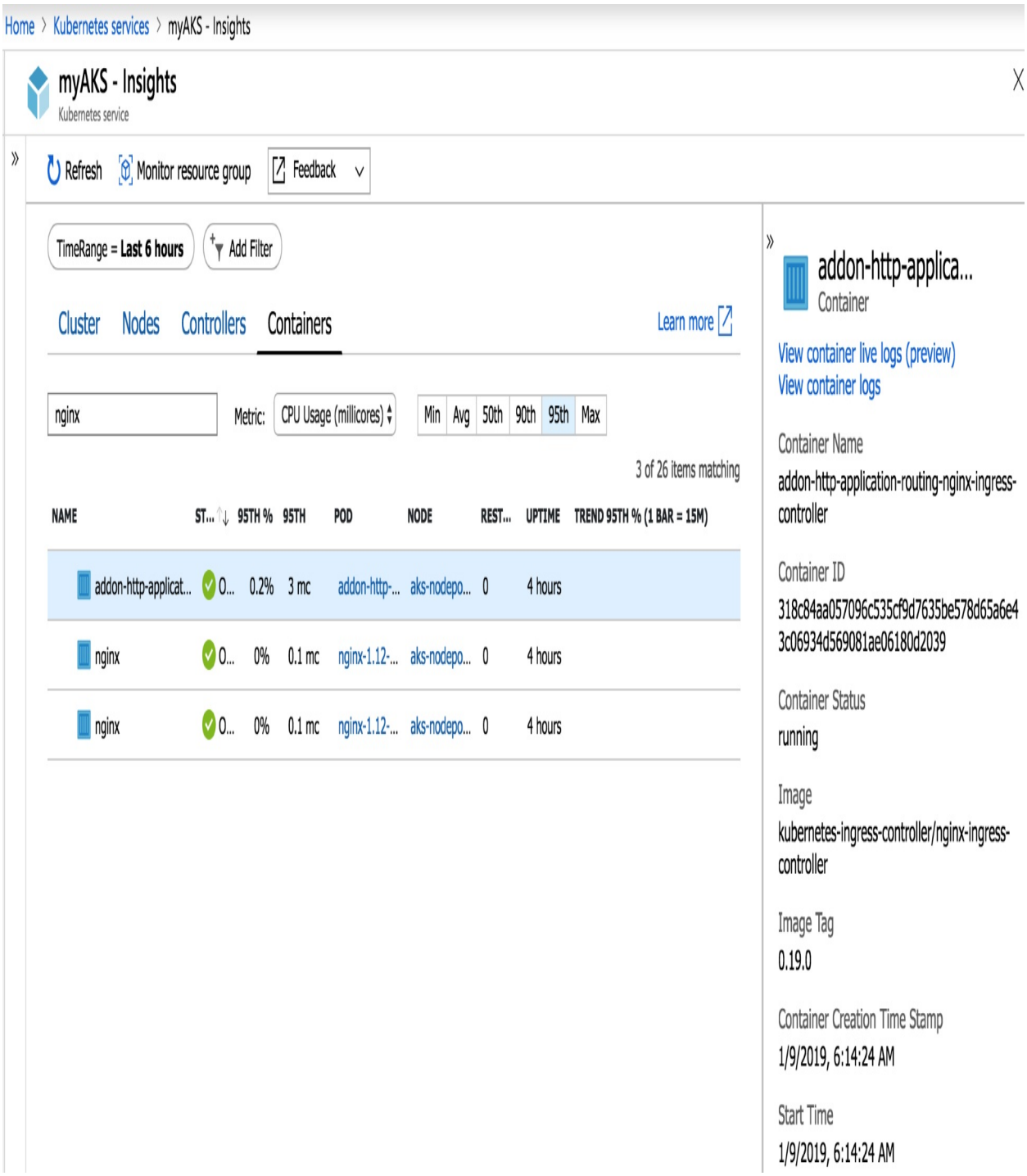
Monitoring and logging

We deployed the monitoring addons when we created the cluster. This lets us observe cluster metrics and logs easily via Azure monitoring. Let's visit the Kubernetes services in the Azure portal. You'll find the Monitoring tag with the insights, metrics, and logs sub-tabs.

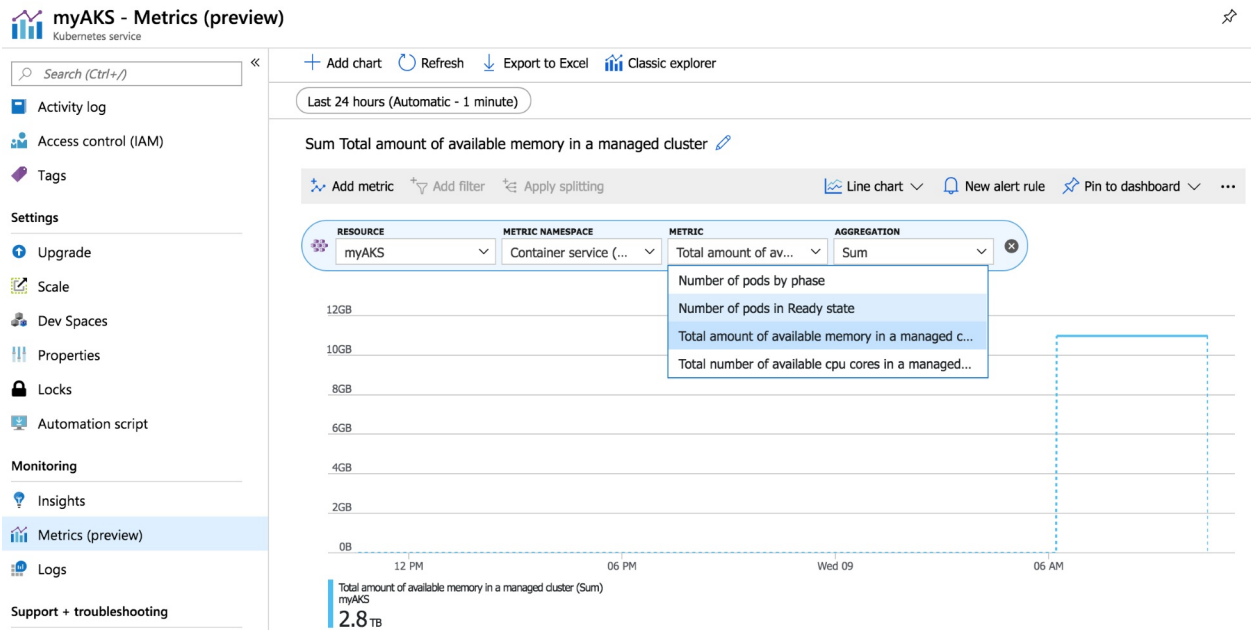
The insights page shows the general cluster metrics, such as the node CPU, the memory utilization, and the node's general health:



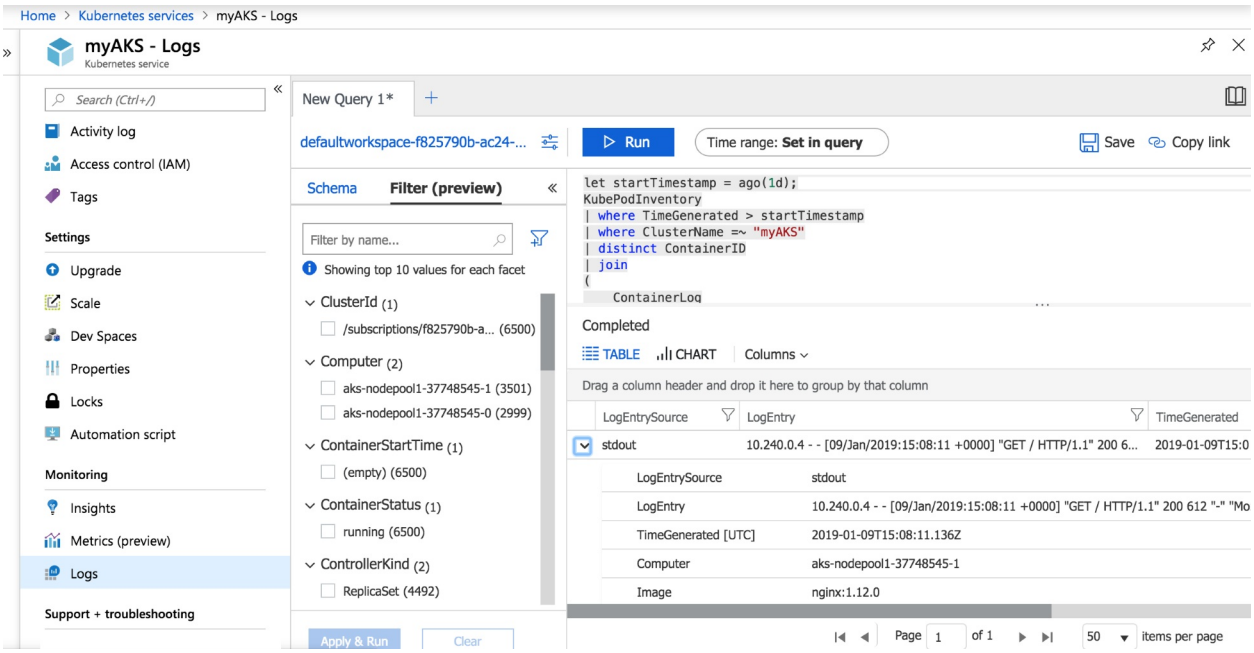
You can also observe the container information from insights:



You can find all the resource metrics that are supported on the Metrics page. For the AKS cluster, it'll show some core metrics (`https://kubernetes.io/docs/tasks/debug-application-cluster/core-metrics-pipeline/`). This is handy because it means we don't need to launch another monitoring system, such as Prometheus:



On the logs page, you can find cluster logs in Azure log analytics. You can run a query to search the logs via the supported syntax. For more information, you can find the Azure monitoring documentation at `https://docs.microsoft.com/en-us/azure/azure-monitor/log-query/log-query-overview`:


Home > Kubernetes services > myAKS - Logs
myAKS - Logs
Kubernetes service
Search (Ctrl+/)
Activity log
Access control (IAM)
Tags
Settings
Upgrade
Scale
Dev Spaces
Properties
Locks
Automation script
Monitoring
Insights
Metrics (preview)
Logs
Support + troubleshooting
New Query 1*
defaultworkspace-f825790b-ac24-...
Run
Time range: Set in query
Save
Copy link
Schema
Filter (preview)
Filter by name...
Showing top 10 values for each facet
ClusterId (1)
/subscriptions/f825790b-a... (6500)
Computer (2)
aks-nodepool1-37748545-1 (3501)
aks-nodepool1-37748545-0 (2999)
ContainerStartTime (1)
(empty) (6500)
ContainerStatus (1)
running (6500)
ControllerKind (2)
ReplicaSet (4492)
Apply & Run
Clear
let startTimestamp = ago(1d);
KubePodInventory
| where TimeGenerated > startTimestamp
| where ClusterName =~ "myAKS"
| distinct ContainerID
| join
(
ContainerLog
Completed
TABLE
CHART
Columns
Drag a column header and drop it here to group by that column
LogEntrySource
LogEntry
TimeGenerated
stdout
10.240.0.4 - - [09/Jan/2019:15:08:11 +0000] "GET / HTTP/1.1" 200 6...
2019-01-09T15:0
LogEntrySource stdout
LogEntry 10.240.0.4 - - [09/Jan/2019:15:08:11 +0000] "GET / HTTP/1.1" 200 612 "-" "Mo
TimeGenerated [UTC] 2019-01-09T15:08:11.136Z
Computer aks-nodepool1-37748545-1
Image nginx:1.12.0
Page 1 of 1
50
items per page

