

KUBERNETES

SUCCINCTLY

BY **RAHUL RAI AND
TARUN PABBI**

Kubernetes Succinctly

By

Rahul Rai and Tarun Pabbi

Foreword by Daniel Jebaraj



Copyright © 2019 by Syncfusion, Inc.

2501 Aerial Center Parkway

Suite 200

Morrisville, NC 27560

USA

All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey

Copy Editor: Courtney Wright

Acquisitions Coordinator: Tres Watkins, content development manager, Syncfusion, Inc.

Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the <i>Succinctly</i> Series of Books.....	9
About the Authors	11
Before We Begin	13
Chapter 1 Hello Kubernetes	14
Kubernetes	15
A brief history	15
Kubernetes architecture	16
Cluster.....	18
Master	18
REST API	20
Etcd	20
Controller manager	20
Scheduler	20
Scheduling workflow.....	20
Nodes.....	22
Pods	23
DNS.....	25
Proxy	25
Kubelet	25
Node supervisor.....	25
Logging agent.....	26
Replica set	26
Services	26
Why do we need services?	26

Types of services	27
Daemon set	27
Deployment	28
Volume	28
Stateful set	28
Namespace	29
Labels and annotations	29
Names	30
Secrets	30
Use cases	31
Summary	31
Chapter 2 Installation.....	32
Setting up a cluster using kubeadm	32
Prerequisite	33
Setting up the Kubernetes master	34
Adding pod network add-on	40
Adding worker nodes	41
Setting up a cluster on a developer machine	43
Prerequisite	43
Kubernetes installation.....	44
Summary	45
Chapter 3 Deploying Applications	46
Requirement.....	46
Remind Me logical design	46
The first release.....	48
Deconstructing YAML configurations.....	55

Summary	58
Chapter 4 Stateful Microservices	59
Stateful sets	59
Volumes	62
Persistent volume and persistent volume claim	63
Adding persistent volume to our service	64
Create persistent volume	64
Create persistent volume claim	65
Map persistent volume claim to the application	65
Adding secrets to an application	68
Kubernetes abstractions	71
Interacting with the Kubernetes DNS service	75
Summary	79
Chapter 5 Upgrades and Scaling	80
Application upgrades	80
Rolling updates	80
Blue-green deployments	85
Scaling	86
kubectl autoscale	86
Horizontal Pod Autoscaler (HPA)	87
HPA commands	88
Limitation	89
Summary	89
Chapter 6 Jobs	90
Job specification	90
Job types	91

Nonparallel jobs	92
Parallel jobs with a fixed completion count.....	92
Parallel jobs with a work queue.....	92
Cron jobs.....	92
Summary.....	95
Chapter 7 Monitoring and Logging.....	96
Kubernetes Dashboard.....	96
Installation	96
Monitoring through Prometheus	97
Installation	97
Connecting Prometheus	100
Centralized logging.....	101
Elasticsearch	102
Kibana	102
Fluentd	103
Installation	103
Summary.....	104
Chapter 8 Packaging Applications with Helm.....	105
Introduction	105
The anatomy of a chart.....	105
Installation	106
Creating charts.....	107
Deploying charts.....	111
Debugging charts	112
Packaging charts.....	113
Summary.....	116

Chapter 9 Next Steps	117
Advanced Kubernetes networking	117
Kubernetes and serverless	117
Kubernetes automation	118
Keel	118
Buddy	118
Kubernetes and Windows.....	118
Kubernetes Kustomize	119
Kubernetes security.....	119
Kubernetes on cloud	120
Kubernetes communities	121
Summary.....	121

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Authors

Rahul Rai

Rahul is a technology consultant focusing on cloud technologies in Sydney, Australia, with over 10 years of work experience.

He has been at the forefront of cloud consulting for businesses around the world. He has worked in the capacities of developer, consultant, and architect for enterprises ranging from small startups to multinational corporations. He worked for over five years with Microsoft, where he delivered innovative solutions on Microsoft Azure for several enterprise customers and governments. At Microsoft, he was a Worldwide Subject Matter Expert in Microsoft Azure. He loves contributing to community initiatives and speaks at some of the well-known conferences.

Rahul is a contributor to Azure open-source initiatives and has authored MSDN articles and publications on Azure. He writes and maintains a blog focused on cloud technologies on [The Cloud Blog](#).

“I dedicate this book to the wonderful women in my life who made me what I am today. My dear mother, Mrs. Indra Rai, my supportive wife, Neha, my amazing sister, Mohini, and my lovely little niece, Advika. Thank you for your kindness and endless support. This book is a result of your selfless efforts.”

Tarun Pabbi

Tarun works as a cloud consultant in Gurgaon, India, with over 11 years of work experience.

As a cloud consultant, he works on large-scale and complex IT engagements for enterprises and helps them build new applications as well as migrate existing applications to the cloud. He worked for over four years with Microsoft, where he was primarily working on Microsoft Azure solutions. He was Microsoft Azure SME for Microsoft Services Worldwide and Microsoft Services India. He has also worked with an augmented reality startup, Foyr.com, to build an automated interior designer product for real estate developers. His interests include DevOps, machine learning, and identity and access management. He has worked extensively on cloud computing and IoT applications and has extensive experience working on cloud platforms including Microsoft Azure, AWS, and GCP.

He blogs at <https://www.tarunpabbi.com>.

“I dedicate this book to my parents and to my wife, Sneha, who made me what I am today. A big thank you, guys, for your support and understanding.”

Acknowledgment

Many people behind the scenes at Syncfusion helped publish this book. A special thanks to Tres Watkins for having the vision to get this book started, and to Jacqueline Bieringer for helping us throughout the journey of authoring this book. We would also like to thank everyone on the editorial, production, and marketing teams. Many thanks to our technical reviewer James M who worked with us through the multiple revisions of the draft.

In our careers, we met people who believed in our capabilities and gave us the opportunities that helped us develop into people capable of writing this book. Our many thanks to Paul Glavich (principal consultant, Telstra) and Atul Malaviya (principal program manager, Microsoft), who are our mentors and reviewers of this book. They spent many hours reading the drafts and provided us with valuable feedback that helped ensure that this book meets the requirements of its audience.

Finally, this book would not have been possible without our families, who supported us throughout our authoring journey. We are advocates of diversity in technology, and therefore, through this book, we want to cherish the women across the world who are our counterparts in the technology workforce.




Before We Begin

This book is your guide to learning Kubernetes and leveraging its many capabilities for developing, validating, and maintaining your applications. We will focus on examples and samples to familiarize you with Kubernetes and its core concepts. Using the examples in this book, you will be able to perform common tasks and take complete advantage of the Kubernetes platform.

As much as it's important to know what this book covers, it's also essential to know what this book does not cover. Kubernetes is a container orchestration platform, and one of the most famous container technologies is Docker. We are not going to discuss Docker, and I will assume that you have a basic familiarity with the Docker platform. You should be able to create Docker containers, understand and write Docker files, and perform common container tasks such as managing container images, deleting containers, and recreating containers.

If you feel that you are not very comfortable with Docker or want to recap core Docker concepts quickly, the *Succinctly* series has an excellent title in its catalog, [Docker Succinctly](#), written by Elton Stoneman.

Other resources associated with this book are available at the following locations.

	Code https://github.com/Kubernetes-Succinctly
	Docker Hub https://hub.docker.com/u/kubernetessuccinctly
	Helm Repo https://github.com/Kubernetes-Succinctly/helmrepo

Welcome to *Kubernetes Succinctly*—We hope that you enjoy reading this book.

Chapter 1 Hello Kubernetes

Organizations all over the world compete on technology, regardless of the sector to which they belong. Every organization wants to make the best use of their infrastructure investments in public or local clouds. However, deploying applications densely on servers to optimally utilize the available infrastructure comes at the cost of management. The unit of deployment in a datacenter started with bare metal, moved on to virtual machine, and is now a container. A fine-grained unit of deployment abstracts the application from the underlying infrastructure, which provides a considerable advantage to application developers: they can treat the underlying infrastructure as immutable. If a server dies, crashes, or becomes unresponsive, no one sheds a tear, and the server simply gets replaced.

Managing multiple applications spread across containers hosted on various virtual machines is a complex task. There are several tools that we can use to manage the containers and the applications hosted by them. Some of the most common tools that are well known and trusted by organizations are Docker Swarm, Kubernetes, and Mesosphere DC/OS. Out of all the tools, Kubernetes is evolving as the de facto container management tool used by organizations and cloud vendors all over the world. However, all the tools fulfill the basic need of automatically placing a user application on infrastructure that can best host the application and manage its lifecycle. The orchestration capabilities of these tools free the Operations (or Ops) team from managing application concerns such as replication, availability, and fault tolerance. Since the orchestrator intimately knows the infrastructure, it ensures optimal utilization and highly efficient use of resources.

Before we proceed, let's visualize the architecture of a datacenter capable of running containerized applications, and see where the orchestrator tools fit into the overall architecture.

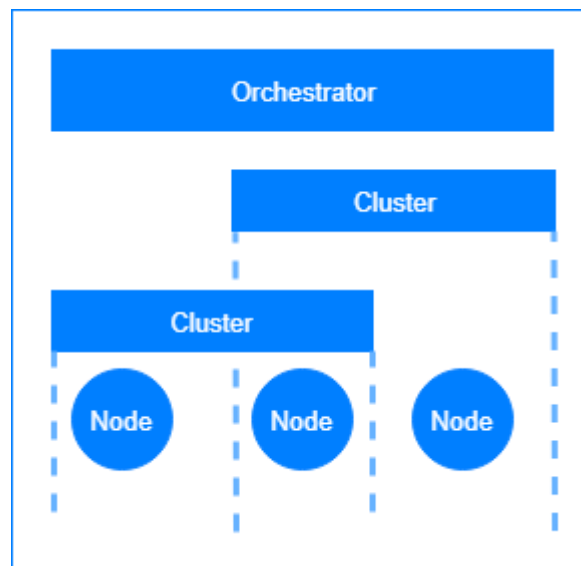


Figure 1: Datacenter architecture for containerized applications.

In distributed systems such as Kubernetes, the workloads are spread across all the available compute units, which are many virtual machines working together. The orchestrator tools themselves can run on a variety of platforms, and therefore, they can support the containerized applications running on any platform.

A node is a physical or virtual machine that can host an application and run some of the orchestrator services. A cluster is a virtual collection of network-connected nodes. Some of the most common clusters deployed in modern data centers are the failover clusters and the load-balancing clusters; however, they can also be purposed to host different application environments, such as development, testing, and production.

Kubernetes

Kubernetes is a container orchestration application developed by Google. It provides a unified API that can be used to manage the deployment and upgrades of containerized applications, batch jobs, and databases. Kubernetes provides a layer of abstraction and isolation to applications from the underlying hardware by deploying them in containers. Kubernetes uses a declarative approach to deploying applications. As a user, you write the application requirements (such as networking, ports, memory, and replication) in a manifest, and hand it over to Kubernetes. Kubernetes then reads this manifest and decides where to place the nodes that run your application in the cluster.



Note: Kubernetes is pronounced as “koo-ber-net-eez,” and is sometimes referred to as K8s (8 because it replaces 8 characters between the K and the S). Kubernetes is the Greek name for a helmsman or captain of a ship, and like a captain, Kubernetes manages containerized applications. The official website of the Kubernetes project is [here](#). The majority of Kubernetes code is written in the Go programming language.

Because of its excellent orchestration and routing capabilities, Kubernetes is an enterprise-grade platform for building microservices applications. Since Kubernetes decides the placement of microservices, it has inbuilt features to support service discovery. Using the service discovery feature of Kubernetes, microservices can talk without knowing each other's physical location. Behind the scenes, Kubernetes is always working to keep the cluster in the desired state, maintaining required replica count and monitoring service health by taking actions like replacing nodes with new ones and providing a holistic view of the cluster to the administrator. Kubernetes also supports some of the most well-known microservice deployment workflows out of the box, such as rolling updates, blue-green deployment, and canary deployments.

A brief history

Google has a massive infrastructure and application footprint. In the years 2003–2004, Google internally introduced a large-scale cluster management system named [Borg](#). Borg can run hundreds of thousands of jobs from several thousand applications in clusters composed of tens of thousands of machines.

After Borg, Google released project [Omega](#) in 2013. Omega, which is a flexible and scalable scheduler for large compute clusters, was the next version of the Borg cluster management system. In 2015, Google launched [Kubernetes](#), which is an open-source version of Borg. In the same year, several industry leaders, such as Microsoft, IBM, and Docker, joined the Kubernetes community. A year later, Google donated Kubernetes to the [Cloud Native Computing Foundation \(CNCF\)](#), which maintains high-quality projects such as Kubernetes and Prometheus, and aims to build a sustainable community and ecosystem around microservices projects that use containers.

Since 2016, Kubernetes has gone mainstream. Deployment of several large-scale applications on Kubernetes clusters, such as Pokemon Go! and GitHub, added to the popularity of the platform. We are now seeing almost all major cloud vendors provide native support for Kubernetes on the cloud, and several projects cropping up in the Kubernetes ecosystem.

Kubernetes architecture

Before we jump into the exciting topic of building and deploying applications with Kubernetes, let's go through a short Kubernetes 101 lesson. In this section, we will look at the components that make up a Kubernetes cluster. The following diagram illustrates how the individual components in the Kubernetes architecture work together.

The master component of Kubernetes is responsible for allocating workloads to nodes and managing the cluster state so that the cluster always remains in the desired state. For a Kubernetes cluster, the desired state means infrastructure and application requirements, such as which applications the user wants to run on a cluster, which containers the application should use, and which network and disk resources the user wants to make available to the application. The nodes, on the other hand, run the actual workloads on containers. Next, we will discuss the various components that make up the Kubernetes system, further divided into different categories.

Cluster

Clusters are collections of computing hosts, storage, and network resources that Kubernetes can use to deploy and manage your applications. In general, very large-scale applications sweep across clusters, and Kubernetes can manage such applications by federating across clusters. Federated clusters are outside the scope of this book, but you can read more about them on the [Kubernetes website](#).

Master

The Kubernetes master makes up the control plane of Kubernetes. It is made up of several components, such as the API server, a scheduler, a controller manager, and a cluster-configuration storage, which is generally implemented using **etcd**—a consistent and highly available key value store that stores Kubernetes cluster data. The following is a high-level architecture diagram of the Kubernetes master and its components.

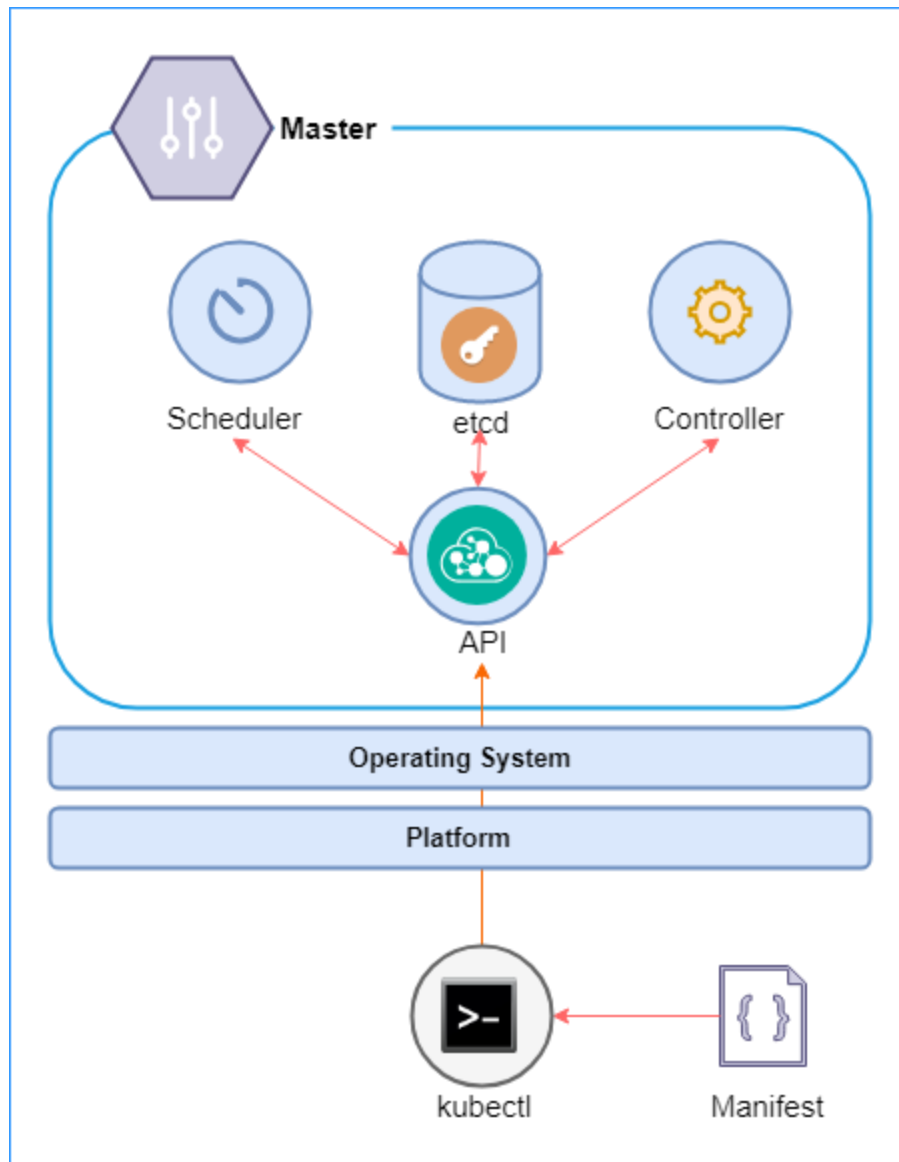


Figure 3: Kubernetes master

We will discuss these components in a little more detail in the following sections. However, at a high level, the master schedules the pods and handles cluster events, such as a health check. The master can be deployed on multiple nodes to make the cluster highly available. The control plane does not host workloads, and it is solely responsible for maintaining the cluster. The master is supported by the underlying operating system (Linux), and it can be hosted on a variety of platforms on the cloud or premises.

All interactions with the master take place through the Kubernetes REST API, which makes it easy to build custom tooling for managing Kubernetes clusters. Users can use the **kubectl** tool, a REST CLI (Command Line Interface) client, to send a manifest with the desired changes to the API server. The desired state rules then kick off, which leads to the scheduler modifying the pods to bring the state of the cluster to the state definition provided in the manifest.

REST API

Any interaction of tools or developers with Kubernetes takes place through the Kubernetes API, the **kube-apiserver**. Interact with this API using the command line interface (CLI) called **kubectl**, which is a simple client built for simplifying interaction with the Kubernetes API. The API stores the state data in the cluster state store: the **etcd** cluster. The API is stateless, so it can be scaled out on demand.

Etcd

Etcd is a reliable, distributed persistent data store that is used by the API server for storing the cluster state and configuration. The **etcd** cluster store is a distributed, always consistent, and watchable data store. Its unusual name is derived from the Linux **etc** directory, plus the word “distributed.” The cluster store is the single source of truth for the cluster and, therefore, you should plan to make the storage redundant and always accessible for highly available clusters. In a small cluster, an instance of **etcd** can run on the master node. However, for nontrivial deployments in which reliability and availability are essential, it is recommended to maintain an odd-numbered count of **etcd** cluster nodes, such as 3 or 5.

Controller manager

The controller manager, **kube-controller-manager**, is the management engine of Kubernetes. It is a daemon that is made up of multiple managers that control replication, pods, services, endpoints, and so on. The managers watch their respective resources through the Kubernetes API and try to direct the cluster to the desired state in case of any inconsistency.

The cloud variant of the controller manager, **cloud-controller-manager**, manages the nodes in the cloud. The cloud controller manager uses vendor-specific APIs to manage the nodes while implementing a standard interface. Therefore, your applications won't be impacted by different implementations of the controller manager.

Scheduler

The Kubernetes scheduler, **kube-scheduler**, is responsible for scheduling (deploying) pods into nodes, which is a physical or virtual machine, based on the desired configuration that you specify in the application deployment manifest. A pod can host multiple containers and is the smallest unit of deployment in Kubernetes. The scheduler considers constraints such as resource requirements, affinity requirements, service requirements, and data localization for scheduling the pods.

Scheduling workflow

To understand how the components of the masters work together, let's look at the workflow that is followed for provisioning a new pod and assigning it an application workload. The following sequence diagram illustrates the actions that are invoked to create a pod for an application.