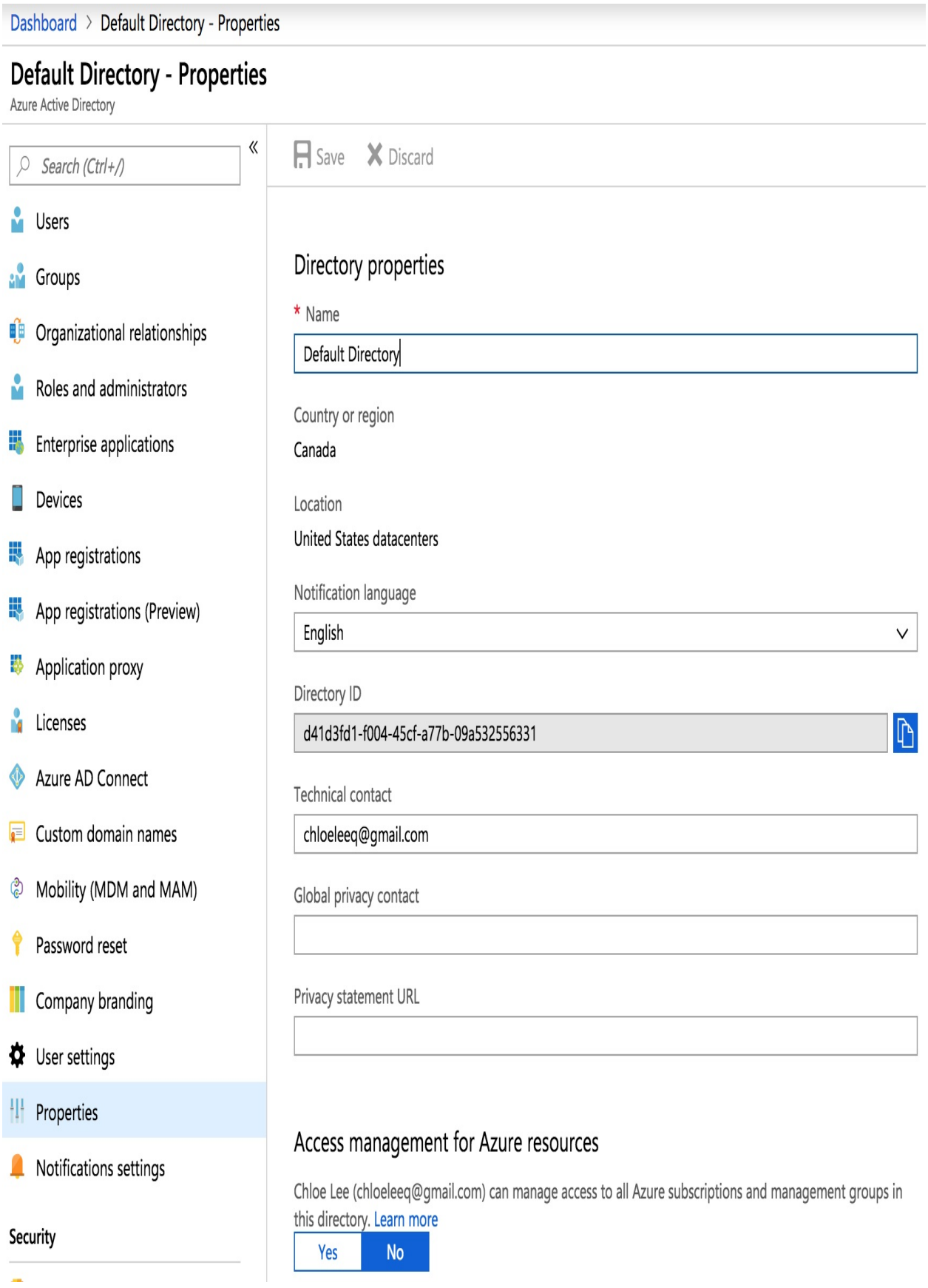
Kubernetes cloud provider

Just like other cloud providers, the cloud controller manager for Azure (https://github.com/kubernetes/cloud-provider-azure) implements a bunch of integrations with Kubernetes. Azure cloud controller manager interacts with Azure and provides a seamless user experience.

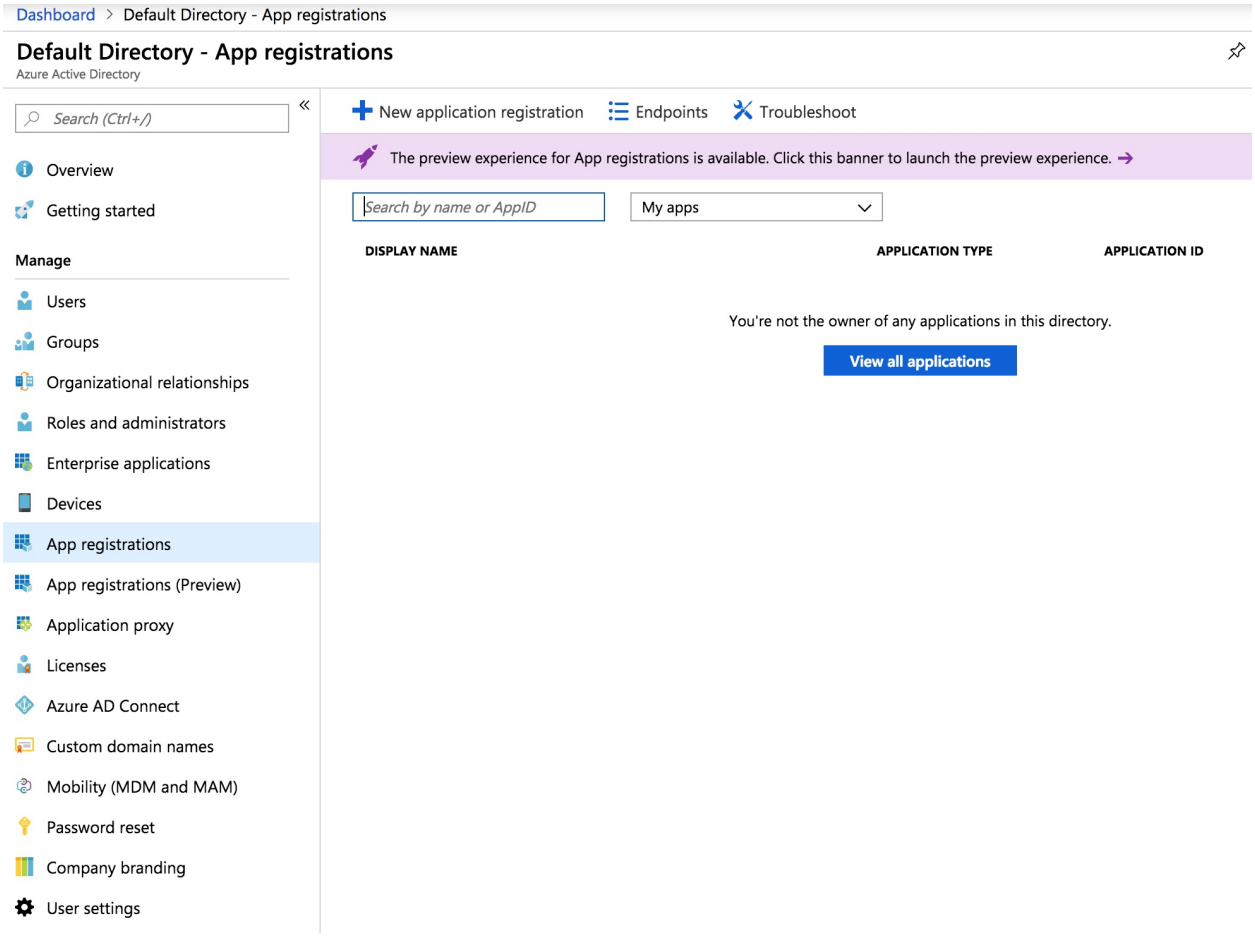
Role-based access control

AKS is integrated with Azure active directory (https://azure.microsoft.com/en-ca/services/active-directory/) via OpenID connect tokens (https://kubernetes.io/docs/reference/access-authn-authz/authentication/#openid-connect-tokens). The **Role-Based Access Control** (**RBAC**) feature can only be enabled during cluster creation, so let's start over to create an RBAC-enabled cluster.

Before creating the cluster, we'll have to prepare two application registrations in the Azure active directory first. The first acts as a server for the users' group membership. The second is like a client that integrates with kubectl. Let's go to the Azure active registry service in the Azure portal first. Go to the Properties tab and record the Directory ID. We'll need this later when we create the cluster:


Dashboard
Default Directory - Properties
Default Directory - Properties
Azure Active Directory
Search (Ctrl+/)
Save
Discard
Users
Groups
Organizational relationships
Roles and administrators
Enterprise applications
Devices
App registrations
App registrations (Preview)
Application proxy
Licenses
Azure AD Connect
Custom domain names
Mobility (MDM and MAM)
Password reset
Company branding
User settings
Properties
Notifications settings
Security
Directory properties
* Name
Default Directory
Country or region
Canada
Location
United States datacenters
Notification language
English
Directory ID
d41d3fd1-f004-45cf-a77b-09a532556331
Technical contact
chloeleeq@gmail.com
Global privacy contact
Privacy statement URL
Access management for Azure resources
Chloe Lee (chloeleeq@gmail.com) can manage access to all Azure subscriptions and management groups in this directory. Learn more
Yes
No

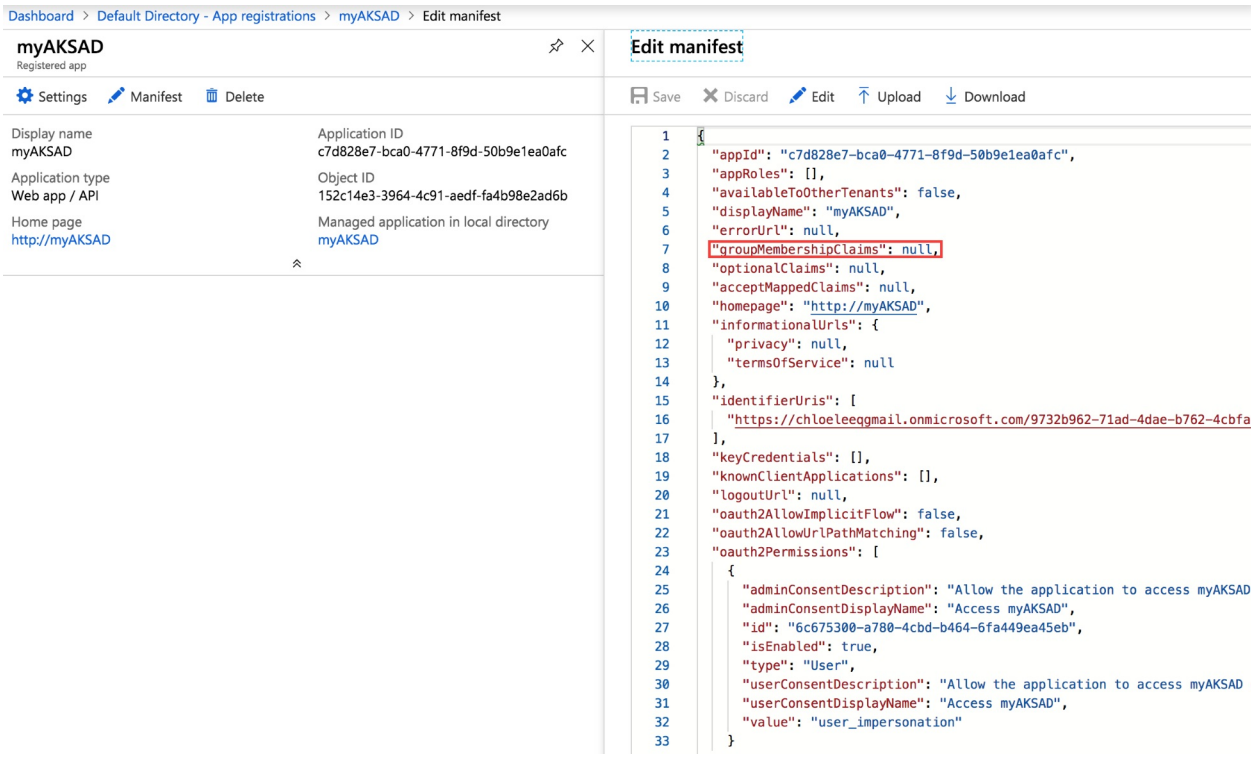
Let's go to the Application registrations of the Azure active registry service in the Azure portal first. The tab on the side shows two application registration options. One is the original design and the other is the preview version for the new console. Their functionalities are basically the same. We'll show the GA version first and the preview version after:



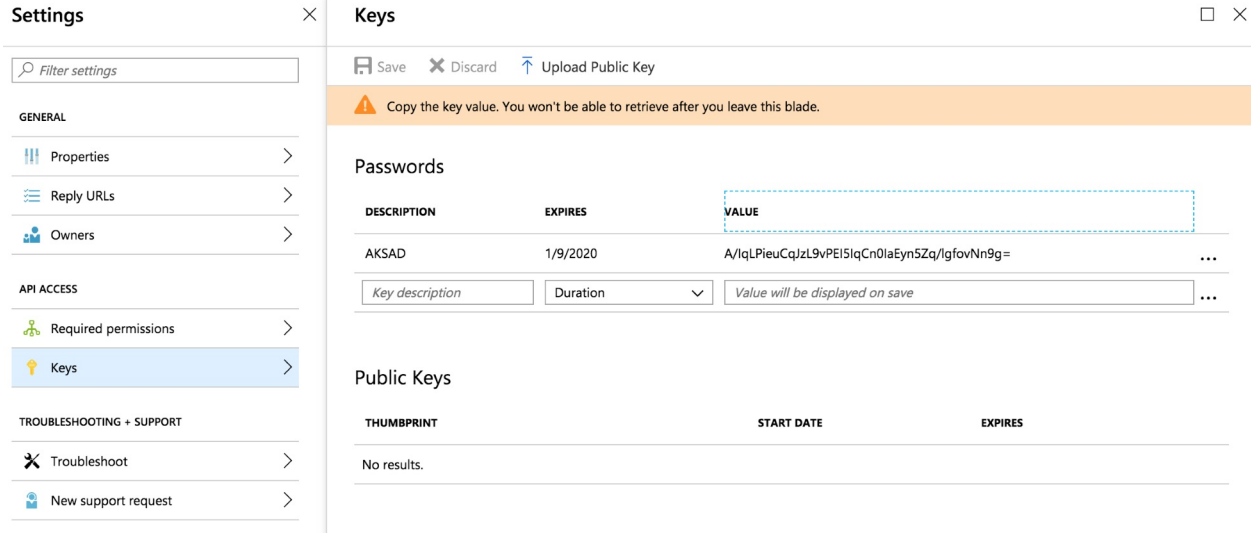
In the App registrations page, click New application registration. Add a name with any URL.

You can also try an operation via `az ad app create --display-name myAKSAD --identifier-uris http://myAKSAD`.