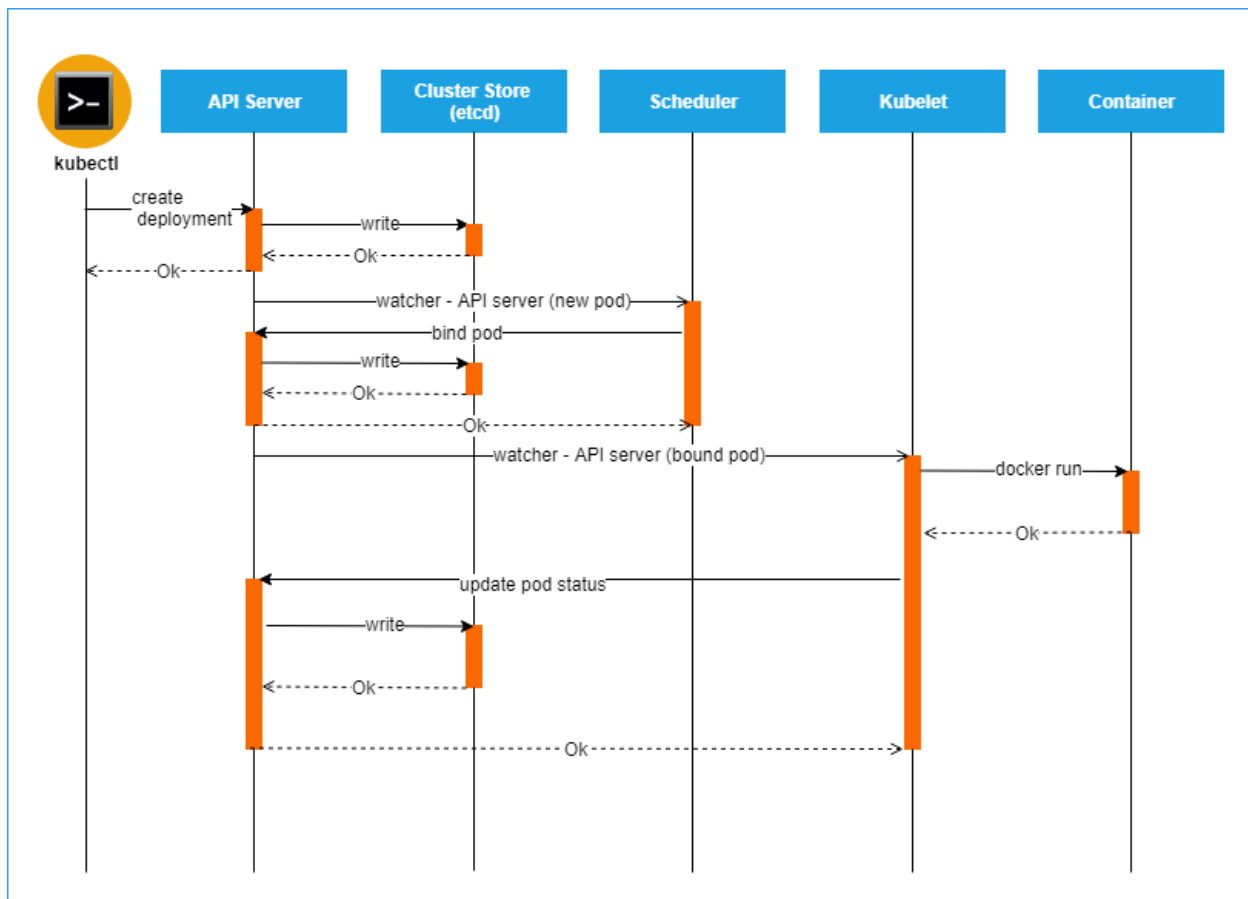


Figure 4: Pod creation sequence. Source: [Heptio](https://heptio.com)

The **kubectl** tool sends the application manifest to the API server. The API server extracts the desired cluster information from the manifest and persists it in the cluster store. The cluster store acknowledges the request, and a success response is sent back to the caller.

At this stage, the Kubernetes watchers come into effect. All Kubernetes object resource types are required to maintain an incremental change feed called the *watch*. A client can send an HTTP request to the watch endpoint of the API to get an incremental feed of the cluster changes.

You can see the feed for yourself by starting the **kubectl** proxy with the following command.

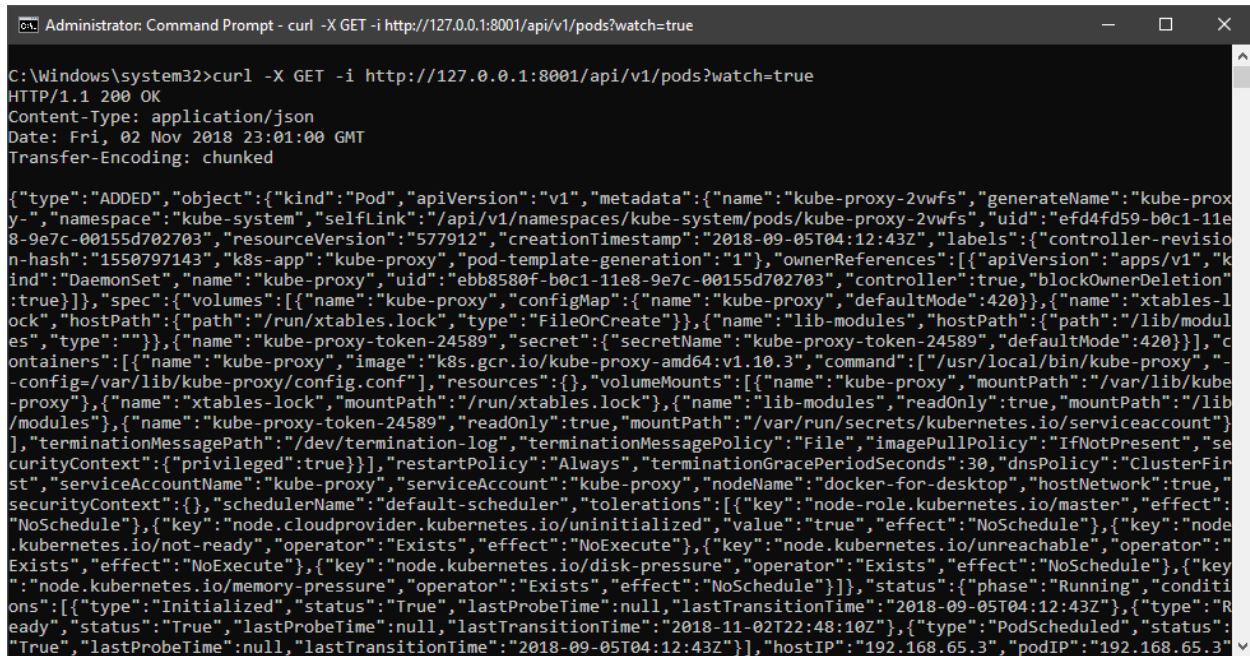
Code Listing 1: kubectl proxy

```
kubectl proxy
```

Next, use another terminal to send a request to the watch feed for observing changes to the pod objects.

Code Listing 2: Observe pod changes

```
curl -X GET -i http://127.0.0.1:8001/api/v1/pods?watch=true
```



```
C:\Windows\system32>curl -X GET -i http://127.0.0.1:8001/api/v1/pods?watch=true
HTTP/1.1 200 OK
Content-Type: application/json
Date: Fri, 02 Nov 2018 23:01:00 GMT
Transfer-Encoding: chunked

{"type":"ADDED","object":{"kind":"Pod","apiVersion":"v1","metadata":{"name":"kube-proxy-2vwfs","generateName":"kube-proxy-","namespace":"kube-system","selflink":"/api/v1/namespaces/kube-system/pods/kube-proxy-2vwfs","uid":"efd4fd59-b0c1-11e8-9e7c-00155d702703","resourceVersion":"577912","creationTimestamp":"2018-09-05T04:12:43Z","labels":{"controller-revision-hash":"1550797143","k8s-app":"kube-proxy","pod-template-generation":"1"},"ownerReferences":[{"apiVersion":"apps/v1","kind":"DaemonSet","name":"kube-proxy","uid":"ebb8580f-b0c1-11e8-9e7c-00155d702703","controller":true,"blockOwnerDeletion":true}]},"spec":{"volumes":[{"name":"kube-proxy","configMap":{"name":"kube-proxy","defaultMode":420}},{"name":"xtables-lock","hostPath":{"path":"/run/xtables.lock","type":"FileOrCreate"}},{"name":"lib-modules","hostPath":{"path":"/lib/modules","type":""}},{"name":"kube-proxy-token-24589","secret":{"secretName":"kube-proxy-token-24589","defaultMode":420}},{"containers":[{"name":"kube-proxy","image":"k8s.gcr.io/kube-proxy-amd64:v1.10.3","command":["usr/local/bin/kube-proxy","-config=/var/lib/kube-proxy/config.conf"],"resources":{"volumeMounts":[{"name":"kube-proxy","mountPath":"/var/lib/kube-proxy"}, {"name":"xtables-lock","mountPath":"/run/xtables.lock"}, {"name":"lib-modules","readOnly":true,"mountPath":"/lib/modules"}, {"name":"kube-proxy-token-24589","readOnly":true,"mountPath":"/var/run/secrets/kubernetes.io/serviceaccount"}]},"terminationMessagePath":"/dev/termination-log","terminationMessagePolicy":"File","imagePullPolicy":"IfNotPresent","securityContext":{"privileged":true}},{"restartPolicy":"Always","terminationGracePeriodSeconds":30,"dnsPolicy":"ClusterFirst","serviceAccountName":"kube-proxy","serviceAccount":{"name":"kube-proxy","node":{"name":"docker-for-desktop","hostNetwork":true,"securityContext":{"schedulerName":"default-scheduler","tolerations":[{"key":"node-role.kubernetes.io/master","effect":"NoSchedule"}, {"key":"node.cloudprovider.kubernetes.io/uninitialized","value":"true","effect":"NoSchedule"}, {"key":"node.kubernetes.io/not-ready","operator":"Exists","effect":"NoExecute"}, {"key":"node.kubernetes.io/unreachable","operator":"Exists","effect":"NoExecute"}, {"key":"node.kubernetes.io/disk-pressure","operator":"Exists","effect":"NoSchedule"}, {"key":"node.kubernetes.io/memory-pressure","operator":"Exists","effect":"NoSchedule"}]},"status":{"phase":"Running"},"conditions":[{"type":"Initialized","status":"True","lastProbeTime":null,"lastTransitionTime":"2018-09-05T04:12:43Z"}, {"type":"Ready","status":"True","lastProbeTime":null,"lastTransitionTime":"2018-11-02T22:48:10Z"}, {"type":"PodScheduled","status":"True","lastProbeTime":null,"lastTransitionTime":"2018-09-05T04:12:43Z"}]},"hostIP":"192.168.65.3","podIP":"192.168.65.3"}}
```

Figure 5: Output pod changes

You will notice that this request does not terminate, and that you will keep getting changes appended to the feed when a pod object changes. A Kubernetes daemon named **raven** is responsible for watching the changes using this feed and notifying the other components of the Kubernetes master whenever it detects a change.

When the scheduler gets a notification to create a new pod in the cluster, it finds the appropriate node in the cluster that will host the pod and responds to the API server with the details of the scheduled pod. The API server persists this information to the cluster store. Next, the daemon sends a notification to the kubelet that is running on the identified pod. The kubelet reads the desired pod configuration from the feed and triggers the Docker daemon to create a new container using the supplied image. The Docker daemon, or **rkt** (pronounced “rocket”), provides container management capabilities for Linux clusters, just like Docker.

Next, let’s discuss the components that are the workhorses of Kubernetes clusters that host your application workloads: nodes.

Nodes

In a cluster, a physical or virtual machine is a node. Each node can host many pods, which is the unit of deployment in Kubernetes. Additionally, every node in the cluster runs several Kubernetes components, most importantly, the kubelet and the kube proxy. Some nodes in the cluster run the Kubernetes master components, whose role in the cluster is to manage the

nodes. The nodes host the application and are controlled by the Kubernetes master. The following diagram shows the high-level architecture of the Kubernetes node.



Figure 6: Kubernetes node

A Kubernetes node is made of three major components: a kubelet, a container engine, and a proxy. We will go through the components in detail shortly, but for now, let's discuss the roles that these components play in the node.

As we previously discussed in the pod scheduling workflow, the kubelet agent registers the node with the cluster and watches the API server for changes. The kubelet instantiates pods and responds to the master with the operation status. The container engine (Docker or `rkt`) is responsible for creating containers from images and starting or stopping them. Another essential component of the node is the proxy. Every pod in the cluster gets an IP address, which is shared by all the containers that are present in the pod. The proxy helps load-balance traffic coming to pods that are part of a service.

Pods

The pod is the unit of deployment in Kubernetes. Each pod contains one or more containers. Although Kubernetes can work with any type of container, such as Docker and `rkt`, you will find a much better integration experience and ecosystem around Docker because of its popularity. Pods provide a persistent environment for containers to run in. Therefore, all containers in a pod share the same IP address, the same storage, and the same kernel namespace. Containers inside a pod can communicate with each other using `localhost` or inter-process communication.

Remember that pods, and not containers, are the deployment units in Kubernetes, so if a container crashes, the entire pod is replaced by the master. Likewise, if you want to scale out the cluster, new replicas of pods will be deployed in nodes.

A pod provides an isolated environment for its containers. A pod can host more than one container; however, since pods are atomic, all the containers that a pod hosts are lost when a pod is decommissioned. In general, you would host more than one container in a pod to implement [sidecars](#), such as log scrapers, which have one-to-one dependency on the main application.

The following diagram illustrates the pod environment, which consists of isolated units of network stacks, cgroups, and namespaces. The diagram also illustrates the network mechanisms available to the containers to interact with each other.

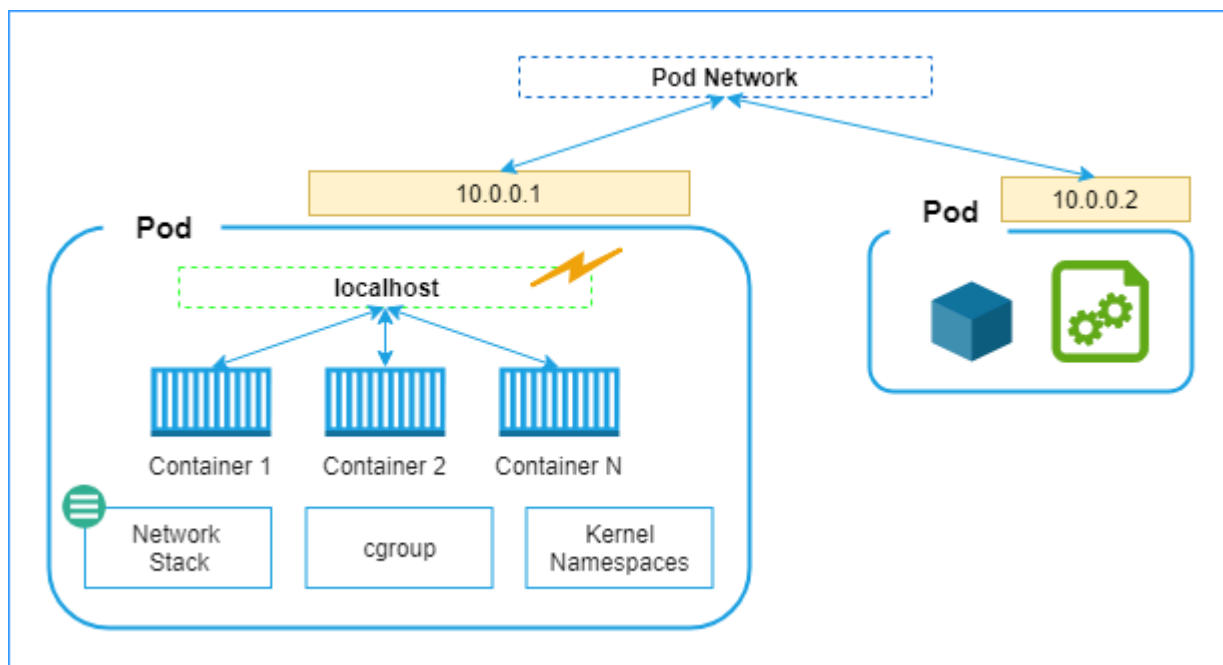


Figure 7: Pod network

You can see that containers present in the same pod can send requests to each other using different ports on the localhost. However, for communicating with containers across pods, they can use the pod network. Each pod gets a unique, network-visible IP address. Pods can send requests to other pods using their IP addresses and the known port numbers of the target containers. However, since pods can die, and new pods can be created any time, it becomes tough for an application to discover pods. Therefore, a higher-level object called **Services** should be used for networking. Amongst a host of features such as load balancing and high availability, services maintain an IP table of the associated pods, which provides a layer of indirection to the pod requests.

DNS

The DNS service, **kube-dns**, is scheduled as a regular pod in a Kubernetes cluster. A service in Kubernetes is a logical set of pods. Take an application that consists of a front-end application and a back-end application as an example. You can deploy replicas of each of the applications in multiple pods, and you can name the two types of pods front ends and back ends, each of which would be a service.

All services in Kubernetes receive a DNS name. So, if your front-end service wants to request some data from a back-end service, then it can resolve the address of the back-end cluster using the DNS. Using DNS, services can be placed anywhere in a cluster without requiring a change in the other services. Just like services, pods also get a DNS record, if enabled. Later in this book, we will interact with the DNS service to connect two services together.

Proxy

A node is made up of several components that render the node a part of the cluster. Only a node in the cluster can get pods scheduled and execute workloads. One of the components of the node is the proxy. The proxy, **kube-proxy**, takes care of the networking requirement of the node. It reflects Kubernetes REST API services and makes them available locally on the node. It also talks to the DNS service, has implicit knowledge of environment variables, and can use them to find out the cluster IP addresses. It can also help carry out communication between nodes and master through TCP and UDP forwarding.

Kubelet

The kubelet is the heart of the node. It connects the node to the master and manages the communication between the node and the master. It manages the lifecycle of pods scheduled on a container. Every time a pod is scheduled on the node, the kubelet carries out the following operations:

- Downloads pod secrets from the API server.
- Mounts storage volumes if requested by the scheduler.
- Runs the containers in pods.
- Runs container health probes.
- Reports health status of node and pods to the master.

Node supervisor

The node supervisor, **supervisord**, runs on each node and maintains and manages the availability of other services, such as **kubelet**, **docker**, or **rkt**, on the nodes. The supervisor creates subprocesses for each service, and its lifetime is managed by **supervisord**. If any subprocess crashes, **supervisord** is notified of that, and it takes necessary action based on the configuration.

Logging agent

The logging agent, such as **fluentd**, runs on each node and is responsible for log collection, parsing, and distribution. The logging agent is usually deployed as a daemon set replica since it must run on each node. The node-level logging agent is the most common approach in Kubernetes since it helps in the faster collection of logs and allows you to transfer them to a relevant aggregator or storage. We will discuss the logging agent (**fluentd**) further in Chapter 7.

There are a few more essential concepts that you need to be aware of before getting your hands dirty with Kubernetes.

Replica set

To ensure high availability of services, you can maintain multiple copies of pods running the same service. The traffic on a replica set is automatically load-balanced. Kubernetes tries to maintain replicas across nodes so that a node failure does not disrupt the service. A replica set can be configured to add and remove pods automatically.

Services

Services are collections of pods that expose functionality to users or other services. Any traffic sent to the service will be forwarded to one of the pods that are associated with the service. The assignment of pods to services happens through label selector, which is a Kubernetes grouping primitive for selecting any pods whose labels match the criteria specified in the selector. We will discuss the concept of services in much more detail in Chapter 4.

Why do we need services?

Pod-to-pod communication in a cluster can take place through IP addresses as each pod gets a new IP address. However, pods are ephemeral, and they get replaced by the cluster manager in several cases, such as scaling. New pods get a new IP, which can break the communication link between the pods. This is also true for external services or users trying to access functionality in a pod.

External users and pods require a permanent IP address that can be used to access the pods. Services group the pods and provide features such as load balancing, endpoints, and service discovery. Using services, you can group the pods that perform a specific functionality. This aggregation of functionality in a cohesive unit is called a microservice. Now you understand why Kubernetes is a platform that has microservices in its DNA.

While we are on the topic of services, let's discuss the types of services in a little more detail. The different service types determine the behavior of services, based on which you can control service accessibility.

Types of services

For most applications, you would want to expose a part of the application (the front end) to the Internet. The different types of Kubernetes services control how the traffic originating from outside the cluster is routed to the pods. The following are the different types of services supported by Kubernetes.

Cluster IP

This service type is used when you want to expose the service within the cluster to other services or applications. It uses the cluster internal IP addresses and is the default service type. Another spinoff of this service type is the headless service that has no IP address, not even an internal cluster IP address, assigned to it. Headless services implement their own service discovery and load-balancing mechanisms. Since headless services can reach any pod directly, you will find that many load balancers and proxies are implemented using them.

Node port

This service type exposes your service externally to the outside world. It uses each node's external IP address and a static port to expose the service. To reach such a service, a user can send a request to **<Node IP>:<Node Port>**, which will ultimately reach the desired node. All the nodes will forward any traffic that they receive on the port to the service. A cluster IP will be created for the service internally, so the nodes will forward the traffic that they receive to the internal cluster IP of the service.

Load balancer

This service type also exposes your service externally. It works for cloud-based clusters, which have external load balancers. It depends on cloud provider capabilities to support external IP addresses. Although its implementation varies with the cloud vendor, in general, for such a service, a node port and cluster IP are created, and the external load balancer will route traffic to the node port.

External name

This type of service acts as a proxy to an external service that has a valid DNS name. Any traffic directed to this service will ultimately be routed to the external service. An external name service does not have a string selector, and therefore, any requests to this service are redirected to the target service by the DNS. This approach is different from any other redirections we have read about so far that use proxying or port forwarding.

Daemon set

The daemon set is responsible for running a copy of a pod on each node. It ensures that at least one copy of a pod is running on each node (or some nodes, in a few scenarios). If a daemon set is deleted, all pods created by it will also get deleted. A common example includes running the metric/log collector or health probe pod on each node. The daemon set can also be defined using YAML spec with **kind** equal to **DaemonSet**. A daemon set specification requires

apiVersion, **kind**, and **metadata** fields. We will see **DaemonSet** in action in the deployment chapter.

Deployment

Deployment controllers are used to update a replica set or a pod. You could use deployments to apply a rolling upgrade of applications on pods, and likewise, rollback an update that failed. For example, if your pods are running version 1.0 of an application and you want to apply version 1.1 of the application on each node, then a command like the following will do a controlled update of the image that your pods use.

Code Listing 3: Update image

```
kubectl set image deployment/your-deployment value-of-name=your-app-image:1.1
```

We will learn more about deployments in Chapter 3.

Volume

Applications can store data on the container on which they are running. However, a container's data store goes away when the container dies. The Kubernetes volume is a directory that is accessible to all containers in the pod, and its lifetime is the same as that of the pod. The storage will survive container restarts, but will be cleared when a pod dies. The media used to provide this storage is determined by the volume type used.

There are a variety of volume types available that can persist volume data on the cloud or local data center, such as AWS EBS, Azure Disks, and local. We will learn more about volumes in Chapter 4.

Stateful set

Stateful applications require data to be persisted even after the pod dies. A stateful set is a collection of pods that require the state to be persisted across activations of the pod. Stateful set pods have:

- A stable hostname in DNS.
- An ordinal index, which is a number ranging from 0 to N-1, where N is the number of replicas of the stateful set.
- Storage mapped to the hostname and the ordinal index.

As you can imagine, in this case, a more persistent volume would be required to maintain the state of pods. This type of storage is called persistent volume (PV), and it is a cluster resource just like nodes. Just like with volumes, developers can choose the type of persistent volume that they want to use. Some commonly used persistent volume types are `GCEPersistentDisk`,

AWSElasticBlockStore, AzureFile, and Cinder. We will learn more about stateful sets in Chapter 4.

Namespace

Kubernetes allows you to segregate your deployments into different namespaces, which act as virtual clusters within the same physical cluster. Like programming namespaces, Kubernetes namespaces also provide a scope boundary that allows you to have unique resource names within the namespace.

You can create an isolation boundary within a set of applications on a cluster using namespaces. A namespace is a virtual cluster, and it can communicate with another namespace using only public interfaces. Nodes and persistent volumes are not part of namespaces. Therefore, Kubernetes can schedule pods from different namespaces on a node. Similarly, pods from different namespaces can mount the same persistent volume.



Note: Namespaces should only be used if you have the proper use case for them, like segregating resources among different users, or applying different access controls. If you have different versions of the same resource, then you should always use labels to separate them.

You can create a Kubernetes namespace using the following command.

Code Listing 4: Create namespace

```
kubectl create namespace remindwebnamespace
```

To view all namespaces in your cluster, you can use the following command.

Code Listing 5: Get namespaces

```
kubectl get namespaces
```

By default, all resources are created in the default namespace. If you don't specify the name of namespace in the specification of a Kubernetes object, it gets created in the default namespace.

Labels and annotations

A label is to Kubernetes what a group name is to a collection of people. Labels are key-value pairs that are used to identify objects or group a set of objects together. Just like people can be part of many groups (and thus have many group names), labels have a many-to-many relationship with objects. Labels have certain restrictions on their design. A label must have a unique key composed of two parts: a prefix and a name. The prefix is optional, but if present, it must be a valid DNS subdomain, and it must be at most 253 characters long. Names are required, and they can be at most 63 characters long. They must start with an alphanumeric

value (a-z, A-Z, 0-9) and can contain alphanumeric characters, dots, dashes, and underscores. Values have similar restrictions as names.

Sometimes, you may want to associate metadata with objects, such as a version number, build number, or other human-readable information. You can use annotations to associate additional information specific to an object. Annotations can be simple values or complex blocks of nested JSON that you can apply on an object.



Tip: You should maintain a map of label names and their meaning or intention in your project's README file (file with essential project details). Some of the common labels that you should have in your resources are: environment, version, application name, and service tier. This will later help you in querying resources using label selectors, such as `kubectl get pods -l version=1.0.0`.

You can refer to an object by using a label selector. A label selector is a simple statement consisting of a key name, operator, and a value. For example, the following statement will select all objects that label key as **role** and value as **frontend**:

```
role = frontend
```

A comma can delimit multiple conditions in selectors.

```
role = frontend, applicationName = fooBar
```

You can also use set-based operations to select objects that match one of the values.

```
role in (frontend, backend)
```

Names

Every object in Kubernetes has a unique identifier (UID) and a name. You can use the name of the object to refer to it. Names must be made of lowercase alphanumeric characters, dashes, and dots. They should be at most 253 characters long. While creating an object, you only specify its name; Kubernetes generates and maintains the object's UID.

Secrets

You don't need to package your application secrets with your application. Custom application secrets and cluster secrets (secrets generated by Kubernetes for its components) are stored as plain text in **etcd**. These are directly accessible by the REST API. However, you can mount the secrets on all the pods that need to use them. The secrets in a pod are only stored in memory and never written to the volume for better security.

If you don't want to store secrets in **etcd**, you can add them as environment variables in pods, and the pods can read them from the variables available only to them. We will learn more about secrets in Chapter 4.

We have covered quite a lot of ground in this chapter. A knowledge of the terms and the components used in Kubernetes will help you map the components to the Kubernetes architecture later. Let's wrap up this chapter with a short discussion of use cases where Kubernetes is gaining popularity. In the next chapter, we will prepare our developer terminals and create our first cluster on Kubernetes.

Use cases

Kubernetes gives developers and operations a consistent experience everywhere, from your laptop to popular public clouds such as Azure, AWS, and Google Cloud. Most enterprises don't like to have their application vendor locked so that they're still able to utilize features mostly available to cloud applications, such as load balancing, persistent storage, and autoscaling. With Kubernetes, you can migrate your applications without any change between clouds and create federated clusters that span multiple clouds.

Since Kubernetes is platform-independent, developers can build applications on any platform and use Kubernetes to orchestrate them. The operations team can also use a consistent experience to manage the applications without investing time and resources in different tools.

Organizations are also using Kubernetes to build their infrastructure platform. Developers can build and package their applications and host them on the platform without worrying about contentions with other applications hosted on the platform as namespaces segregate them. Developers also get a consistent environment in all stages of deployment, from development to production.

Summary

In this chapter, we covered some of the fundamental components that make up the Kubernetes platform. We will discuss the rest of them as we keep enriching our knowledge of the platform.

Reading about the many subcomponents of the Kubernetes platform might seem a little overwhelming to you at first. However, we will interact with several of the components that we previously discussed to deploy and manage a microservices application. As you follow along, you will be able to piece the various parts of the Kubernetes puzzle together in your mind. In the following chapter, we will prepare a Kubernetes cluster on bare metal and prepare our development environment for deploying applications on Kubernetes.

Chapter 2 Installation

Now that we have a good understanding of primary Kubernetes concepts, let's get our hands in the game by provisioning our first Kubernetes cluster. In this chapter, we will cover the installation of a Kubernetes cluster on an Ubuntu 17.10 server using **kubeadm**. Although the steps to install on other operating systems (OS) remain [more or less the same](#), it is essential that you refer to the steps of your specific OS for any prerequisites.

In this chapter, we will also cover installing a Kubernetes cluster on a local Windows machine and a local Mac machine, which will help you build and debug your applications before publishing them on a live cluster.

Setting up a cluster using kubeadm

kubeadm helps in creating a minimum viable Kubernetes cluster. It focuses on the initial setup and bootstrapping of the cluster on already provisioned machines. **kubeadm init** and **kubeadm join** are the two main commands:

- **kubeadm init**: This initializes a Kubernetes master with the desired configuration. It has many flags, which you can specify to configure your cluster. Some of these configurations are dependent upon your pod network add-on.
- **kubeadm join**: This initializes and joins one or more Kubernetes nodes to an existing Kubernetes cluster. It takes credentials for your cluster and joins the nodes to it.

As explained in Chapter 1, Kubernetes consists of many components as part of its setup. However, there are two main components that are the building blocks of your cluster:

- Kubernetes master
- Kubernetes node(s)

The Kubernetes master is the core of a Kubernetes cluster, and manages all the components in the cluster.

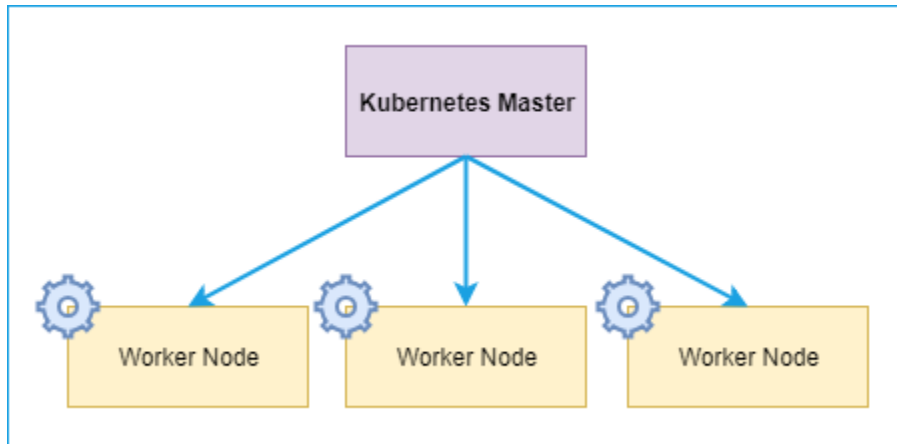


Figure 8: Kubernetes installation component

Prerequisite

The following are the prerequisites for setting up a Kubernetes cluster:

- 2 GB+ RAM per machine.
- 2 CPUs or more.
- Network connectivity among all machines in the cluster (public or private).
- A unique hostname, MAC address, and **product_uuid** for every node.
- Required ports open on your machines.



Tip: You can get the MAC address of the network interfaces using the command `ifconfig -a`. To check the **product_uuid**, use the following command: `sudo cat /sys/class/dmi/id/product_uuid`.

The following is a summary of ports that need to be open for the cluster components to communicate with each other and with the external traffic.

Table 1: Port list (Source: [Kubernetes](https://kubernetes.io/docs/concepts/cluster-administration/networking/))

PortRange	Purpose	Node
6443	Kubernetes API server	Master
2379-2380	etcd server client API	Master
10250	Kubelet API	Master
10251	kube-scheduler	Master
10252	kube-controller-manager	Master
10250	Kubelet API	Worker

PortRange	Purpose	Node
30000-32767	NodePort Services	Worker

You will need to work through the documentation of your OS network configuration and firewall settings to enable the ports listed in the table.



Tip: You can check whether the required port is in use or not by using the following command: `netstat -a |grep LISTEN`.

For clouds such as Microsoft Azure, Amazon AWS, and Google GCP, you can expose the required ports via their management portals.

Setting up the Kubernetes master

Once all the prerequisites are satisfied, you can start working on the Kubernetes master installation. Start with the update of existing system packages.

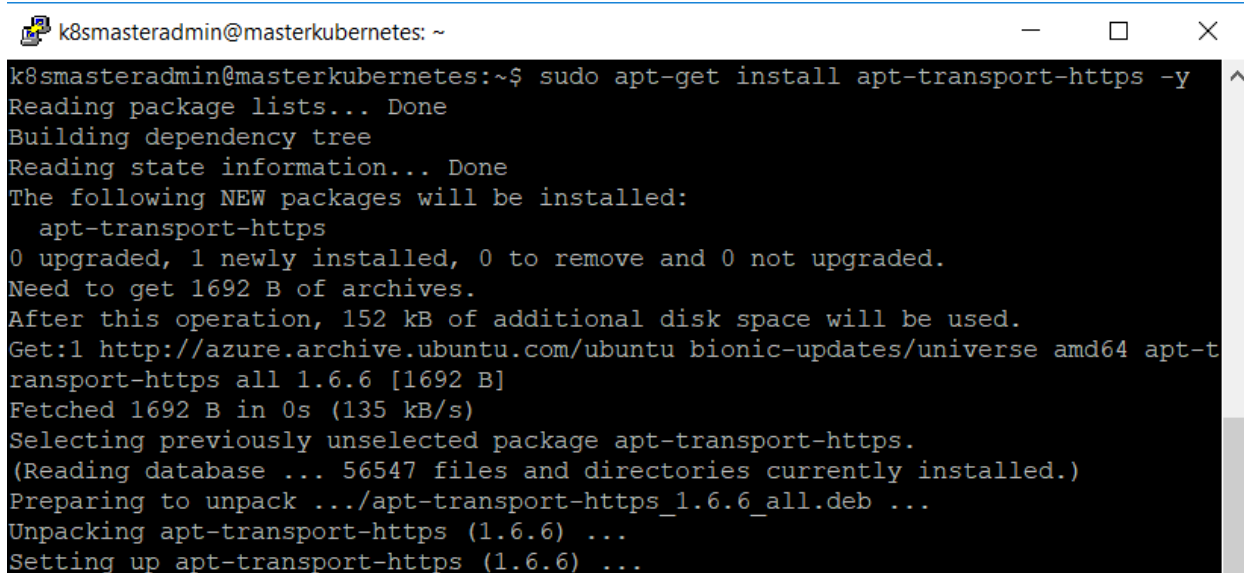
Code Listing 6: Prepping machine

```
#Updates the list of packages.
sudo apt-get update
#Upgrades to latest packages.
sudo apt-get upgrade -y
```

The next step will be to install the **apt-transport-https** package, which is required for accessing the https URL of some of the repositories used in the installation.

Code Listing 7: Installing apt-transport-https package

```
sudo apt-get install apt-transport-https -y
```

A terminal window titled 'k8smasteradmin@masterkubernetes: ~' with standard window controls. The terminal output shows the command 'sudo apt-get install apt-transport-https -y' being executed. The output includes: 'Reading package lists... Done', 'Building dependency tree', 'Reading state information... Done', 'The following NEW packages will be installed: apt-transport-https', '0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.', 'Need to get 1692 B of archives.', 'After this operation, 152 kB of additional disk space will be used.', 'Get:1 http://azure.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 apt-transport-https all 1.6.6 [1692 B]', 'Fetched 1692 B in 0s (135 kB/s)', 'Selecting previously unselected package apt-transport-https.', '(Reading database ... 56547 files and directories currently installed.)', 'Preparing to unpack .../apt-transport-https_1.6.6_all.deb ...', 'Unpacking apt-transport-https (1.6.6) ...', and 'Setting up apt-transport-https (1.6.6) ...'.

```
k8smasteradmin@masterkubernetes:~$ sudo apt-get install apt-transport-https -y
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following NEW packages will be installed:
  apt-transport-https
0 upgraded, 1 newly installed, 0 to remove and 0 not upgraded.
Need to get 1692 B of archives.
After this operation, 152 kB of additional disk space will be used.
Get:1 http://azure.archive.ubuntu.com/ubuntu bionic-updates/universe amd64 apt-transport-https all 1.6.6 [1692 B]
Fetched 1692 B in 0s (135 kB/s)
Selecting previously unselected package apt-transport-https.
(Reading database ... 56547 files and directories currently installed.)
Preparing to unpack .../apt-transport-https_1.6.6_all.deb ...
Unpacking apt-transport-https (1.6.6) ...
Setting up apt-transport-https (1.6.6) ...
```

Figure 9: Output apt-transport-https package

Next, we need to install the **docker** package on which all our containers will run. Run the following four commands in the order in which they are listed.

Code Listing 8: Installing Docker

```
# Add Docker official GPG key. Trailing '-' means use stdin.
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key
add -

# Set up the stable repository.
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) \
    stable"

# Update the apt package index.
sudo apt-get update

# Install the latest version of docker ce.
sudo apt-get install docker-ce
```

```
k8smasteradmin@masterkubernetes: ~  
k8smasteradmin@masterkubernetes:~$ sudo apt-get install docker-ce  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  aufs-tools cgroupfs-mount libltdl7 pigz  
The following NEW packages will be installed:  
  aufs-tools cgroupfs-mount docker-ce libltdl7 pigz  
0 upgraded, 5 newly installed, 0 to remove and 0 not upgraded.  
Need to get 40.4 MB of archives.  
After this operation, 199 MB of additional disk space will be used.  
Do you want to continue? [Y/n] y  
Get:1 https://download.docker.com/linux/ubuntu bionic/stable amd64 docker-ce amd64 18.06.1~ce~3-0~ubuntu [40.2 MB]  
Get:2 http://azure.archive.ubuntu.com/ubuntu bionic/universe amd64 pigz amd64 2.4-1 [57.4 kB]  
Get:3 http://azure.archive.ubuntu.com/ubuntu bionic/universe amd64 aufs-tools amd64 1:4.9+20170918-1ubuntu1 [104 kB]  
Get:4 http://azure.archive.ubuntu.com/ubuntu bionic/universe amd64 cgroupfs-mount all 1.4 [6320 B]  
Get:5 http://azure.archive.ubuntu.com/ubuntu bionic/main amd64 libltdl7 amd64 2.4.6-2 [38.8 kB]  
Fetched 40.4 MB in 1s (27.5 MB/s)  
Selecting previously unselected package pigz.  
(Reading database ... 56551 files and directories currently installed.)  
Preparing to unpack .../archives/pigz_2.4-1_amd64.deb ...  
Unpacking pigz (2.4-1) ...  
Selecting previously unselected package aufs-tools.  
Preparing to unpack .../aufs-tools_1%3a4.9+20170918-1ubuntu1_amd64.deb ...  
Unpacking aufs-tools (1:4.9+20170918-1ubuntu1) ...  
Selecting previously unselected package cgroupfs-mount.  
Preparing to unpack .../cgroupfs-mount_1.4_all.deb ...  
Unpacking cgroupfs-mount (1.4) ...  
Selecting previously unselected package libltdl7:amd64.  
Preparing to unpack .../libltdl7_2.4.6-2_amd64.deb ...  
Unpacking libltdl7:amd64 (2.4.6-2) ...  
Selecting previously unselected package docker-ce.  
Preparing to unpack .../docker-ce_18.06.1~ce~3-0~ubuntu_amd64.deb ...  
Unpacking docker-ce (18.06.1~ce~3-0~ubuntu) ...  
Setting up aufs-tools (1:4.9+20170918-1ubuntu1) ...  
Processing triggers for ureadahead (0.100.0-20) ...  
Setting up cgroupfs-mount (1.4) ...  
Processing triggers for libc-bin (2.27-3ubuntu1) ...  
Processing triggers for systemd (237-3ubuntu10.3) ...  
Setting up libltdl7:amd64 (2.4.6-2) ...  
Processing triggers for man-db (2.8.3-2ubuntu0.1) ...  
Setting up pigz (2.4-1) ...  
Setting up docker-ce (18.06.1~ce~3-0~ubuntu) ...  
Created symlink /etc/systemd/system/multi-user.target.wants/docker.service → /li
```

Figure 10: Output Docker installation

Once Docker is installed, it's time to start and enable the Docker daemon.

Code Listing 9: Enabling Docker

```
# Start the Docker service.  
sudo systemctl start docker  
  
# Enable the Docker service.  
sudo systemctl enable docker
```

If you didn't encounter any errors, then your system is ready for installing Kubernetes. Let's start with adding a public key for the Google Kubernetes packages that are required for accessing Google packages from the web. We will also create a file in `/etc/sources` and add a link for downloading the Kubernetes package.

Code Listing 10: Adding Google Cloud key and link

```
# Add the official Google GPG key.

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

# Open file in your preferred editor.

sudo nano /etc/apt/sources.list.d/kubernetes.list

# Add following key in Kubernetes.list file and save it.

deb http://apt.kubernetes.io/ kubernetes-xenial main
```

Now, run **apt-get update** to download the newly added package.

Code Listing 11: Update machine

```
sudo apt-get update
```

Now we can install the **kubeadm**, **kubelet**, **kubect**, and **kubernetes-cni** packages.