Here, we use `myAKSAD` as the name. After creation, we record the APPLICATION ID first. Here, we get `c7d828e7-bca0-4771-8f9d-50b9e1ea0afc`. After that, click Manifest and change the `groupMemebershipClaims` to `All`, which will get the group claims in the JWT token for all users that belong:

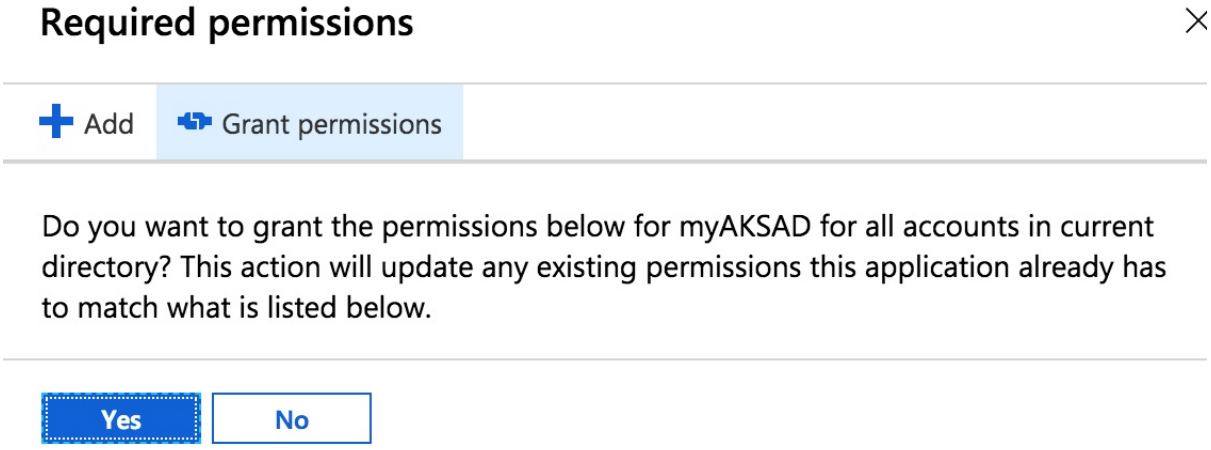


After saving the settings, go to these Settings page then the Keys page to create a key. Here, we specified expires as one year. This value was generated by the portal directly. We'll need this password later when we create the cluster:



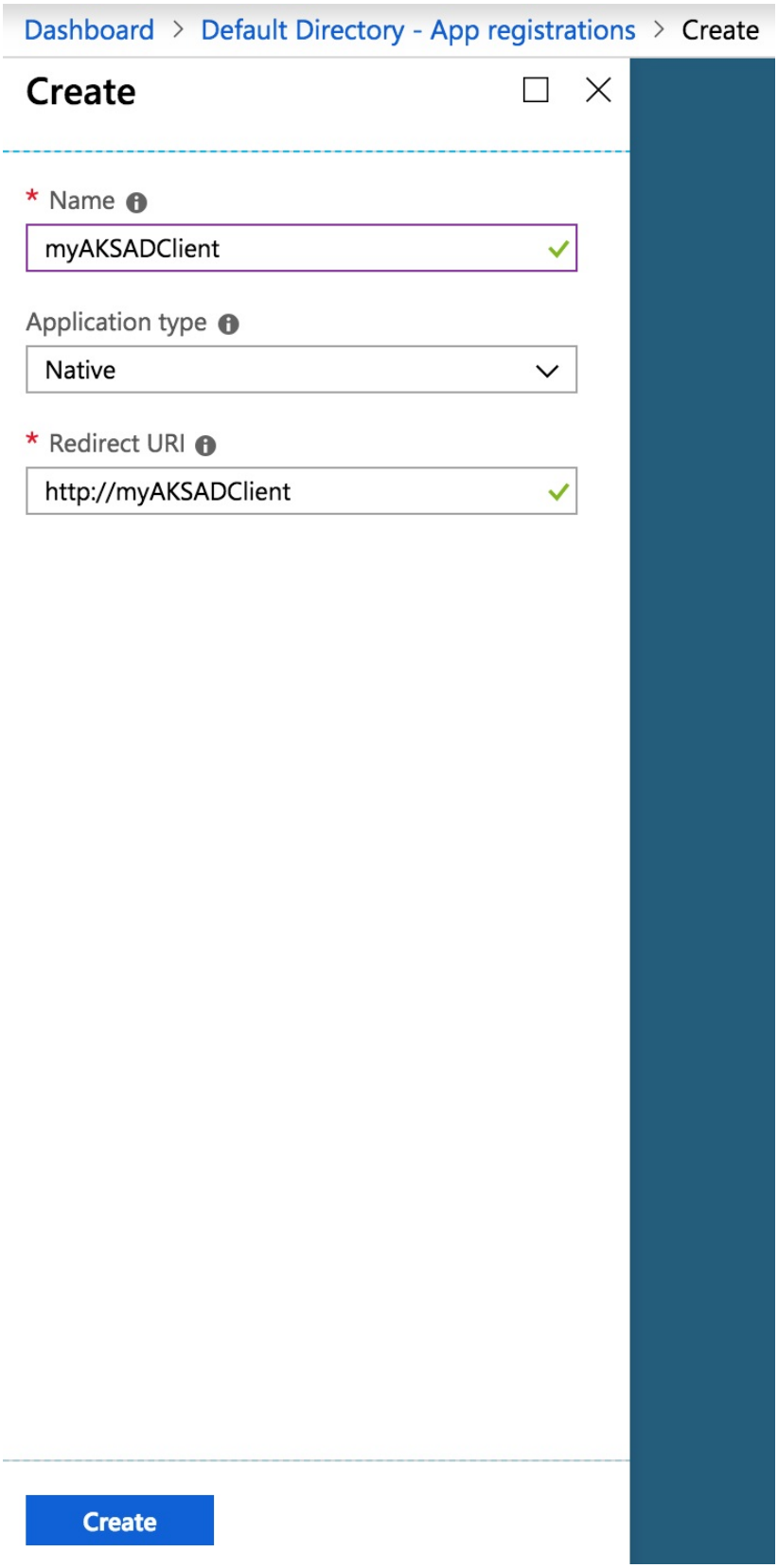
Next, we'll define a set of applications and delegated permissions for this registration. Hit the Required permissions tab and click Add, before selecting Microsoft Graph. Here, we'll select **Read directory data** under the Application permissions category and Sign in and read user profile and Read directory dat**a** under the delegated permissions so that the server can read the directory data and

verify the users. After clicking the Save button, we'll see the following screen in the Required permissions page:

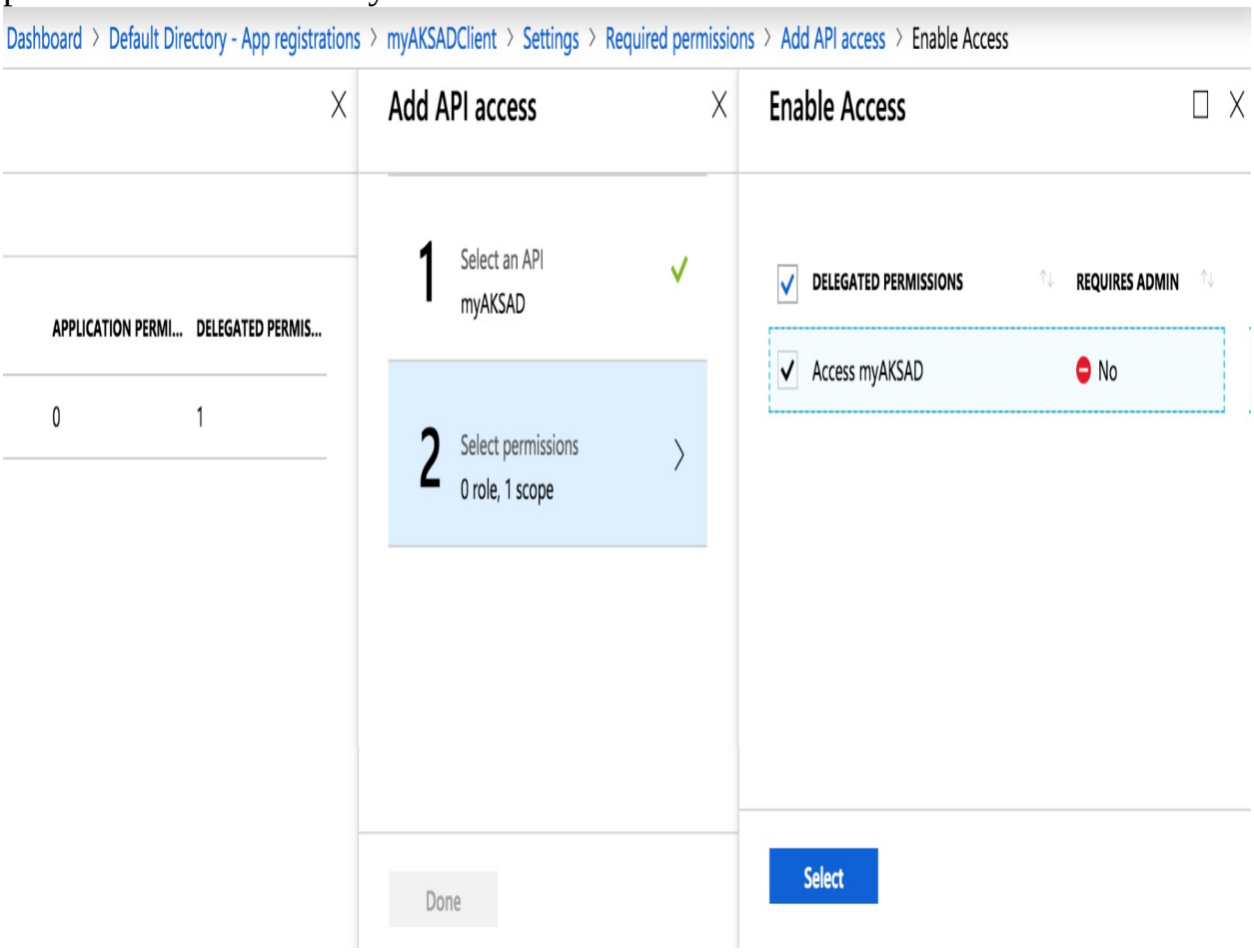


As the admin, we can grant permission for all users in this directory by clicking the Grant permission button. This will make a window pop up that will double-check this with you. Click Yes to continue.

It's now time to create a second application registration for the client. The name we used here is `myAKSADClient`. Record its Application ID after creation. Here, we get `b4309673-464e-4c95-adf9-afeb27cc8d4c`:


Dashboard > Default Directory - App registrations > Create
Create
* Name
myAKSADClient
Application type
Native
* Redirect URI
http://myAKSADClient
Create

For the required permissions, the client just needs to access the application, search the delegated permissions, and find the Access permission with the display name that you created for the server. Don't forget to hit the Grant permission button after you are done:



Right now, it's time to create our AKS cluster with Azure AD integration. Make sure that you have recorded the following information from the previous operations:

Information we recorded	**Corresponding argument in aks create**
The server application ID	`--aad-server-app-id`
The server application key (password)	`--aad-server-app-secret`
The client application ID	`--aad-client-app-id`