Code Listing 12: Installing Kubernetes components

```
sudo apt-get install -y kubelet kubeadm kubectl kubernetes-cni
```

```
k8smasteradmin@masterkubernetes: ~  
Reading package lists... Done  
Building dependency tree  
Reading state information... Done  
The following additional packages will be installed:  
  cri-tools socat  
The following NEW packages will be installed:  
  cri-tools kubeadm kubect1 kubelet kubernetes-cni socat  
0 upgraded, 6 newly installed, 0 to remove and 0 not upgraded.  
Need to get 54.8 MB of archives.  
After this operation, 363 MB of additional disk space will be used.  
Get:2 http://azure.archive.ubuntu.com/ubuntu bionic/main amd64 socat amd64 1.7.3.2-2ubuntu2 [342 kB]  
Get:1 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 cri-tools amd64 1.12.0-00 [5343 kB]  
Get:3 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubernetes-cni amd64 0.6.0-00 [5910 kB]  
Get:4 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubelet amd64 1.12.2-00 [24.6 MB]  
Get:5 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubect1 amd64 1.12.2-00 [9599 kB]  
Get:6 https://packages.cloud.google.com/apt kubernetes-xenial/main amd64 kubeadm amd64 1.12.2-00 [8989 kB]  
Fetched 54.8 MB in 3s (17.3 MB/s)  
Selecting previously unselected package cri-tools.  
(Reading database ... 56851 files and directories currently installed.)  
Preparing to unpack .../0-cri-tools_1.12.0-00_amd64.deb ...  
Unpacking cri-tools (1.12.0-00) ...  
Selecting previously unselected package kubernetes-cni.  
Preparing to unpack .../1-kubernetes-cni_0.6.0-00_amd64.deb ...  
Unpacking kubernetes-cni (0.6.0-00) ...  
Selecting previously unselected package socat.  
Preparing to unpack .../2-socat_1.7.3.2-2ubuntu2_amd64.deb ...  
Unpacking socat (1.7.3.2-2ubuntu2) ...  
Selecting previously unselected package kubelet.  
Preparing to unpack .../3-kubelet_1.12.2-00_amd64.deb ...  
Unpacking kubelet (1.12.2-00) ...  
Selecting previously unselected package kubect1.  
Preparing to unpack .../4-kubect1_1.12.2-00_amd64.deb ...  
Unpacking kubect1 (1.12.2-00) ...  
Selecting previously unselected package kubeadm.  
Preparing to unpack .../5-kubeadm_1.12.2-00_amd64.deb ...  
Unpacking kubeadm (1.12.2-00) ...  
Setting up kubernetes-cni (0.6.0-00) ...
```

Figure 11: Output Kubernetes component installation

If there are no errors, then we are good to proceed with cluster initialization. For initializing our cluster, we will need the public IP of our master node. Use the following command to initialize the Kubernetes cluster.

Code Listing 13: Initializing cluster using kubeadm

```
sudo kubeadm init --pod-network-cidr= 192.168.0.0/16 --apiserver-  
advertise-address=13.76.45.201 <---ipaddress of master-->
```

```
k8smasteradmin@masterkubernetes: ~  
space  
[addons] Applied essential addon: CoreDNS  
[addons] Applied essential addon: kube-proxy  
  
Your Kubernetes master has initialized successfully!  
  
To start using your cluster, you need to run the following as a regular user:  
  
  mkdir -p $HOME/.kube  
  sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config  
  sudo chown $(id -u):$(id -g) $HOME/.kube/config  
  
You should now deploy a pod network to the cluster.  
Run "kubectl apply -f [podnetwork].yaml" with one of the options listed at:  
  https://kubernetes.io/docs/concepts/cluster-administration/addons/  
  
You can now join any number of machines by running the following on each node  
as root:  
  
  kubeadm join 52.163.80.156:6443 --token ou5r8q.c5rnrjulqcvbyscr --discovery-to  
ken-ca-cert-hash sha256:5a061673a0ee1bcc4fffea47b562cc4ed11aca1238c97a9d6f498b10  
6a064531
```

Figure 12: Output cluster initialization

Congratulations, your Kubernetes cluster is now ready for use. Your token and certificate hash are displayed as output of the previous command, and they can be used to join nodes to your cluster.



Note: If you get a message that your Docker version is not compatible with your Kubernetes version, you can downgrade your Docker version using this command: `sudo apt-get install docker-ce=<VERSION>`. You can check for the available Docker versions using this command: `apt-cache madison docker-ce`.

You can copy the command displayed in your console (`kubeadm join`) to use in our node(s) to join the cluster.



Note: The `--pod-network-cidr= 192.168.0.0/16` flag is only required if you are using the Calico pod network add-on. We will be using the Calico network add-on for our cluster, but you can use any other network add-on. See the section “Adding pod network add-on” in this chapter for details.

Before adding nodes to your cluster, execute the commands in the following code listing, which will set relevant permissions in your Kubernetes cluster config files.

Code Listing 14: Setting permission

```
# Create .kube directory in the Home path.

mkdir -p $HOME/.kube

# Copy default configuration to new directory.

sudo cp -i /etc/kubernetes/admin.conf $HOME/.kube/config

# Set the relevant permissions.

sudo chown $(id -u):$(id -g) $HOME/.kube/config
```



Note: You can execute `kubectl get nodes` to get the number of nodes running in your cluster. At this stage, you should get one node running as master. Your node will be in the “Not Ready” state because we have not applied a pod network add-on to the cluster.

We will use the `kubeadm join` command in our node(s) machine. However, before we do that, we need to apply the pod network add-on.

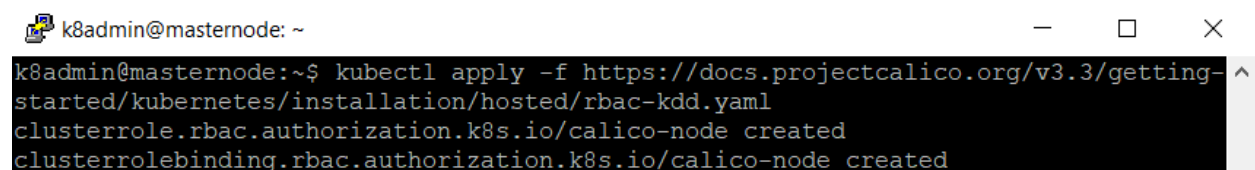
Adding pod network add-on

The pod network add-on is required for pods to communicate with each other. There are many default network add-ons available, all of which you can read about on the [Kubernetes.io site](https://kubernetes.io). We will use the Calico add-on to set up networking, which requires a Calico configuration. The configuration itself is divided into two parts. First, we will install role-based authorization for our network.

Code Listing 15: Applying pod network rbac

```
$ kubectl apply -f https://docs.projectcalico.org/v3.3/getting-
started/kubernetes/installation/hosted/rbac-kdd.yaml
```

After you get an output that looks like the following listing, you are good to proceed. The operation should not take more than a few seconds to complete.



```
k8admin@masternode: ~
k8admin@masternode:~$ kubectl apply -f https://docs.projectcalico.org/v3.3/getting-
started/kubernetes/installation/hosted/rbac-kdd.yaml
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
```

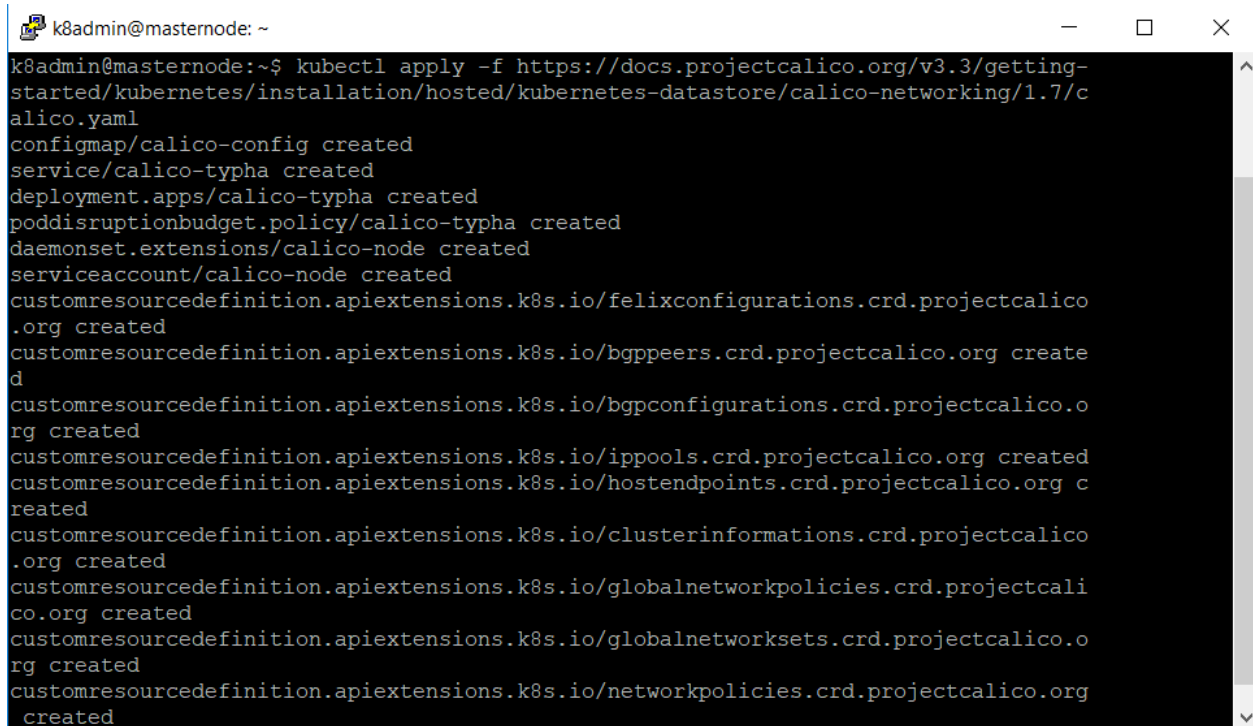
Figure 13: Output Role-Based Access Control (RBAC) network

The following command will install the network add-on to our cluster.

Code Listing 16: Applying pod network add-on

```
$ kubectl apply -f https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml
```

It will deploy the appropriate configuration and daemon set, along with network configuration. You should see output similar to the following listing.

A terminal window with a dark background and light text. The prompt is 'k8admin@masternode: ~'. The command entered is 'kubectl apply -f https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml'. The output lists several Kubernetes resources being created, including configmaps, services, deployments, daemonsets, serviceaccounts, and custom resource definitions for various Calico components like felix, bgp, and network policies.

```
k8admin@masternode: ~  
k8admin@masternode:~$ kubectl apply -f https://docs.projectcalico.org/v3.3/getting-started/kubernetes/installation/hosted/kubernetes-datastore/calico-networking/1.7/calico.yaml  
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.org created  
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/bgppolicies.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created  
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
```

Figure 14: Output network add-on

After successful execution, the **kubectl get nodes** command should give you the current status for your master node. You can check for more details about other components of Kubernetes master by using **kubectl get pods -all-namespaces** command. All the pods should be in running state before you start adding nodes.

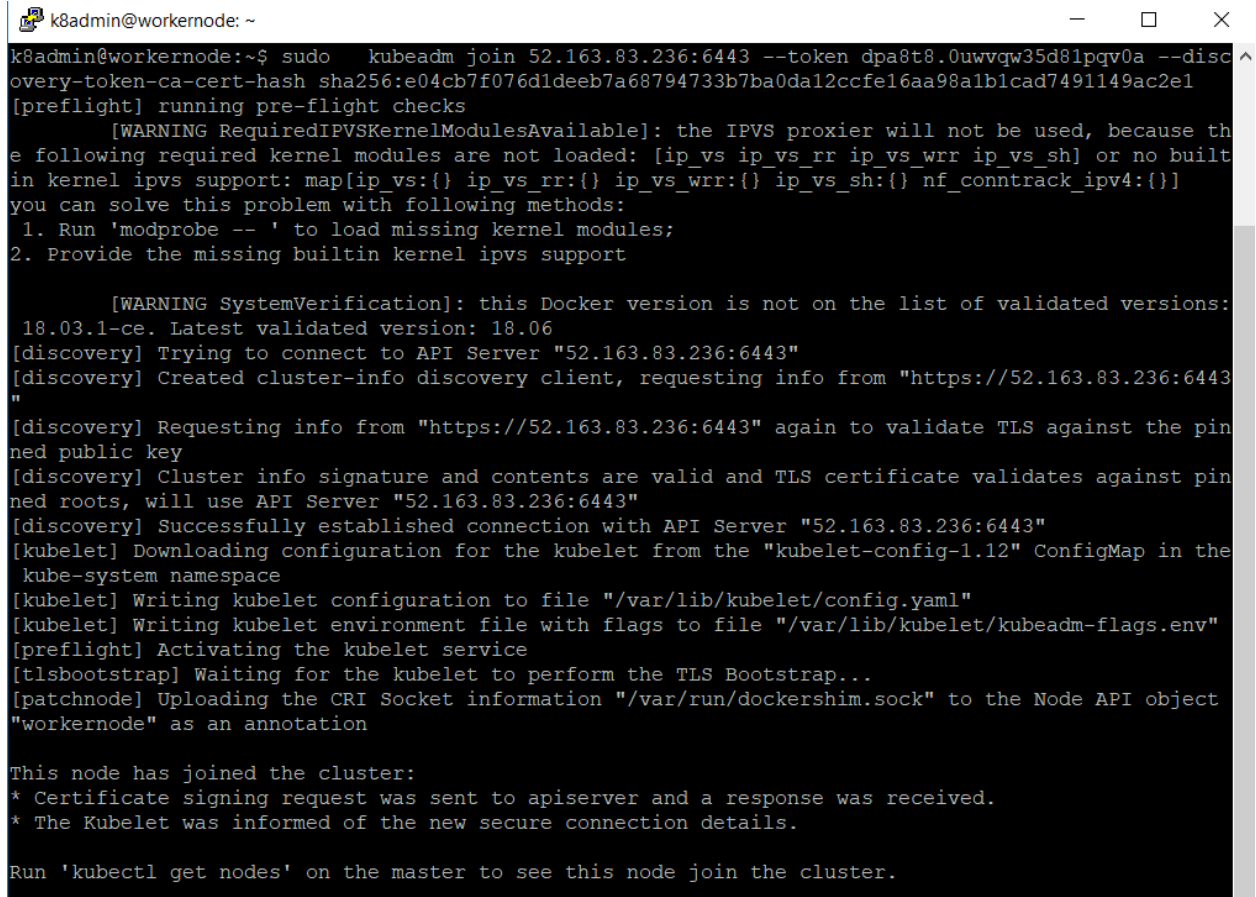
Adding worker nodes

Now that we've set up the master node, it's time to add worker nodes. Worker nodes are the workhorses of a Kubernetes cluster that host the workloads running on the cluster. Prepare each worker node by following the same steps you followed for the Kubernetes master, up to the installation of the **kubeadm**, **kublet**, **kubectl**, and **kubernetes-cni** packages.

After successful installation, use the **kubeadm join** command that you copied from the output of the Kubernetes master initialization in the previous steps.

Code Listing 17: Joining worker nodes

```
kubeadm join 13.76.45.201:6443 --token hkjku5.vqo1puk7qekto77g --discovery-  
token-ca-cert-hash  
sha256:ba95f86079b6bd81e93b8f1569177104da282a42a987538a4aae91c60e5a00d8
```

A terminal window titled 'k8admin@workernode: ~' showing the output of the 'kubeadm join' command. The output includes pre-flight checks, warnings about IPVS kernel modules and Docker version, discovery of the API server, and successful configuration of the kubelet. It concludes with a message that the node has joined the cluster and provides instructions to run 'kubectl get nodes' on the master.

```
k8admin@workernode:~$ sudo kubeadm join 52.163.83.236:6443 --token dpa8t8.0uwvqw35d81pqv0a --disc  
covery-token-ca-cert-hash sha256:e04cb7f076d1deeb7a68794733b7ba0da12ccfe16aa98a1b1cad7491149ac2e1  
[preflight] running pre-flight checks  
[WARNING RequiredIPVSKernelModulesAvailable]: the IPVS proxier will not be used, because th  
e following required kernel modules are not loaded: [ip_vs ip_vs_rr ip_vs_wrr ip_vs_sh] or no built  
in kernel ipvs support: map[ip_vs:{} ip_vs_rr:{} ip_vs_wrr:{} ip_vs_sh:{} nf_conntrack_ipv4:{}]  
you can solve this problem with following methods:  
1. Run 'modprobe -- ' to load missing kernel modules;  
2. Provide the missing builtin kernel ipvs support  
  
[WARNING SystemVerification]: this Docker version is not on the list of validated versions:  
18.03.1-ce. Latest validated version: 18.06  
[discovery] Trying to connect to API Server "52.163.83.236:6443"  
[discovery] Created cluster-info discovery client, requesting info from "https://52.163.83.236:6443  
"  
[discovery] Requesting info from "https://52.163.83.236:6443" again to validate TLS against the pin  
ned public key  
[discovery] Cluster info signature and contents are valid and TLS certificate validates against pin  
ned roots, will use API Server "52.163.83.236:6443"  
[discovery] Successfully established connection with API Server "52.163.83.236:6443"  
[kubelet] Downloading configuration for the kubelet from the "kubelet-config-1.12" ConfigMap in the  
kube-system namespace  
[kubelet] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"  
[kubelet] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"  
[preflight] Activating the kubelet service  
[tlsbootstrap] Waiting for the kubelet to perform the TLS Bootstrap...  
[patchnode] Uploading the CRI Socket information "/var/run/dockerhim.sock" to the Node API object  
"workernode" as an annotation  
  
This node has joined the cluster:  
* Certificate signing request was sent to apiservert and a response was received.  
* The Kubelet was informed of the new secure connection details.  
  
Run 'kubectl get nodes' on the master to see this node join the cluster.
```

Figure 15: Output joining nodes

You can repeat the same steps for all worker nodes that you want to join in the cluster.

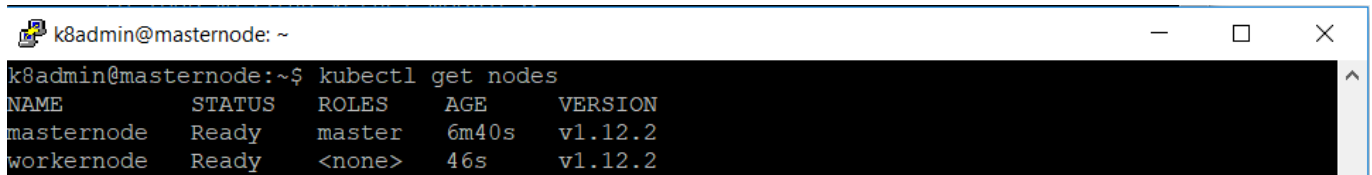


Note: If you lose the cluster join token that you previously generated, this command will print it again for you: `kubeadm token create --print-join-command`.

To verify that a node has successfully joined the cluster, execute the command `get nodes` on the master node of your cluster.

Code Listing 18: Get nodes

```
kubect1 get nodes
```



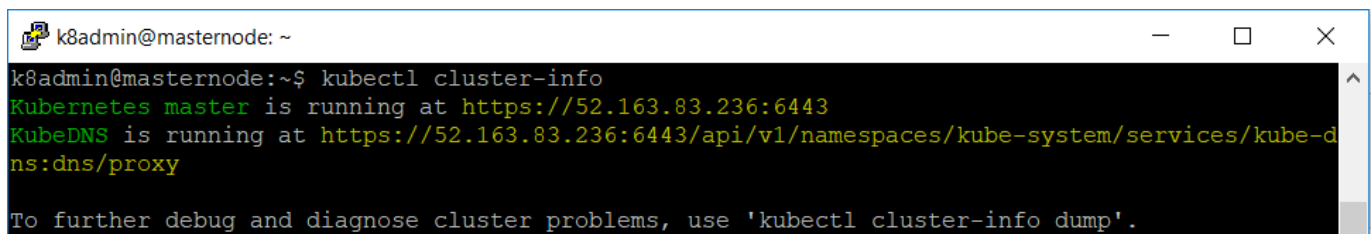
```
k8admin@masternode: ~  
k8admin@masternode:~$ kubectl get nodes  
NAME          STATUS    ROLES    AGE   VERSION  
masternode    Ready    master   6m40s v1.12.2  
workernode    Ready    <none>   46s   v1.12.2
```

Figure 16: Output get nodes

You can also view the status of your cluster by using the **cluster-info** command.