The directory ID	`--aad-tenant-id`

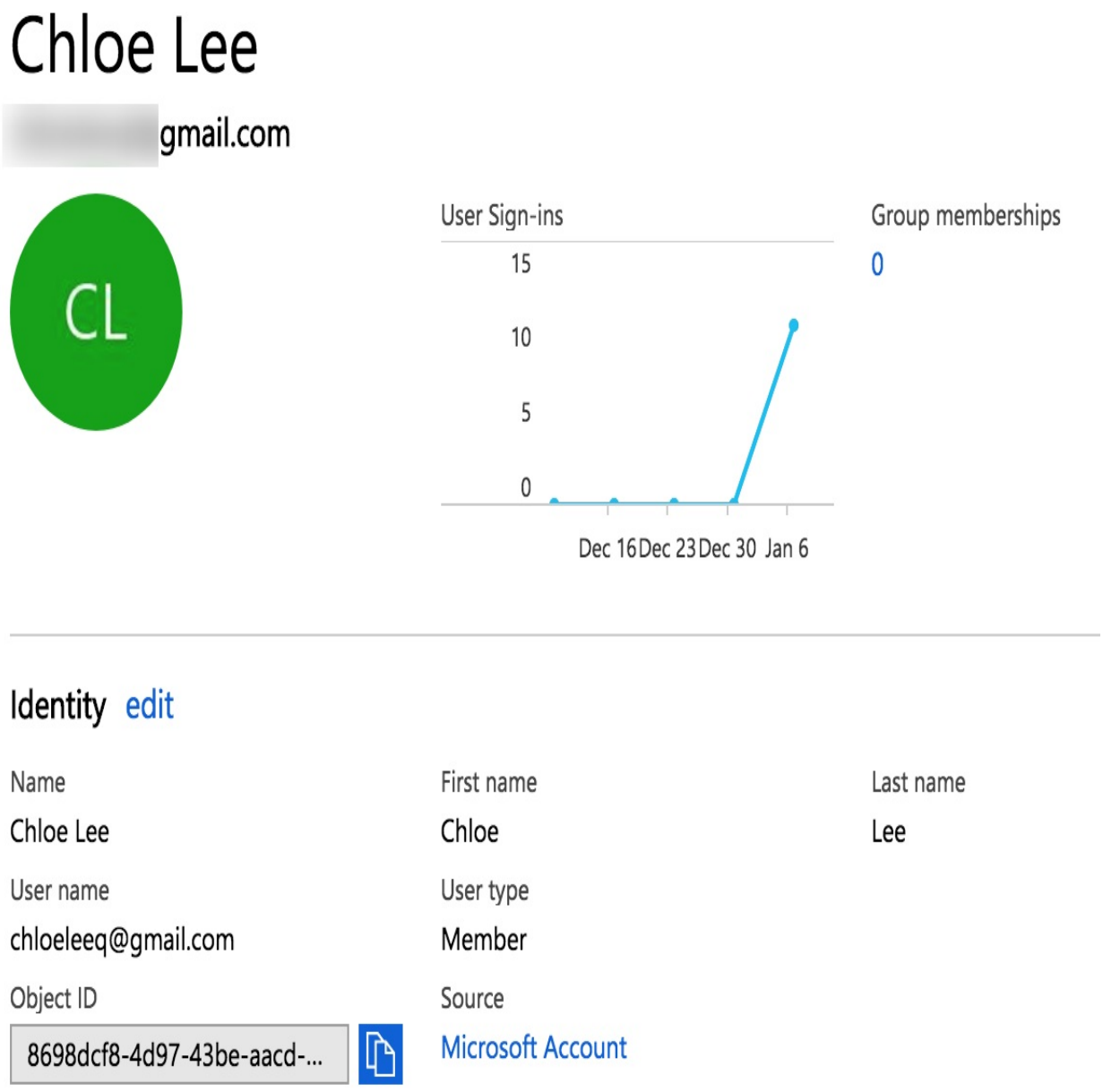
It is now time to launch the cluster: **# az aks create --resource-group devops --name myADAKS --generate-ssh-keys --aad-server-app-id c7d828e7-bca0-4771-8f9d-50b9e1ea0afc --aad-server-app-secret 'A/IqLPieuCqJzL9vPEI5IqCn0IaEyn5Zq/lgfovNn9g=' --aad-client-app-id b4309673-464e-4c95-adf9-afeb27cc8d4c --aad-tenant-id d41d3fd1-f004-45cf-a77b-09a532556331 --node-count 1 --enable-addons monitoring,http_application_routing --generate-ssh-keys**

In Kubernetes, a role binding or a cluster role binding binds the role to a group of users. A role or cluster role defines a set of permissions.

After the cluster is successfully launched, to integrate with OpenID, we'll have to create the role binding or cluster role binding first. Here, we'll use the existing cluster role in the cluster, which is `cluster-admin`. We'll bind the users to the `cluster-admin` cluster role so that the users can be authenticated and act as cluster admins: **// login as admin first to get the access to create the bindings**
 # az aks get-credentials --resource-group devops --name myADAKS --admin

// list cluster roles
 # kubectl get clusterrole
NAME AGE
admin 2h
cluster-admin 2h
...

For a single user, we'll have to find the username. You can find the Object ID for the target user under the users page in the Azure AD portal:



Use user subjects and specify the Object ID as the name, as shown here: **# cat 12-3_user-clusterrolebinding.yaml**
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
name: azure-user-cluster-admins
roleRef:
apiGroup: rbac.authorization.k8s.io
kind: ClusterRole

name: cluster-admin
subjects:
- apiGroup: rbac.authorization.k8s.io
kind: User
name: "8698dcf8-4d97-43be-aacd-eca1fc1495b6"

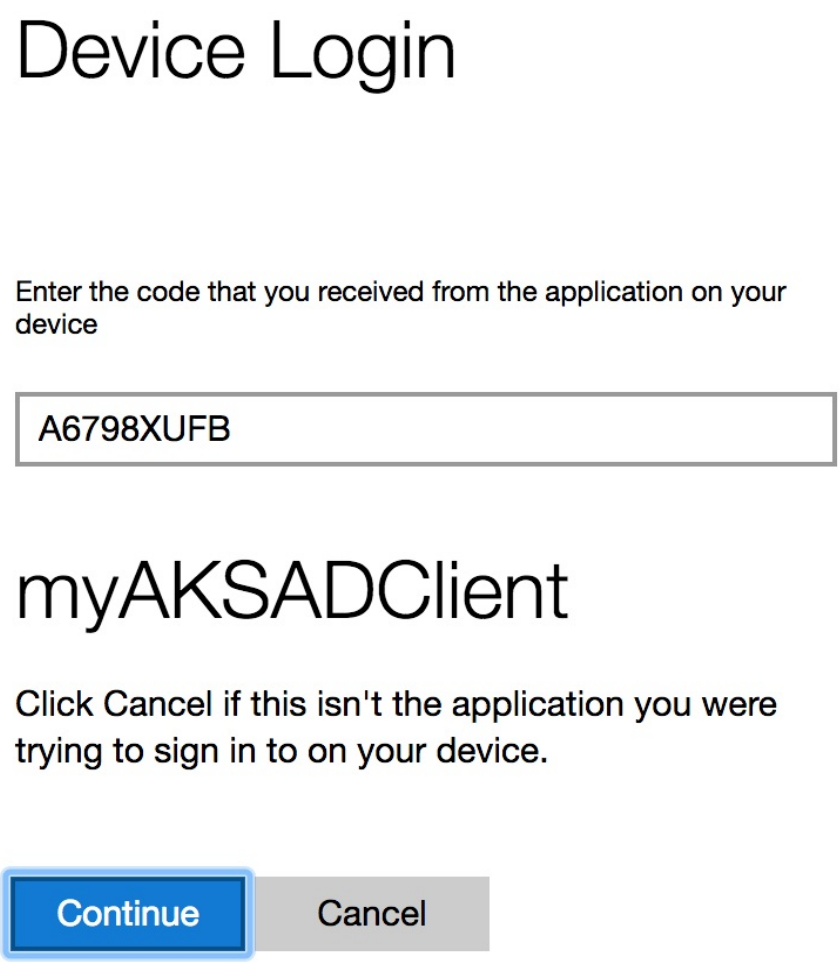
 # kubectl create -f 12-3_user-clusterrolebinding.yaml
clusterrolebinding.rbac.authorization.k8s.io/azure-user-cluster-admins created

After that, we can start over to access the cluster resources, as shown here: //
(optional) clean up kubeconfig if needed.
 # rm -f ~/.kube/config

// setup kubeconfig with non-admin
 # az aks get-credentials --resource-group devops --name myADAKS

// try get the nodes
 # kubectl get nodes
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code A6798XUFB to authenticate.