Code Listing 19: Get cluster-info

```
kubectl cluster-info
```



```
k8admin@masternode: ~  
k8admin@masternode:~$ kubectl cluster-info  
Kubernetes master is running at https://52.163.83.236:6443  
KubeDNS is running at https://52.163.83.236:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy  
  
To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

Figure 17: Output cluster-info

Your Kubernetes cluster is now up, running, and ready to host applications. If you faced any issues during cluster installation, you can refer to [this Kubernetes page](#) for details on common issues and troubleshooting them.

Setting up a cluster on a developer machine

We need everything running locally on our machines to debug and test our applications before promoting them to the production environment. Kubernetes has extensive support for running a small version of a massive, enterprise-grade cluster on a developer machine. Up until recently, the most popular solution for running Kubernetes locally was [Minikube](#). Minikube sets up a single node cluster on a local machine inside a VM and supports almost all the standard Kubernetes commands. This is helpful for running and testing applications on development systems. It is also a good starting point for learning Kubernetes.

With the latest Docker for Windows/Mac release, Docker has inbuilt support for Kubernetes, making setting up the local Kubernetes environment on the developer machine easy. Docker Swarm, which is a clustering and scheduling tool for Docker containers, was the only option in the previous version of Docker for Windows/Mac as an orchestrator. However, Docker now provides the option of choosing between Docker Swarm and Kubernetes.

Prerequisite

Docker for Windows and Docker for Mac are community editions for each OS. You need to have an account on the Docker store to download Docker for Windows or Mac.

Let's install Docker for Windows from the [Docker store](#). For Docker for Mac, the package and installation steps are available [on the Docker website](#).

The following are the basic system configurations required to install Docker for Windows:

- Windows 10 64 bit: Pro, Enterprise, or Education (1607 Anniversary Update, Build 14393 or later).
- Virtualization enabled in BIOS. Typically, virtualization is enabled by default (this is different from having Hyper-V enabled).
- CPU SLAT-capable feature.
- At least 4GB of RAM.
- Microsoft Hyper-V.

For Docker for Mac setup, the [following conditions](#) must be satisfied:

- Mac hardware must be a 2010 or newer model, with Intel's hardware support for memory management unit (MMU) virtualization, including Extended Page Tables (EPT) and Unrestricted Mode.
- macOS El Capitan 10.11 and newer macOS releases are supported.
- At least 4GB of RAM.
- VirtualBox prior to version 4.3.30 must NOT be installed (it is incompatible with Docker for Mac).



Tip: You can check to see if your Mac machine has this support by running the following command in a terminal: `sysctl kern.hv_support`.

The following components are installed as part of Docker for Windows or Mac setup:

- Docker Engine: Default docker daemon that runs in the background.
- Docker CLI client: Command-line client to interact with Docker daemon.
- Docker Compose: Used to run multicontainer Docker applications and enables support for YAML-based deployment.
- Docker machine: Helps in installing the Docker engine on virtual hosts.
- Kitematic: Automates the Docker installation and setup process and provides a graphical user interface (GUI) for running and managing Docker containers.



Note: If your system does not meet the requirements to run Docker for Windows/Mac, you can install [Docker Toolbox](#), which uses Oracle Virtual Box instead of Hyper-V.

If all the previous prerequisites are satisfied by your machine, you can go ahead and install the appropriate Docker setup.

Kubernetes installation

Once Docker for Windows or Mac is installed successfully on your local machine, go to **Settings** and click **Enable Kubernetes**. This will download the required images and set up the Kubernetes cluster for you. Awesome, right? How simple was this compared to the `kubeadm` setup that we just did for Linux cluster?

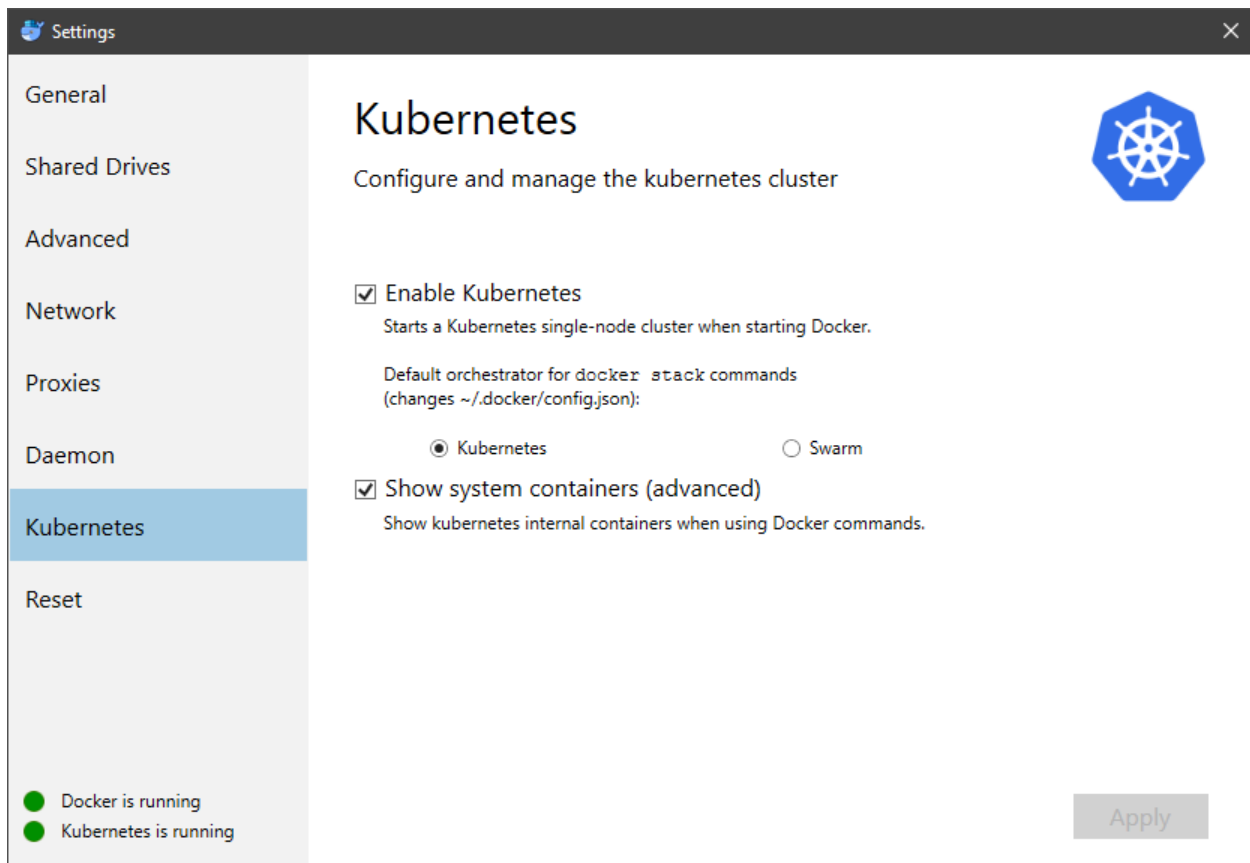



Figure 18: Docker for Kubernetes

You can also click **Show system containers**, which will make all the system containers visible. The Kubernetes server runs locally within your Docker instance, is not configurable, and is a single-node cluster.

 **Note:** Make sure the `kubectl` path (`C:\Program Files\Docker\Docker\Resources\bin\kubectl.exe`) is set in your cli `PATH` variable. For Mac, the path is `/usr/local/bin/kubectl`.

Once everything is green, and the “Kubernetes is running” message is shown, you can open the Docker CLI and start using `kubectl` commands.

Summary

In this chapter, we set up our Kubernetes cluster on Ubuntu VM using `kubeadm`. We also set up and installed Docker for Windows and Mac OS. Let’s move on to the next chapter to deploy our first application to our newly created cluster.

Chapter 3 Deploying Applications

The moment you have been waiting for has arrived. In a short while, we will deploy our first application on our Kubernetes cluster. To keep the examples relevant to the real world, we will incrementally develop our application and deliver value to our customers in phases. In every phase, we will make the solution a little better and deliver it to keep our customers happy and engaged for the entire duration of the project. Let's go through the requirements for the application we will be building.

Requirement

Our target customers are highly particular about planning their day-to-day activities. They need an application that provides an easy-to-use interface to add reminders and send them an email every night to remind them about the events that they have planned for the following day. After an unusually lengthy discussion, the customers came up with a unique name for the application—**Remind Me**.

Remind Me logical design

Given the requirements we have, we came up with the following logical design of the solution that we will build.

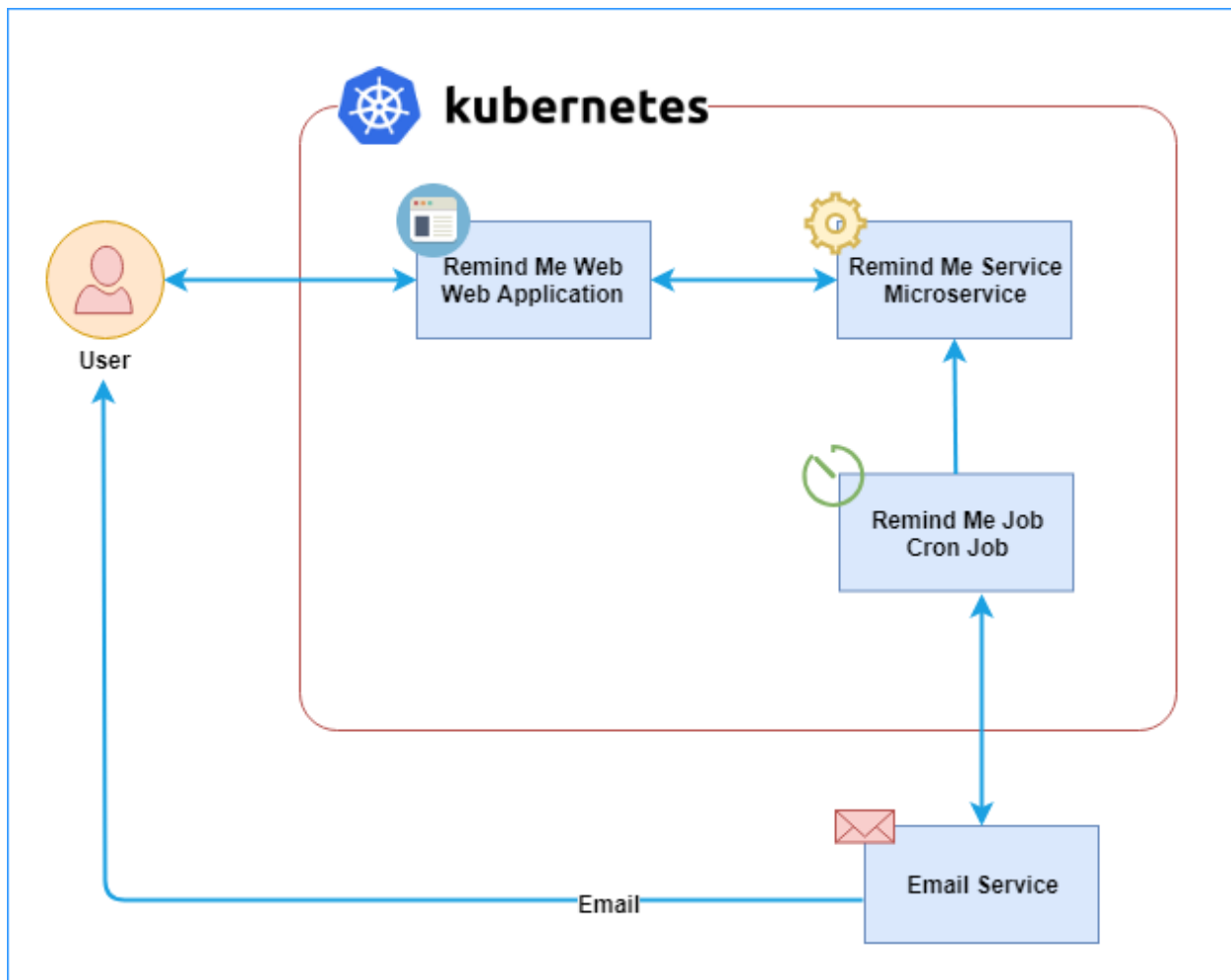


Figure 19: Remind Me logical design diagram

The application itself is straightforward. However, it has most of the components required by almost all enterprise applications in the real world. The following is a brief description of the roles of the various components of the application:

- **Remind-Me-Web:** This is the front end of the application that accepts reminders from the users. A reminder consists of a title, which is the name of the reminder, and a time when the user should be notified.
- **Remind-Me-Service:** This is a simple microservice that performs CRUD (Create, Read, Update, Delete) operations on reminders. The service uses a persistent volume (see Chapter 1) to save the data.
- **Remind-Me-Job:** This is a daily scheduled job, which sends an email to the user if it finds any reminders scheduled for the day when this job is triggered.

The technology and the logic we used for building the applications are not essential. You can build these components using any platform and technology stack and deploy the application on Kubernetes. We chose the following stack to build our applications. If you don't know anything about one or all these technologies, don't worry, we are not going to refer to the application code anywhere in this book.

Table 2: Remind Me technology stack

Component	Technology
Remind-Me-Web	Angular 5 SPA on Dot Net Core 2.1
Remind-Me-Service	.Net Core 2.1 WebAPI
Remind-Me-Job	.Net Core 2.1 Console Application

We have published the entire source code of all the components and the deployment instructions on GitHub. At relevant places, we will point you toward the repositories on GitHub where you can get the source code.

The first release

It is day one on the project and everyone is excited. In this iteration, we will build only the web application and deploy it on a Kubernetes cluster. Our developers put their heads down and produced a simple application that stores reminders in memory. This doesn't look very exciting right now, but gets the job done.



Note: The source code for this application is available in this [repo on GitHub](#). You can find the steps to build and deploy the application in the repository.

The following screenshot shows the application in action, and it looks good to go for production release.

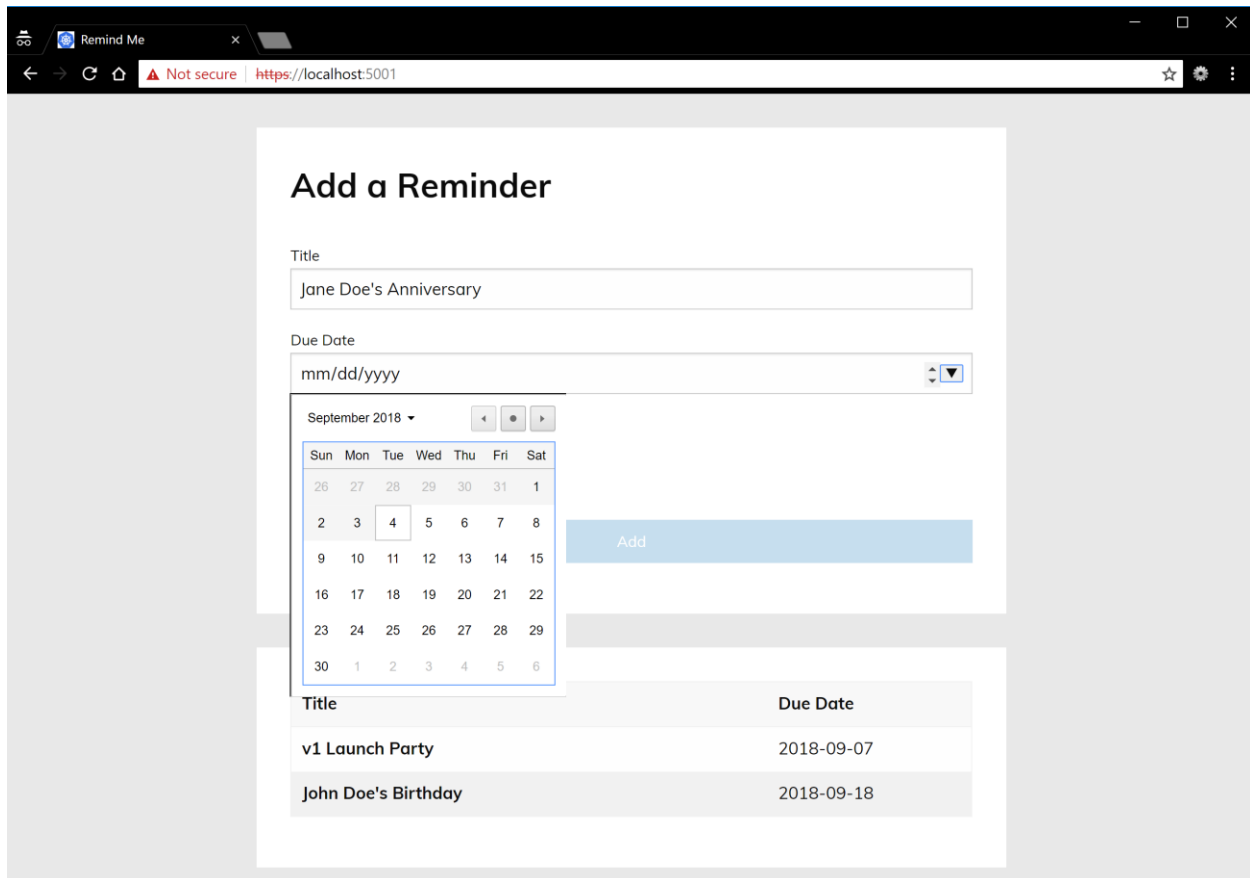


Figure 20: Remind-Me web v1.0 on localhost

The application is packaged in a Docker image using a simple Docker file, available in the [repository](#). Since this is the first release, we tagged the image with version 1.0.0 and released this image in the wild (public repository) by releasing it on the *Kubernetes Succinctly* Docker Hub [repository](#).



Note: You can find all the images for this title on the Docker Hub [repository](#) for *Kubernetes Succinctly*.

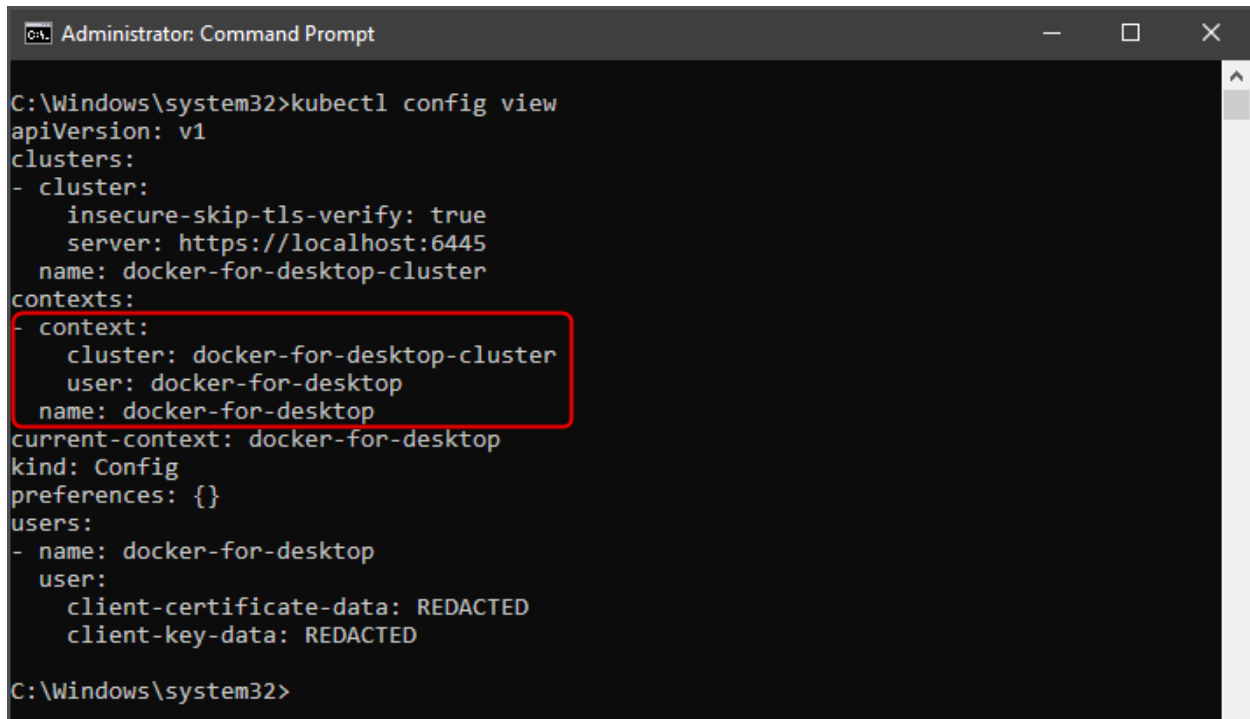
It's now time for deploying this image on our Kubernetes cluster. Let's start with testing the deployment on Docker with Windows (or Minikube, if you prefer) on our machine before deploying the application on a production cluster.

Before we begin issuing commands to the cluster, **kubectl** needs to be told about the cluster so that it can interact with the REST API there. The file that contains the connection information is called the kubeconfig file. By default, **kubectl** searches for a file named config at `$Home/.kube` in Linux or `%userprofile%/.kube/config` in Windows. You can choose your file by setting the custom kubeconfig file path in the **KUBECONFIG** environment variable or by using the **--kubeconfig** flag in your **kubectl** command.

Installing Minikube or Docker on Windows with Kubernetes support adds the connection details of your local cluster in this file. The connection information for a cluster is grouped under a context element in the kubeconfig file. A name uniquely identifies the context element, and it consists of three elements: cluster, namespace (empty for default), and user. To view the config file in your terminal, execute the following command.

Code Listing 20: View cluster config