It seems like we need to log in before listing any resources. Go to the `https://microsoft.com/devicelogin` page and input the code, as the prompt suggests:



Log in to your Microsoft account and return to the terminal. The integration looks great!

```
# kubectl get nodes
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and ent
NAME STATUS ROLES AGE VERSION
aks-nodepool1-42766340-0 Ready agent 26m v1.9.11
```

Rather than specifying a set of users, you could choose to specify users as a group via the Azure AD group object ID. Replace the subject in the cluster role binding configuration so that all the users in that group will have cluster admin access: **- apiGroup: rbac.authorization.k8s.io**
kind: Group
name: "$group_object_id"

kubectl get storageclass
NAME PROVISIONER AGE
default (default) kubernetes.io/azure-disk 4h
managed-premium kubernetes.io/azure-disk 11h

kubectl describe storageclass default
Name: default
IsDefaultClass: Yes
Annotations: ...
,storageclass.beta.kubernetes.io/is-default-class=true
Provisioner: kubernetes.io/azure-disk
Parameters: cachingmode=None,kind=Managed,storageaccounttype=Standard_LRS
AllowVolumeExpansion: <unset>
MountOptions: <none>
ReclaimPolicy: Delete
VolumeBindingMode: Immediate
Events: <none>

kubectl describe storageclass managed-premium
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"

id="kobo.25.1">Name: managed-premium
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.26.1">IsDefaultClass: No
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.27.1">Annotations: ...
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.28.1">Provisioner: kubernetes.io/azure-disk
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.29.1">Parameters:
kind=Managed,storageaccounttype=Premium_LRS
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.30.1">AllowVolumeExpansion: <unset>
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.31.1">MountOptions: <none>
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.32.1">ReclaimPolicy: Delete
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.33.1">VolumeBindingMode: Immediate
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.34.1">Events: <none>

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.40.1">// create storage class with Premium and Standard ZRS

<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.41.1"># cat chapter12/storageclass.yaml

<span xmlns="http://www.w3.org/1999/xhtml"
class="koboSpan" id="kobo.42.1">kind: StorageClass
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.43.1">apiVersion: storage.k8s.io/v1
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.44.1">metadata:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.45.1">
name: fastzrs
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.46.1">provisioner: kubernetes.io/azure-disk
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.47.1">parameters:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.48.1">
skuName: Premium_ZRS
<span

xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.49.1">
location: centralus

<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.50.1">---

<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.51.1">kind: StorageClass
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.52.1">apiVersion: storage.k8s.io/v1
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.53.1">metadata:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.54.1">
name: slowzrs
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.55.1">provisioner: kubernetes.io/azure-disk
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.56.1">parameters:
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.57.1">
skuName: Standard_ZRS
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.58.1">
location: centralus

<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.59.1">//
create the storage class
<span
xmlns="http://www.w3.org/1999/xhtml" class="koboSpan" id="kobo.60.1">#
kubectl create -f chapter12/storageclass.yaml
<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.61.1">storageclass.storage.k8s.io/fastzrs created

<span xmlns="http://www.w3.org/1999/xhtml" class="koboSpan"
id="kobo.62.1">storageclass.storage.k8s.io/slowzrs created

The mapping of the `skuName` field and the storage types is shown here:

Class	**Replication**	**skuName**
Premium	LRS	`Premium_LRS`
Premium	ZRS	`Premium_ZRS`
Standard	GRS	`Standard_GRS`
Standard	LRS	`Standard_LRS`
Standard	RAGRS	`Standard_RAGRS`
Standard	ZRS	`Standard_ZRS`

L4 LoadBalancer

When you create an AKS cluster in basic networking, Azure will provision a load balancer called kubernetes in a dedicated resource group. When we create a service with a LoadBalancer type, AKS will provision a frontend IP and link to the service object automatically. This is how we could access `nginx` using external IPs earlier in this chapter. There are a set of annotations you can set in your service, and the Azure cloud controller will provision it based on the specified annotations. You can find a list of supported annotations here: https://github.com/kubernetes/kubernetes/blob/master/pkg/cloudprovider/providers/azure/azure_loadbalancer.go.

// we could find a set of resources starting with the name addon-http-application-routing-...
kubectl get pod -n kube-system
NAME READY STATUS RESTARTS AGE
addon-http-application-routing-default-http-backend-6584cdnpq69 1/1 Running 0 3h
addon-http-application-routing-external-dns-7dc8d9f794-4t28c 1/1 Running 0 3h
addon-http-application-routing-nginx-ingress-controller-78zs45d 1/1 Running 0 3h
...

// search the routing zone name.
az aks show --resource-group devops --name myADAKS --query addonProfiles.httpApplicationRouting.config.HTTPApplicationRoutingZoneNam
"46de1b27375a4f7ebf30.centralus.aksapp.io"

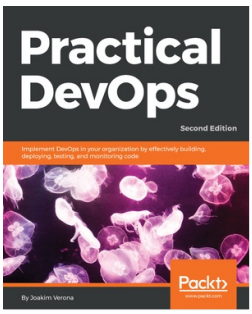
Summary

Microsoft Azure is a powerful and enterprise-grade cloud computing platform. Beside AKS, it also provides various services in different fields, such as analytics, virtual reality, and much more. In this chapter, we touched the surface of Azure virtual network, subnets, and load balancing. We also learned how to deploy and administrate the Kubernetes service in Azure. We walked through how Azure provides Kubernetes resources via the cloud controller manager for Azure. We got to know how the cloud controller manager for Azure provides a seamless experience for Azure users, such as by creating an Azure load balancer when a LoadBalancer service in Kubernetes is requested or pre-creating an Azure disk storage class.

This is the last chapter of this book. We have tried to walk through both basic and more advanced concepts in this Kubernetes learning journey. Because Kubernetes is evolving rapidly, we strongly encourage you to join the Kubernetes community (`https://kubernetes.io/community/`) to get inspired, discuss, and contribute!

Other Books You May Enjoy

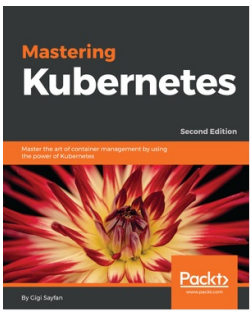
If you enjoyed this book, you may be interested in these other books by Packt:



Practical DevOps - Second Edition
Joakim Verona

ISBN: 9781788392570

- Understand how all deployment systems fit together to form a larger system
- Set up and familiarize yourself with all the tools you need to be efficient with DevOps
- Design an application suitable for continuous deployment systems with DevOps in mind
- Store and manage your code effectively using Git, Gerrit, Gitlab, and more
- Configure a job to build a sample CRUD application
- Test your code using automated regression testing with Jenkins Selenium
- Deploy your code using tools such as Puppet, Ansible, Palletops, Chef, and Vagrant



Mastering Kubernetes - Second Edition

Gigi Sayfan

ISBN: 9781788999786

- 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

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!