```
kubectl config view
```



```
C:\Windows\system32>kubectl config view
apiVersion: v1
clusters:
- cluster:
    insecure-skip-tls-verify: true
    server: https://localhost:6445
    name: docker-for-desktop-cluster
contexts:
- context:
    cluster: docker-for-desktop-cluster
    user: docker-for-desktop
    name: docker-for-desktop
current-context: docker-for-desktop
kind: Config
preferences: {}
users:
- name: docker-for-desktop
  user:
    client-certificate-data: REDACTED
    client-key-data: REDACTED

C:\Windows\system32>
```

Figure 21: Kubectl config view output

If you have multiple clusters available, you can set the current context against which all the further commands will be executed.

Code Listing 21: Change context

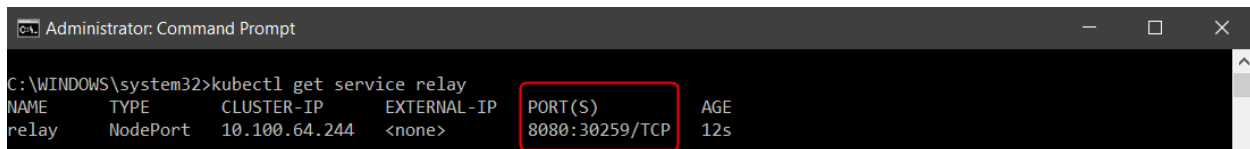
```
kubectl config use-context docker-for-desktop
```

There are two ways to deploy an application to Kubernetes: imperative approach and declarative approach. The imperative approach is to supply container creation options as part of command line arguments and follow up the command with further instructions to set up resources, such as external IP. For example, executing the following commands in sequence will create a pod using Google's echo server image. This application simply relays whatever you pass to it.

Code Listing 22: Deploying image to cluster

```
# Create deployment with Google Echoserver image and expose it on port 8080.
kubectl run relay --image=gcr.io/google_containers/echoserver:1.4 --
port=8080
# Create a service of type NodePort for the relay deployment and expose
the deployment on a random port on the cluster.
kubectl expose deployment relay --type=NodePort
# Get the port where the service is exposed.
kubectl get service relay
```

After executing the last command, you will get an output like the following, which presents the details of the service and the cluster port the service is available on. Notice how port 8080 of the service is mapped to 30722 of the cluster.



NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
relay	NodePort	10.100.64.244	<none>	8080:30259/TCP	12s

Figure 22: Kubectl get service relay output

Try sending a request to the service from the command line or your browser, and see the service relay the request details back to you. You can use the following command for this purpose.

Code Listing 23: Request relay

```
curl http://127.0.0.1:30722/we_love_kubernetes
```

As you can imagine, running a set of commands from the terminal for a large system is error prone and not intuitive, and that's where the declarative approach steps in. Kubernetes supports storing the desired configuration of your system in a YAML or JSON file and applying changes to the cluster using a simple `kubectl apply` command.



Tip: *YAML originally stood for "Yet Another Markup Language," but was redefined as "YAML Ain't Markup Language." If you are not familiar with it, you should read more on the [official website](#). It's easy to understand and supports comments, unlike its JSON counterpart.*

Storing the configuration of your cluster in a configuration file also allows you to track it in a version control system and treat it as a code. You can also create a composite configuration that is made up of several configuration files, which will keep the size of individual configuration files small. The `-f` parameter of the `kubectl apply` command lets you specify the location of your configuration file. The `-R` flag instructs the command to recursively parse the directories at the location to compose the parts of a composite deployment.

Let's now create a cluster configuration file for our application and deploy it on our local cluster. We have decorated key statements with comments to make it easy for you to understand the elements in the file.

Code Listing 24: Remindmeweb YAML

```
# Create a deployment. This will deploy our app on multiple nodes.
apiVersion: apps/v1
kind: Deployment
metadata:
  name: remindmeweb-deployment # name of our deployment.
spec:
  selector:
    matchLabels:
      app: remindmeweb # we'll deploy to all pods that have this label.
  replicas: 2 # run our app on 2 pods, please!
  template: # create pods using pod definition in this template.
    metadata:
      labels:
        app: remindmeweb # Label of the pods created by this template.
    spec:
      containers:
        - name: frontend # name of the container
          image: kubernetes Succinctly/remind-me-web:1.0.0
          ports:
            - containerPort: 80 # port on which the service is running.
              protocol: TCP
---
# Describe a service that consists of our web app pods.
apiVersion: v1
kind: Service
metadata:
  name: remind-me-web-svc
  labels:
    app: remind-me-web-svc
spec:
  type: NodePort # type of service.
  ports:
    - port: 8080 # any service in the same namespace can talk to this
      service using this port.
      protocol: TCP
      targetPort: 80 # our web application is running on this port in the
        pod. By default, targetPort = port.
      nodePort: 31120 # external users can access this service on port
        31120 using kube-proxy.
  selector:
    app: remindmeweb
```

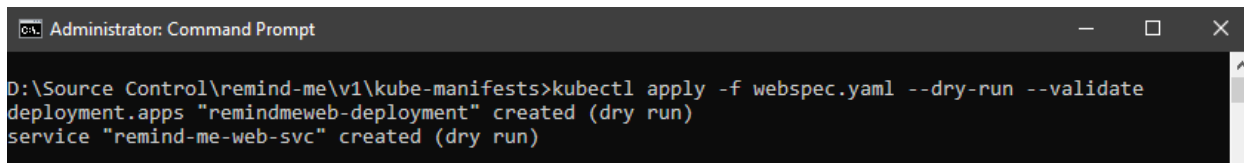
This manifest is available in the GitHub [repository](#) for this application. We now need to hand this manifest over to our Kubernetes cluster to execute it.

Kubernetes allows us to validate a configuration file for issues such as missing fields before deploying it by using the `--validate` option. The `--dry-run` option will show you the object that will be sent to the cluster to provision. Let's first validate the configuration by executing the following command from the shell. Before executing the command, change to the **kube-manifests** folder.

Code Listing 25: Validating YAML

```
kubectl apply -f webspec.yaml --dry-run --validate
```

After running the command, you will see an output like the following. Otherwise, you will see an error that you can fix, and then validate the configuration again.



```
D:\Source Control\remind-me\v1\kube-manifests>kubectl apply -f webspec.yaml --dry-run --validate
deployment.apps "remindmeweb-deployment" created (dry run)
service "remind-me-web-svc" created (dry run)
```

Figure 23: Output YAML validation

Finally, to deploy the application, execute the following command.

Code Listing 26: Deploying application to cluster

```
kubectl apply -f webspec.yaml
```

You can also use the `kubectl create` command to apply the previous specification to your cluster.



Note: `kubectl apply` is part of the declarative approach, where changes applied are maintained even after applying more changes. `kubectl create` is another command that can be used, but it is part of the imperative approach, where you directly tell the Kubernetes API how you want to create or delete your resource, depending upon the configuration.

After the deployment succeeds, you can inspect the status of deployment by executing the following command.

Code Listing 27: Deployment status

```
kubectl describe deployment/remindmeweb-deployment
```

If the deployment rolls out correctly, then you should see an output like the following.

```
Administrator: Command Prompt

D:\Source Control\remind-me\v1\kube-manifests>kubectl describe deployment/remindmeweb-deployment
Name: remindmeweb-deployment
Namespace: default
CreationTimestamp: Sun, 16 Dec 2018 17:18:10 +0530
Labels: <none>
Annotations: deployment.kubernetes.io/revision=1
             kubectl.kubernetes.io/last-applied-configuration={"apiVersion":"apps/v1","kind":"Deployment",
"metadata":{"annotations":{"name":"remindmeweb-deployment","namespace":"default"},"spec":{"replicas":2,"s...
Selector: app=remindmeweb
Replicas: 2 desired | 2 updated | 2 total | 2 available | 0 unavailable
StrategyType: RollingUpdate
MinReadySeconds: 0
RollingUpdateStrategy: 25% max unavailable, 25% max surge
Pod Template:
  Labels: app=remindmeweb
  Containers:
    frontend:
      Image: kubernetestsuccinctly/remind-me-web:1.0.0
      Port: 80/TCP
      Host Port: 0/TCP
      Environment: <none>
      Mounts: <none>
      Volumes: <none>
Conditions:
  Type           Status  Reason
  ----           -
  Available      True    MinimumReplicasAvailable
  Progressing    True    NewReplicaSetAvailable
OldReplicaSets: <none>
NewReplicaSet: remindmeweb-deployment-7bc957566c (2/2 replicas created)
Events:
  Type           Reason             Age   From                      Message
  ----           -
  Normal        ScalingReplicaSet   35s   deployment-controller     Scaled up replica set remindmeweb-deployment-7bc957566c to 2
```

Figure 24: Output deployment status

Your application should be available on <http://localhost:31120> (the node port) at this time. You can also view the details of your service by using the following command.

Code Listing 28: Get service details

```
kubectl get service remind-me-web-svc
```

This command will produce a response like the following, which shows the port on which your service is available.

```
Administrator: Command Prompt

D:\Source Control\remind-me\v1\kube-manifests>kubectl get service remind-me-web-svc
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
remind-me-web-svc  NodePort    10.96.67.191  <none>         8080:31120/TCP   14m
```

Figure 25: Output get service

Congratulations, you have finished deploying the application to a cluster.

Deconstructing YAML configurations

A YAML file consists of only two types of structures: maps and lists. Apart from the two primitive structures, you will find different combinations of the structures in a standard configuration, such as lists of lists, maps of lists, lists of maps, and so on.

YAML maps are used to associate names with values. You will find that several elements in a standard YAML, such as **apiVersion** and **kind**, are described in this format. For example, the following configuration describes two keys and their respective values.

Code Listing 29: Simple maps in YAML

```
apiVersion: v1
kind: Pod
---
```

Three dashes in a row denote a separator. They are used to demarcate one structure from another in the same file.

You can extend the values in the maps to be another map, such as the one in the **metadata** section of the specification that we previously applied.

Code Listing 30: Maps with complex values

```
metadata:
  name: remind-me-web-svc
  labels:
    app: remind-me-web-svc
```

The metadata helps users or services select the objects in Kubernetes with which they want to work. For example, we used the metadata property named **app** to select all the pods that would be part of our service—**remind-me-web-svc**. As you can see in the previous code listing, the key named **metadata** has another key named **label** as its value. Since the keys **name** and **label** are at the same indentation level, they belong to the same parent, **metadata**. You should avoid using tabs to indent objects in YAML.

A list in YAML is just a sequence of objects. A simple list containing string values looks like the following.

Code Listing 31: Simple lists in YAML

```
args
- arg1
- arg2
- arg3
```

The elements of the list are very easy to recognize because they start with a hyphen (-). Just like maps, a list can contain complex objects as values. For example, in the specification that we previously applied, you can see that values of the **containers** element are maps, and values such as **ports** are lists.

Code Listing 32: Lists with complex values

```
containers:
  - name: frontend # name of the container.
    image: kubernetes Succinctly/remind-me-web:1.0.0
    ports:
      - containerPort: 80 # port on which the service is running.
        protocol: TCP
```

You can see that using just two structures, lists and maps, we can create multiple complex structures with YAML.

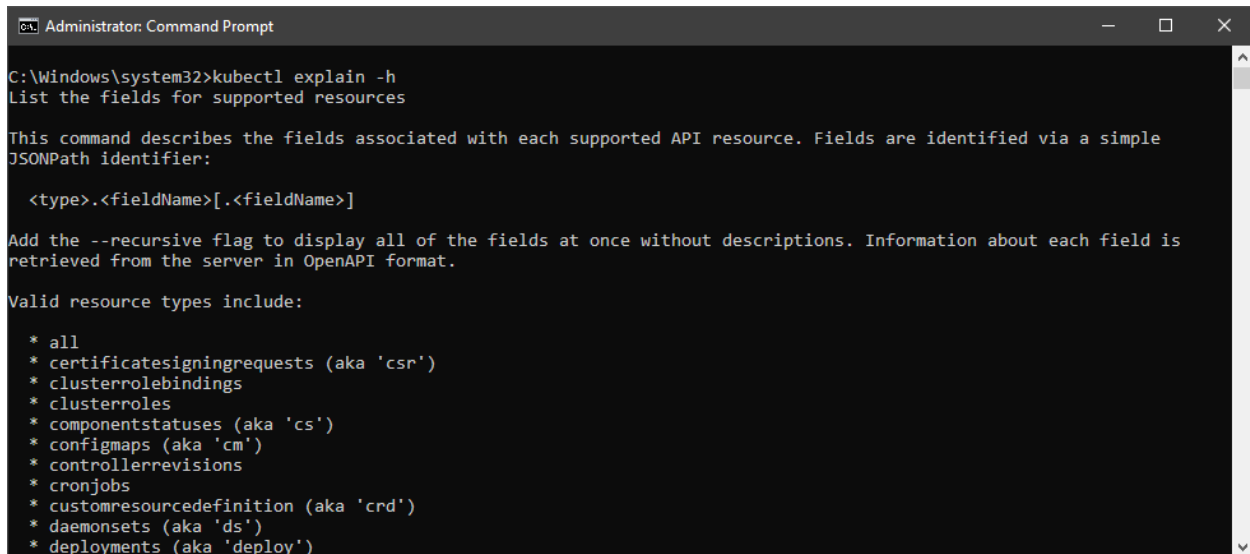
Now that you understand the structure of YAML configurations, you might want to know the various fields that you can configure for a Kubernetes object and what they do. The Kubernetes command line tool, **kubectl**, has an excellent, built-in command called **explain** that you can use as a guide.

The following command will list the resources that you can get further information on.

Code Listing 33: kubectl explain

```
kubectl explain -h
```

The following output generated by this command lists the resources that you can get further help on. We trimmed the output of the command in the following image to save space.



```
Administrator: Command Prompt
C:\Windows\system32>kubectl explain -h
List the fields for supported resources

This command describes the fields associated with each supported API resource. Fields are identified via a simple
JSONPath identifier:

    <type>.<fieldName>[.<fieldName>]

Add the --recursive flag to display all of the fields at once without descriptions. Information about each field is
retrieved from the server in OpenAPI format.

Valid resource types include:

* all
* certificatesigningrequests (aka 'csr')
* clusterrolebindings
* clusterroles
* componentstatuses (aka 'cs')
* configmaps (aka 'cm')
* controllerrevisions
* cronjobs
* customresourcedefinition (aka 'crd')
* daemonsets (aka 'ds')
* deployments (aka 'deploy')
```

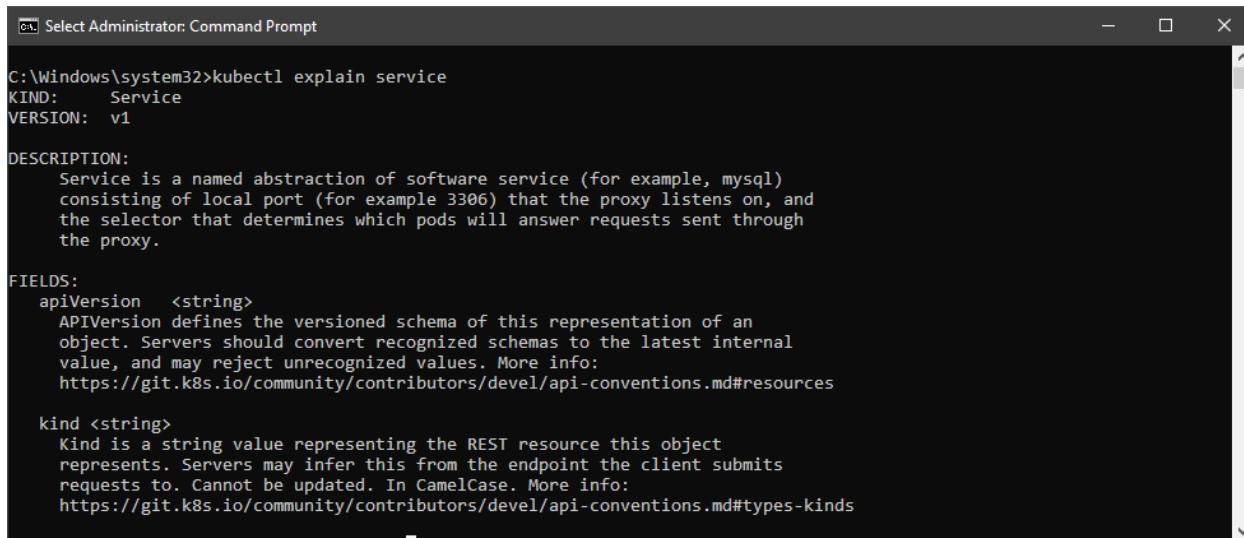
Figure 26: Output explain help

If we want to understand which fields are available to us to configure a service, we can use the same command with an argument.

Code Listing 34: Explain Service

```
kubectl explain service
```

This will generate a concise documentation of the **service** object and the various fields that we can configure. We trimmed the output of the command in the following image to save space.



```

C:\Windows\system32>kubectl explain service
KIND:      Service
VERSION:   v1

DESCRIPTION:
  Service is a named abstraction of software service (for example, mysql)
  consisting of local port (for example 3306) that the proxy listens on, and
  the selector that determines which pods will answer requests sent through
  the proxy.

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

```

Figure 27: Output Explain Service

If you want more details, you can use the **recursive** flag with the **explain** command, which will list all the fields and subfields.

Code Listing 35: Explain service recursive

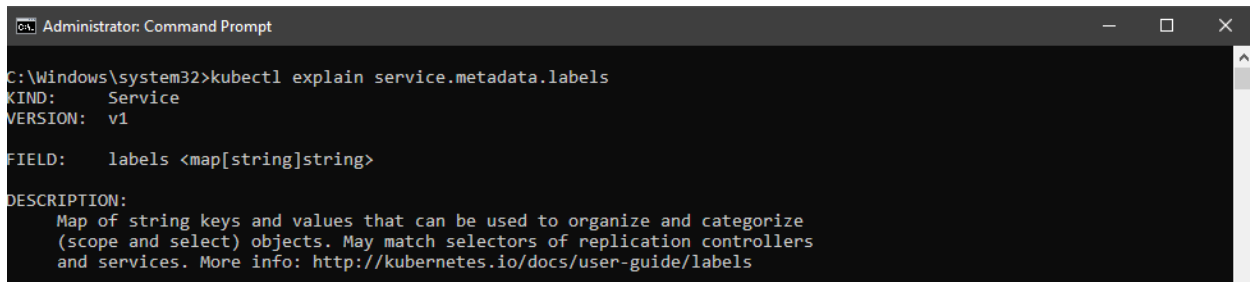
```
kubectl explain service --recursive
```

The **explain** command can give you details for a particular field by specifying a scoped argument. You can scope the argument using dots in the format: **parent.child1.child2**, and so on. For example, to get further details about the **label** field, which is a child of the **metadata** field, we can execute the following command.

Code Listing 36: Explain child elements

```
kubectl explain service.metadata.label
```

As shown in the following output, executing the previous command will generate helpful documentation for the field.



```
Administrator: Command Prompt
C:\Windows\system32>kubectl explain service.metadata.labels
KIND:      Service
VERSION:   v1

FIELD:     labels <map[string]string>

DESCRIPTION:
  Map of string keys and values that can be used to organize and categorize
  (scope and select) objects. May match selectors of replication controllers
  and services. More info: http://kubernetes.io/docs/user-guide/labels
```

Figure 28: Output explain child elements

This tool will be very helpful to you during development or exploration of the Kubernetes platform.

Summary

In this chapter, we discussed how we can deploy a simple stateless application to a Kubernetes cluster. We also set the stage for the application that we will be iteratively building throughout this book. We hope that you are enjoying your journey so far, and having fun deploying applications on Kubernetes.

In the next chapter, we will cover some of the monitoring and logging options available on Kubernetes. We will then proceed to enhance our application and give it a stateful back end.

Chapter 4 Stateful Microservices

Since our first increment did not persist any data, it wasn't very useful for our customers. The next high-priority task assigned to our team is to add state to the application. Therefore, in this iteration, our team will build a stateful back end for our application. In this chapter, we will learn several useful concepts that will help you deploy and configure stateful applications on Kubernetes.

To persist temporary data, you can use the storage attached to a container. However, storage in a container is ephemeral, and any data saved in the container will be lost between container restarts. Saving data in a store outside of the container solves many challenges associated with sharing data between containers and maintaining data beyond the lifetime of containers.

Stateful sets

Stateful microservices require one or more of the following characteristics:

- Unique network identifiers that remain the same across pod scheduling operations.
- Persistent storage.
- Ordered deployment and ordered scaling.
- Ordered and automated rolling updates.

The **StatefulSet** controller manages a group of pods while providing each pod a stable and unique identity, and specific scaling order. Just like the **Deployment** or **ReplicaSet** controllers, the **StatefulSet** controller takes a **StatefulSet** object as input and makes cluster changes to reach the desired state.

StatefulSet has a dependency on a particular type of service that needs to be provisioned before it, called a headless service. A headless service doesn't have a cluster IP, but it provides network identity in the form of individual endpoints to all the pods that are associated with it. A client connecting to a pod will need to specify the endpoint of the specific pod that it wants to access.

Let's create a stateful microservice now to understand it in detail. As we previously discussed, we need to create a headless service before we create a stateful set. The following code listing will create a headless service for us. This looks a lot like the previous service that we created, with the difference being that we ask Kubernetes not to provide a cluster IP to this service (which is what makes this a headless service).

Code Listing 37: Headless service

```
# Describe a headless service that consists of our web API pods.
apiVersion: v1
kind: Service
metadata:
  name: remind-me-api-svc
```

```

labels:
  app: remind-me-api-svc
spec:
  ports:
    - port: 8080 # service will accept traffic on port 8080.
      protocol: TCP
      targetPort: 80 # traffic on port 8080 will be forwarded to port 80
of service.
  clusterIP: None # this makes it a Headless Service.
  selector:
    app: remindmeapi
---
```

Next, we will create a **StatefulSet** for our service. Observe the following spec fragment that will create a **StatefulSet** for our API. We have deliberately left out some parts of the spec to make it easier to understand.

Code Listing 38: StatefulSet

```

# Create a stateful set.
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: remindmeapi-statefulset # name of our stateful set.
spec:
  serviceName: "remind-me-api-ss" # domain name of stateful service
  selector:
    matchLabels:
      app: remindmeapi # We'll deploy to all pods that have this label.
  replicas: 3 # run our app on 3 pods, please!
  template: # create pods using pod definition in this template.
    metadata:
      labels:
        app: remindmeapi # Label of the pods created by this template.
    spec:
      containers:
        - name: backend # name of the container.
          imagePullPolicy: IfNotPresent
          image: kubernetestsuccinctly/remind-me-api:1.0.0
          ports:
            - containerPort: 80 # port on which the service is running.
              protocol: TCP
---
```

You might have noticed that we are using a policy for getting images from the container registry whose value we have specified in the **imagePullPolicy** field. This field affects how **kubelet** attempts to pull an image from the container registry. The following are the various values that this field can take:

- **Never:** The image exists locally and **kubelet** should not attempt to pull the image.
- **Always:** **kubelet** should pull the image every time the pod is restarted. This setting helps during development, as you don't need to provide a new tag for the image every time you fix a bug and build a new image. If you don't supply the image pull policy and don't specify a tag except **latest** for the container image, then the policy **Always** is applied.
- **IfNotPresent:** **kubelet** will attempt to pull the image if it is not locally available. If you omit the policy in the specification and supply a tag for your image, except the **latest** tag, then the policy **IfNotPresent** is applied.

To view the API that you just deployed, you can use the **proxy** verb. **proxy** is quite helpful in debugging deployed services that don't have an external IP. In your terminal, execute the following command to start the proxy.

Code Listing 39: kubectl proxy

```
kubectl proxy
```

This command creates a proxy that we can access on localhost:8001.



Tip: By default, *kubectl proxy* works on the local machine only. If you are running a remote cluster, then you need to enable remote access in the desired service, or have cluster credentials.

To access your service running on the pod **remindmeapi-statefulset-0** via the proxy, you can navigate to the following URL on your browser:

<http://localhost:8001/api/v1/namespaces/default/pods/remindmeapi-statefulset-0/proxy/index.html>

This URL will take you to the [Open API specification](#) of the API (v1), which looks like the following.

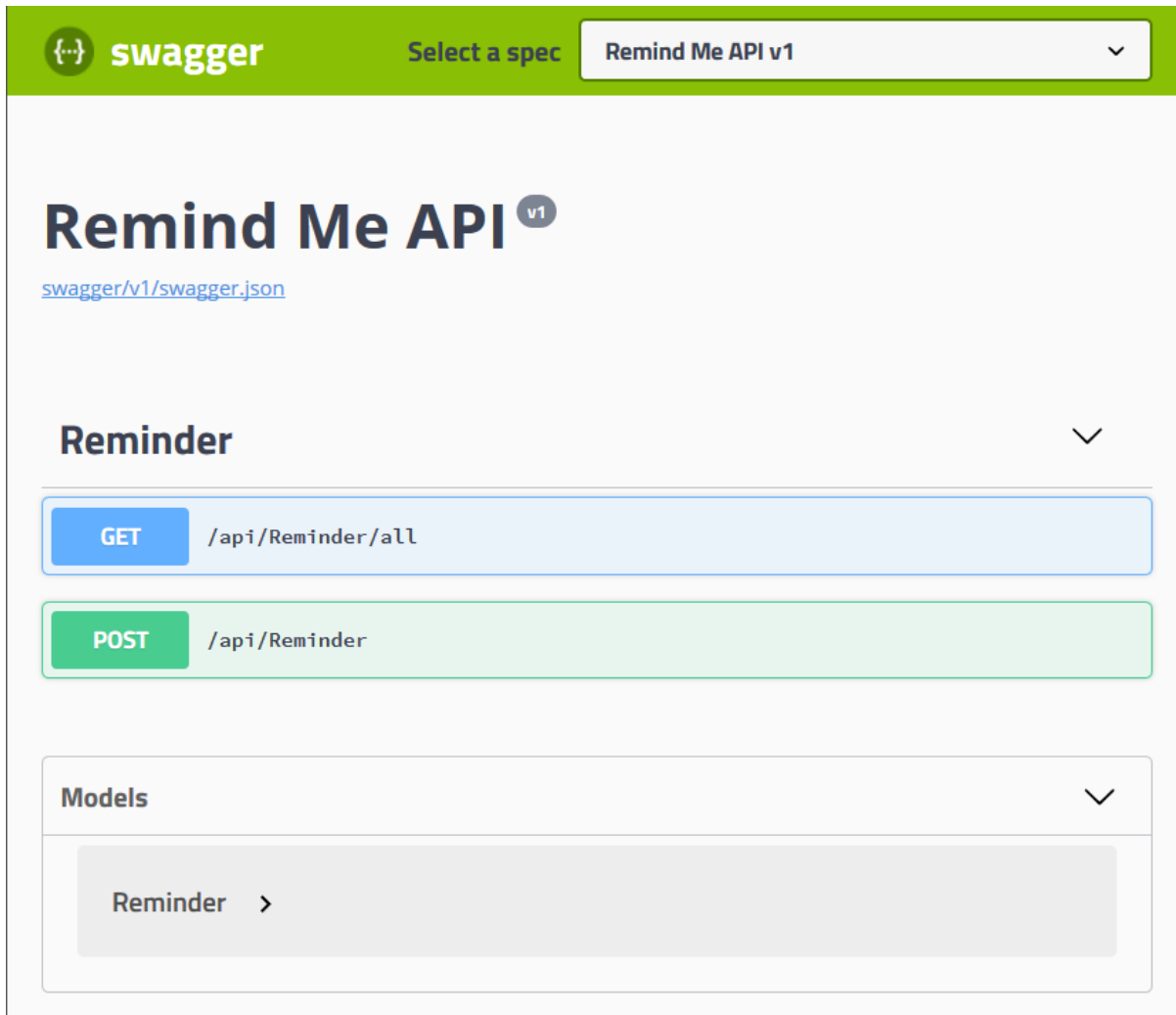


Figure 29: Remind Me API v1 open API definition

We have accessed this service through a proxy. However, our use case is to enable this service to communicate with the web application, and for that purpose we need to get the Cluster DNS record that points to this service. Let's quickly discuss how we can interact with the Kubernetes DNS service to find out the DNS records that map to the location where our service is available.

Volumes

The concept of volumes is easy to understand. They are directories with or without data that are accessible to the container. There are different types of volume options available, each with their unique characteristics. Some of the popular volumes available are:

- **Downward API:** This is used to expose pod configuration to the container.
- **emptyDir:** This can be used to provide temporary storage to the containers in your pods. It can be backed on any medium available to the local node.
- **secret:** A secret volume is an in-memory directory that is used to pass sensitive information such as passwords to pods.

To use a volume, you need to specify it under the **volumes** section in your specification, and reference it under a **volumeMounts** section of your container to specify where you want to use it. Here is a small YAML snippet to illustrate the specification.

Code Listing 40: Specifying volumes

```
spec:
  containers:
    - name: name
      image: containerImage
      imagePullPolicy: Always
      ports:
        - containerPort: 5000
      volumeMounts:
        - name: myVolume # name of volume we want to use.
          mountPath: /cache # volume will be available to container at
            this path.
          readOnly: true # volume specifications.
  volumes:
    - name: myVolume # name of volume.
      emptyDir: # type of volume.
      medium: Memory # volume specifications.
```

Volumes persist their data across container restarts and crashes. However, when a pod is deleted from the node, then volumes are also eliminated.

Persistent volume and persistent volume claim

Unlike volumes, persistent volumes can retain your data independently of your pod lifecycle. A persistent volume is backed by persistent storage that may live outside of the cluster. Some of the most popular persistent volumes use cloud storage, for example, **awsElasticBlockStore**, **azureDisk**, or **gcePersistentDisk**, available from public clouds such as Azure, AWS, and GKE.

Kubernetes separates the need for storage by applications from the actual storage implementation by using persistent volume and persistent volume claims. As an application developer, you specify the storage need of your application using a persistent volume claim (PVC). You can think of a PVC as an instruction to the cluster to give the application persistent storage of X capacity with the features that you need.

An administrator, on the other hand, defines one or more storage classes for the cluster. A storage class defines storage policies such as backup policy and quality-of-service for persistent volumes that are available to the cluster. All the volumes available from the cloud have storage classes predefined. For development, Minikube or Kubernetes on Docker for Windows use a particular storage class, which uses the volume of type **HostPath**.

Adding persistent volume to our service

The stateful back-end service that we are building in this increment uses the [LiteDB](#) database, which is a small, embedded NoSQL database. For complex applications, you might want to use SQL Server or Redis back end, but the system requirement would mostly remain the same—use a persistent volume to store data.

Specifying a persistent storage volume for your application requires the developers and operations to provide the following configurations:

Table 3: Persistent volume workflow

Persistent Volume Workflow	
Configure the storage class	Operations
Create persistent volumes	Operations
Create persistent volume claims	Developer
Map persistent volume claim to application	Developer

To support persistent volumes in our application, we will use the simplest persistent volume types—the **HostPath** in our application. Since the storage class for this PV is preconfigured in Minikube (or Kubernetes on Docker), we will skip the step of specifying the storage class for our volume.

Create persistent volume

Add the following code to your application spec to create a **HostPath** persistent volume.

Code Listing 41: HostPath persistent volume

```
# Define a PV.
kind: PersistentVolume
apiVersion: v1
metadata:
  name: remind-me-api-pv-volume
  labels:
    type: local
spec:
  storageClassName: hostpath
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/data"
---
```


This configuration will use the **data** directory on the node to emulate network attached storage. It is a great and lightweight option for building and testing your applications.

Create persistent volume claim

A pod requests **PersistentVolumeClaim** to request persistent storage. Add the following configuration to your spec for creating a PVC.

Code Listing 42: Persistent volume claim

```
# Define a PVC.
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: remind-me-api-pv-claim
spec:
  storageClassName: hostpath
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
---
```

The configuration request simply states that it requires 3GB of storage with read/write access to a single node. Different types of volumes can support different types of access modes, and you can use any of the supported access modes in your claim. For example, NFS (a type of volume) can support multiple read/write clients. However, you can create a read-only persistent volume claim on NFS so that the pod cannot write data to the volume.

The following access mode values are currently supported:

- **ReadWriteOnce (RWO)**: A single pod can mount this volume, and can read and write data to it.
- **ReadOnlyMany (ROX)**: Many pods can mount this volume, but only read data from it.
- **ReadWriteMany (RWX)**: Many pods can mount this volume and read and write data to it.

Be sure to check the documentation of the persistent volume to see what it can support.

Map persistent volume claim to the application

Finally, we will map the persistent volume claim to the application and mount the persistent volume on the container that will host our application.

Code Listing 43: Mapping PVC to application

```
# Create a deployment. This will deploy our app on multiple nodes.
```

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: remindmeapi-deployment # name of our deployment.
spec:
  selector:
    matchLabels:
      app: remindmeapi # we'll deploy to all pods that have this label
  replicas: 1 # run our app on 2 pods, please!
  template: # create pods using pod definition in this template.
    metadata:
      labels:
        app: remindmeapi # Label of the pods created by this template.
    spec:
      containers:
        - name: backend # name of the container.
          imagePullPolicy: IfNotPresent
          image: remind-me-api:1.0.0
          ports:
            - containerPort: 80 # port on which the service is running.
              protocol: TCP
          volumeMounts:
            - name: localvolume
              mountPath: /data
      volumes:
        - name: localvolume
          persistentVolumeClaim:
            claimName: remind-me-api-pv-claim
---

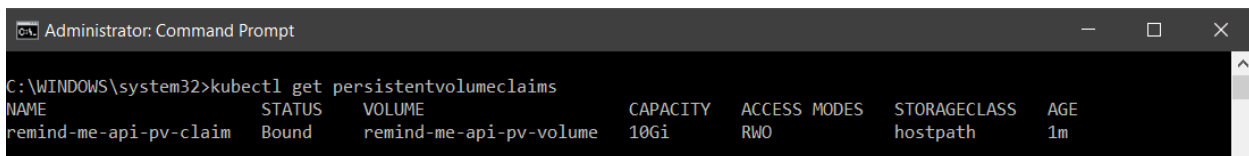
```

You can deploy these changes to your cluster and check the status of resources that you configured by executing the following commands.

Code Listing 44: Get PVC

```
kubectl get persistentvolumeclaims
```

This command will generate an output like the following. You can see that the status of the persistent volume claim is set to **bound**, which means that the persistent volume claim is now bound or attached to your persistent volume.



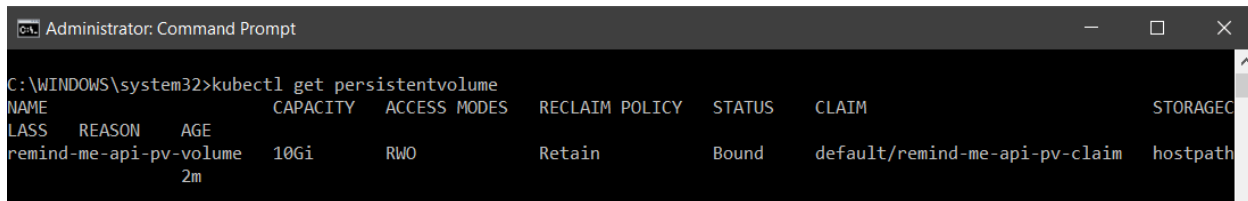
NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
remind-me-api-pv-claim	Bound	remind-me-api-pv-volume	10Gi	RWX	hostpath	1m

Figure 30: kubectl get persistent volume claims

To check the list of persistent volume, use the following command.

Code Listing 45: Get persistent volume

```
kubectl get persistentvolume
```



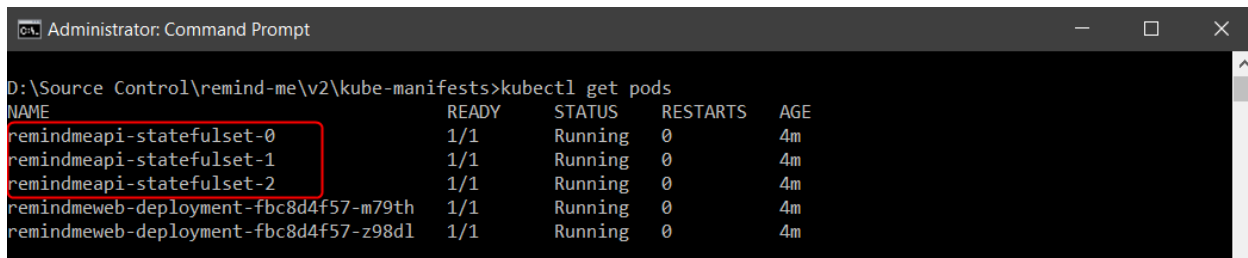
NAME	REASON	AGE	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	CLAIM	STORAGECLASS
remind-me-api-pv-volume		2m	10Gi	RWO	Retain	Bound	default/remin-me-api-pv-claim	hostpath

Figure 31: kubectl get persistent volume

Let's launch an interactive terminal in our container to view the file system. To find out the name of the pod, execute the following command.

Code Listing 46: Get pods

```
kubectl get pods
```



NAME	READY	STATUS	RESTARTS	AGE
remindmeapi-statefulset-0	1/1	Running	0	4m
remindmeapi-statefulset-1	1/1	Running	0	4m
remindmeapi-statefulset-2	1/1	Running	0	4m
remindmeweb-deployment-fbc8d4f57-m79th	1/1	Running	0	4m
remindmeweb-deployment-fbc8d4f57-z98dl	1/1	Running	0	4m

Figure 32: kubectl get pods

This will display all the pods deployed in the default namespace.



Note: If you have used a namespace while creating the service, then you need to specify the namespace tag to the get pods command.

Next, execute the following command to launch an interactive shell on the pod.

Code Listing 47: Launching shell inside container

```
kubectl exec -it remindmeapi-statefulset-0 -- /bin/bash
```



Note: The syntax presented in the previous code listing is a well-known command for launching a shell with Docker.

As shown in the output, execute the commands inside the terminal to see the contents of the **data** directory that we mounted previously.

```
Administrator: Command Prompt - kubectl exec -it remindmeapi-statefulset-0 -- /bin/bash

D:\Source Control\remind-me\v2\kube-manifests>kubectl exec -it remindmeapi-statefulset-0 -- /bin/bash
root@remindmeapi-statefulset-0:/app# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay          59G   3.1G   53G   6% /
tmpfs            64M    0    64M   0% /dev
tmpfs           991M    0   991M   0% /sys/fs/cgroup
overlay          991M   780K   990M   1% /data
tmpfs           991M   4.0K   991M   1% /app/secrets
/dev/sda1        59G   3.1G   53G   6% /etc/hosts
shm             64M    8.0K   64M   1% /dev/shm
tmpfs           991M   12K   991M   1% /run/secrets/kubernetes.io/serviceaccount
tmpfs           991M    0   991M   0% /proc/acpi
tmpfs           991M    0   991M   0% /sys/firmware
root@remindmeapi-statefulset-0:/app# cd ..
root@remindmeapi-statefulset-0:/# cd data/
root@remindmeapi-statefulset-0:/data# ls
reminder.db
```

Figure 33: kubectl launch interactive terminal

Adding secrets to an application

In general, applications use configurations to act as knobs for modifying the application behavior. These configurations are visible to team members and systems internally, and they do not generally pose a risk to the security of the system. Kubernetes supports storing configurations in a Kubernetes resource called **ConfigMap**. **ConfigMap** is a very simplistic system that saves data in the form of key value pairs. It does not support encryption, so it is not suitable for storing sensitive information. You should store well-known configuration values, such as the API version and the name of environment (development, test, production) in **ConfigMap**.

Most of the applications rely on security settings, such as database connection strings and token decryption keys, among other things, for functionality. Such configurations require a more controlled mechanism for safety. Kubernetes supports saving application secrets in a resource that is aptly named **secrets**.

Kubernetes secrets are internally stored in Base64 encoding format, and they are not persisted on disks, but instead present in the memory of the pods. Secrets, just like other resources, are persisted in the **etcd** cluster, and they are not encrypted by default. Therefore, for production clusters, ensure that you use the encrypting-secret-data-at-rest feature of Kubernetes to safeguard confidential information.

Exposing **Secrets** and **ConfigMaps** to a pod is very similar. You can choose to expose secrets as environment variables, or as a file in a volume that can be mounted on the pod. Our application relies on secrets to store a database connection string, for which we use the following configuration.

Code Listing 48: Create secret

```
---
# Create secret.
apiVersion: v1
kind: Secret
metadata:
  name: api-secret
data:
  appsettings.secrets.json:
    'eyJkYlBhdGgiOiAiL2RhdGEvcnVtaW5kZXIuZGIifQ==' # Base 64 encoding of:
    {"dbPath": "/data/reminder.db"}
---
```

Note that we have our secret stored in Base64 encoded format.



Note: *ConfigMaps should only be used for nonsensitive configuration, while for sensitive configuration, you should use secrets.*

To mount this volume to the container, we will use the following (familiar) configuration.

Code Listing 49: Mount secret

```
---
# Create a stateful set.
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: remindmeapi-statefulset # name of our stateful set.
spec:
  serviceName: 'remind-me-api-ss' # domain name of stateful service.
  selector:
    matchLabels:
      app: remindmeapi # we'll deploy to all pods that have this label.
  replicas: 3 # run our app on 3 pods, please!
  template: # create pods using pod definition in this template.
    metadata:
      labels:
        app: remindmeapi # Label of the pods created by this template.
    spec:
      containers:
        - name: backend # name of the container.
          imagePullPolicy: IfNotPresent
```

```

    image: remind-me-api:test
    ports:
      - containerPort: 80 # port on which the service is running.
        protocol: TCP
    volumeMounts:
      - name: secretvolume
        mountPath: /app/secrets
  volumes:
    - name: secretvolume
      secret:
        secretName: api-secret
---

```

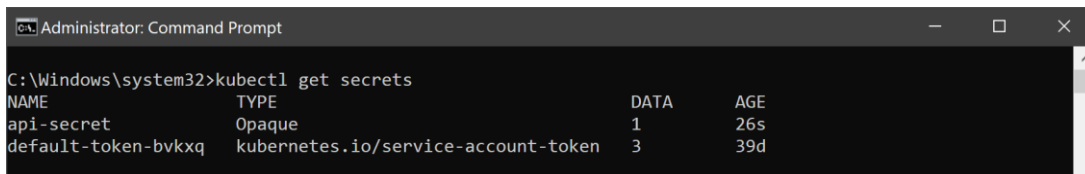
You can check out the complete code listing for this configuration on the companion [GitHub repository](#) for this title.

To get all secrets present in the cluster, you can execute the following command.

Code Listing 50: Get secrets

```
kubectl get secrets
```

This command will generate output like the following. The secret that we described in our specification is called **api-secret**. The additional data fields in the output show the type and number of key-value pairs in the secret.



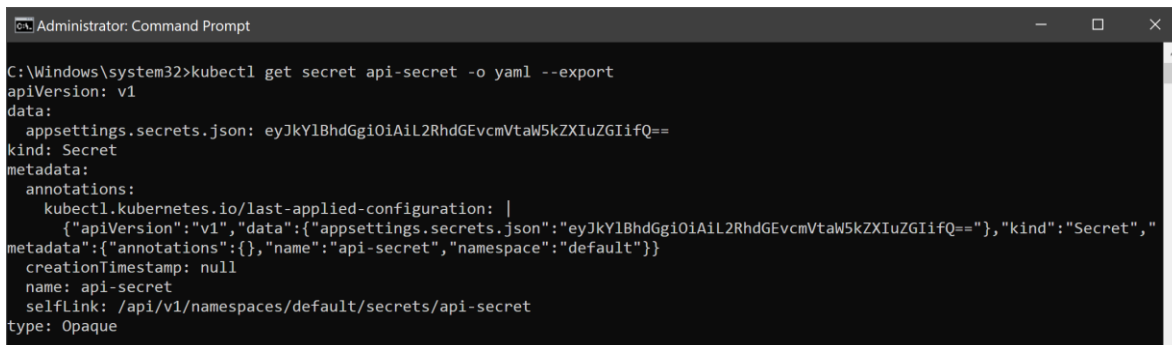
```

C:\Windows\system32>kubectl get secrets
NAME                TYPE          DATA   AGE
api-secret           Opaque        1       26s
default-token-bvqxq  kubernetes.io/service-account-token  3       39d

```

Figure 34: Output get secrets

You can use the following command to see the secret as a Base64-encoded string. You can decode this string to view the secret in plain text.



```

C:\Windows\system32>kubectl get secret api-secret -o yaml --export
apiVersion: v1
data:
  appsettings.secrets.json: eyJkY1BhdGgiOiAiL2RhdGEvcmlVtaW5kZXIuZGIiQ==
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"v1","data":{"appsettings.secrets.json":"eyJkY1BhdGgiOiAiL2RhdGEvcmlVtaW5kZXIuZGIiQ=="},"kind":"Secret","
  metadata":{"annotations":{},"name":"api-secret","namespace":"default"}}
  creationTimestamp: null
name: api-secret
selfLink: /api/v1/namespaces/default/secrets/api-secret
type: Opaque

```

Figure 35: View secret

We are now ready to deploy the new application on our cluster. This time, we have split the specifications into two separate files, **servicepec.yaml** and **webspec.yaml**, for ease of maintenance. Execute the following command to apply all the specifications in the **kube-manifests** folder to your cluster.

Code Listing 51: Apply specification from folder

```
kubectl apply -f kube-manifests/
```

At this point in time, your application will be backed by a stateful API, and the reminders that you save will be persisted in a database. However, you might have noticed that we are not using **StatefulSet** and **ReplicaSet** objects in our specifications, and there is a reason for that: Kubernetes abstractions. Let us discuss how the Kubernetes objects are structured, and which objects you should use for maximum flexibility in building your applications.

Kubernetes abstractions

The Kubernetes API uses abstractions such as services, pods, etc. to help you define the cluster desired state, such as applications that you want to run on the cluster, number of replicas of an application, and so on.

The fine-grained components that define the cluster state are called objects. You can use these objects to define which applications should run on your cluster, the network and disks that should be available to your cluster, and other information about your cluster. Kubernetes objects consist of the following:

- Pod
- Service
- Volume
- Namespace

Kubernetes contains higher-level abstractions as well, called controllers. The controllers internally use the objects and provide additional functionality and convenient features that would otherwise require you to configure multiple objects. For example, in Figure 36 you can see how the controllers build upon other objects as well as other controllers.

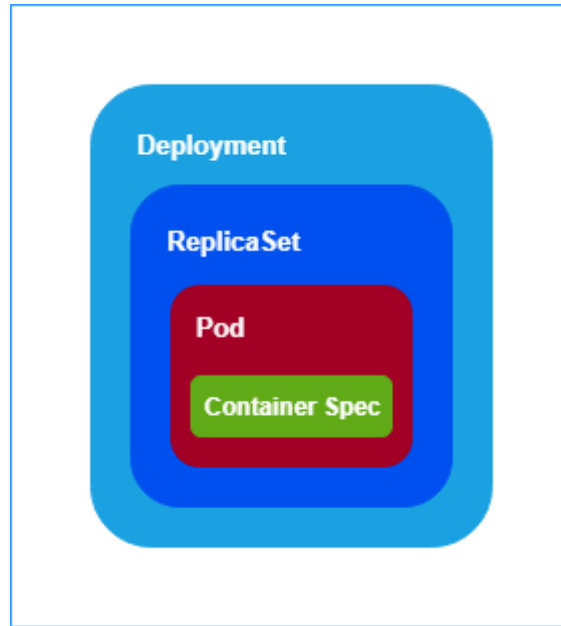


Figure 36: Controllers

Controllers in Kubernetes include the following:

- **ReplicaSet**
- **Deployment**
- **StatefulSet**
- **DaemonSet**
- **Job**

It is recommended to always use controllers rather than objects, unless you have a very specific use case that the controllers don't currently support. For example, the **StatefulSet** controller is an abstraction on a pod object that guarantees ordering and uniqueness of pod objects. Therefore, if you want to build stateful applications, then you should use **StatefulSets** rather than pod objects. However, no controller extends the service object, and therefore, services should be used to expose a group of pods externally.

A service in Kubernetes is a grouping object that defines a logical set of pods and a policy to access the pods that are a part of the set. The pods themselves are another class of objects in Kubernetes. We use services to logically aggregate pods in a cohesive unit in such a manner that any pod connected to a service will respond to a request in a similar manner. The following diagram shows how services are used to aggregate pods in a Kubernetes cluster.

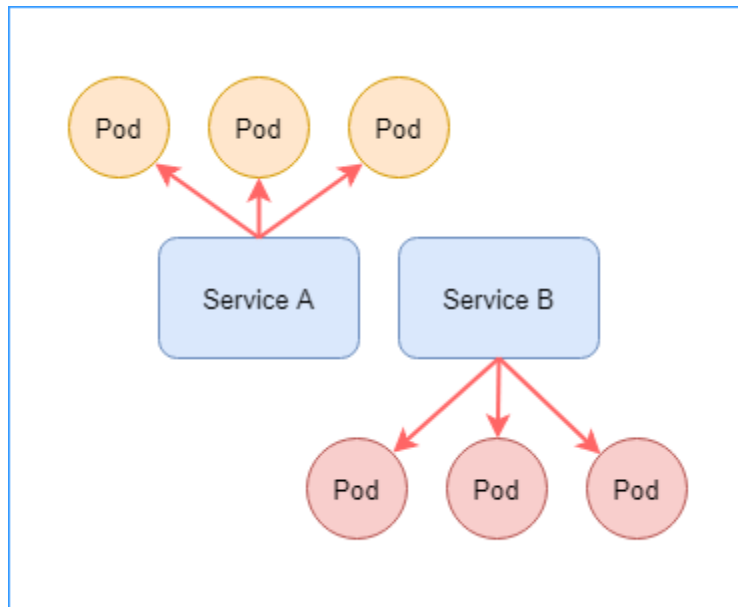


Figure 37: Pod aggregation

Services themselves differ from each other based on how they allow traffic to reach them. The following diagram illustrates how the different types of services accept traffic originating from outside the cluster.

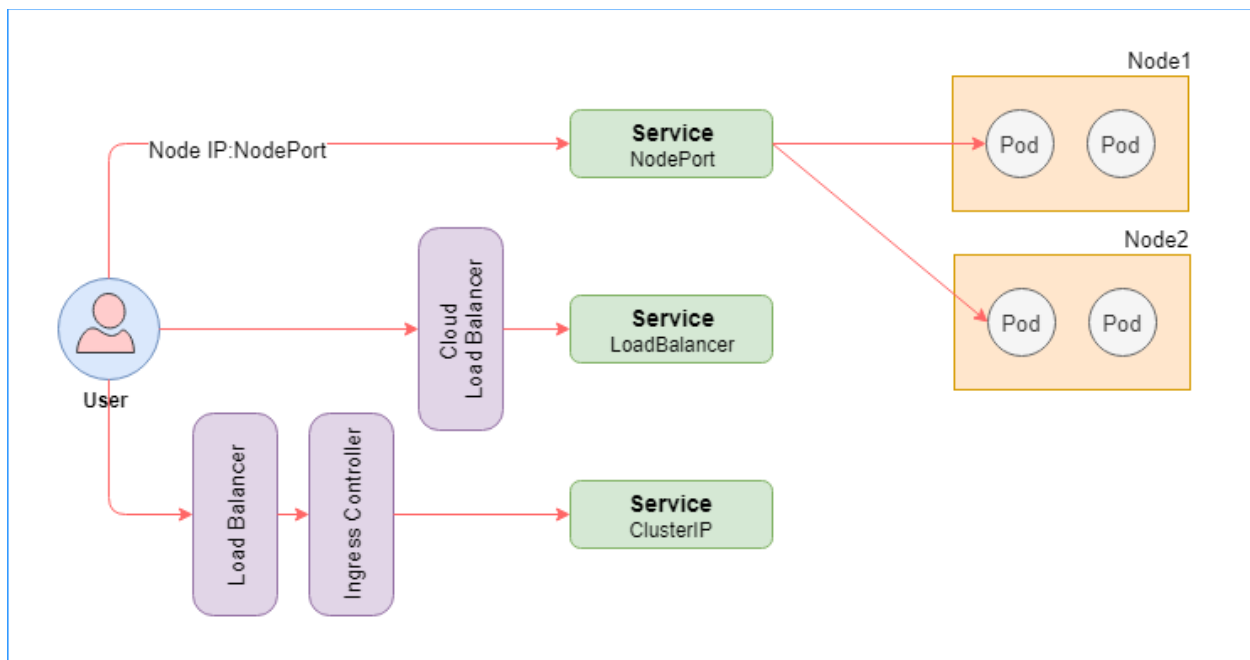


Figure 38: Service types

The most basic service type is the node port. Traffic arriving on a specific port of any node is forwarded to the service, which in turn directs the traffic to a pod in a node. A **NodePort** service is generally used with an external **LoadBalancer** so that the client doesn't need to know about individual node IPs.

The next type of the service is the load balancer. This service leaves the implementation of **LoadBalancer** up to the cloud vendor, such as AWS or Azure. Any traffic that arrives on a certain port of the load balancer will be forwarded to the mapped service. Each service that you expose through the load balancer gets its own address, so this option can get quite expensive if you have a high number of services running in your cluster.

The third type of service is the cluster IP, which is also the default service type. The cluster IP service gets a unique cluster IP address that can be used by other components within the cluster to communicate with the service. By default, traffic originating outside the cluster cannot reach this service. Even though this service is visible only inside the cluster, you might find that it is the most-used service in enterprise applications because it uses a router component called Ingress.

Ingress is a smart router that sits inside the cluster and can accept external traffic. There are many Ingress controllers available that you can use in a cluster, such as Nginx, Contour, and Istio, with each differing in terms of features. Fundamentally, an ingress maps a host such as <https://a.hostname.com> to Service A or <https://hostname.com/b> to Service B. Thus, a single ingress can be used to expose multiple services on the same IP address. Ingress is implemented using the Ingress Controller in Kubernetes.

Ingress provides additional features, such as [TLS termination](#) and Server Name Identification ([SNI](#)), so that they need not be implemented by individual services. Because a single IP address is used by the ingress, you pay only for a single IP address when integrating Ingress with Load Balancer on cloud, making it the most cost-effective option.

Another category of abstractions in Kubernetes depends on the lifecycle of pods. **StatefulSet**, **DaemonSet**, and job controllers collectively form this category.



Figure 39: Kubernetes abstractions

Jobs are used for applications that require a one-off pod to execute a workload, after which the container can be reused for other applications. We will use Jobs for sending out nightly emails to the users of the Remind Me application.

A **DemonSet** controller is spawned on a pod in every node. **DemonSet** is used for deploying background tasks that do not require user input. You will find monitoring and logging daemons, such as **collectd** and **fluentd**, implemented as **DemonSet**.

StatefulSet handles the pod lifecycle events in a different manner to support sequential creation and deletion of pods, amongst other things. They also provide a unique and persistent identity to a set of pods and save pod state data in a persistent storage. Stateful sets are used to build stateful microservices (microservices that persist state data), and they are suitable to deploy applications such as Redis, SQL Server, and Kafka.

Interacting with the Kubernetes DNS service

We discussed in the first chapter that every Kubernetes cluster has a DNS service that is scheduled as a regular pod. Any communication among services happens using the DNS service, which contains records for all services running in the cluster. Often, to connect to a service, you need to know the DNS name of the service. For example, for the application that we are working on, we want to know the DNS name of the **remind-me-api** service so that we can supply it to our web application, **remind-me-web**.

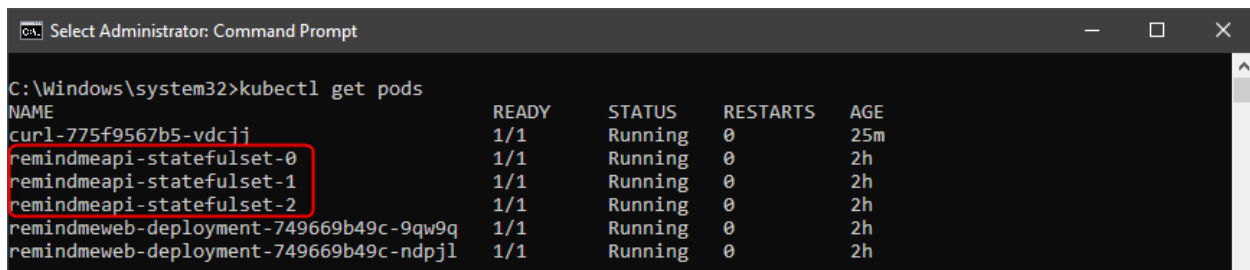
For interacting with Kubernetes, a commonly used trick to interact with resources inside the cluster is to deploy a pod in the cluster that supports running a prompt. [BusyBox](#) is a commonly used tool for this purpose. It is a set of common Linux programs packaged together. One of its most useful features is getting a prompt running inside the cluster.

To create a pod in the cluster, execute the following command in your terminal.

Code Listing 52: BusyBox pod

```
kubectl run curl --image=radial/busyboxplus:curl -i --tty
```

This command will launch the BusyBox terminal, in which you can execute any command in the cluster.



NAME	READY	STATUS	RESTARTS	AGE
curl-775f9567b5-vdcjj	1/1	Running	0	25m
remindmeapi-statefulset-0	1/1	Running	0	2h
remindmeapi-statefulset-1	1/1	Running	0	2h
remindmeapi-statefulset-2	1/1	Running	0	2h
remindmeweb-deployment-749669b49c-9qw9q	1/1	Running	0	2h
remindmeweb-deployment-749669b49c-ndpjl	1/1	Running	0	2h

Figure 40: Output get BusyBox pod status

Let's find out what the DNS name for the **remind-me-web-svc** service is by executing the following command.

Code Listing 53: nslookup

```
nslookup remind-me-web-svc
```

When executed, this command will list the DNS records for our service.

```
Administrator: Command Prompt
C:\Windows\system32>kubectl run curl --image=radial/busyboxplus:curl -i --tty
If you don't see a command prompt, try pressing enter.
[ root@curl-775f9567b5-smjg7:/ ]$ nslookup remind-me-web-svc
Server:      10.96.0.10
Address 1:  10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      remind-me-web-svc
Address 1: 10.102.19.72 remind-me-web-svc.default.svc.cluster.local
```

Figure 41: Output nslookup remind-me-web-svc

As you can see, there is a single DNS name available for our stateless application, **remind-me-web**. Any traffic sent to the address **remind-me-web-svc.default.svc.cluster.local** will reach one of the randomly selected replicas of the service.



Tip: During development, each developer should provision their unique BusyBox pod in the cluster so that they can work independently of each other. If you want to reuse the terminal after a session, then find out the pod ID of the pod running curl (`kubectl get pods`), and then execute this command to relaunch the prompt: `kubectl attach <POD ID> -c curl -i -t`. To clean up the BusyBox deployment, execute this command: `kubectl delete deployment curl`.

We know that pods in a stateful set have a unique ordinal index. If you execute the following command, it will display the various pods that make up your stateful service.

Code Listing 54: Get pods status

```
kubectl get pods
```

In the following command output, you can see that the pods that make up the stateful set have a unique ordinal index in their names.

```
Select Administrator: Command Prompt
C:\Windows\system32>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
curl-775f9567b5-vdcjj              1/1     Running   0           25m
remindmeapi-statefulset-0           1/1     Running   0           2h
remindmeapi-statefulset-1           1/1     Running   0           2h
remindmeapi-statefulset-2           1/1     Running   0           2h
remindmeweb-deployment-749669b49c-9qw9q 1/1     Running   0           2h
remindmeweb-deployment-749669b49c-ndpjl 1/1     Running   0           2h
```

Figure 42: Output kubectl get pods for stateful set

Let's execute the **nslookup** command for our stateful service to see what we get. This time, execute this command in your BusyBox: **nslookup remind-me-api-svc**. This will generate an output like the following.

```
Administrator: Command Prompt - kubectl attach curl-775f9567b5-vdcjj -c curl -i -t
C:\Windows\system32>kubectl attach curl-775f9567b5-vdcjj -c curl -i -t
If you don't see a command prompt, try pressing enter.
[ root@curl-775f9567b5-vdcjj:/ ]$ nslookup remind-me-api-svc
Server:      10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:        remind-me-api-svc
Address 1: 10.1.2.141 remindmeapi-statefulset-0.remind-me-api-svc.default.svc.cluster.local
Address 2: 10.1.2.142 remindmeapi-statefulset-1.remind-me-api-svc.default.svc.cluster.local
Address 3: 10.1.2.143 remindmeapi-statefulset-2.remind-me-api-svc.default.svc.cluster.local
```

Figure 43: Output BusyBox nslookup

As you can see, unlike the stateless service that we provisioned for the web application, each pod of the stateful service is individually addressable. Therefore, the client of a stateful service should implement a load balancing policy to ensure that the requests to stateful service instances are properly distributed.



Tip: There is a much simpler technique for running `nslookup` commands. You can execute the `nslookup` command after opening an interactive terminal to an existing pod using this command: `kubectl exec <running pod name> -it -- /bin/sh`.

To keep things simple, we will send all the requests to the first pod (one with ordinal index 0) of the API service. Let's set the URL of the API endpoint in the environment variable of the web application in the web application specification. You can read the complete code listing in the companion GitHub [repository](#).

Code Listing 55: Front end manifest

```
# Create a deployment. This will deploy our app on multiple nodes.
apiVersion: apps/v1
kind: Deployment
metadata:
  name: remindmeweb-deployment # name of our deployment.
spec:
  selector:
    matchLabels:
      app: remindmeweb # we'll deploy to all pods that have this label.
  replicas: 2 # run our app on 2 pods please!
  template: # create pods using pod definition in this template.
    metadata:
      labels:
        app: remindmeweb # Label of the pods created by this template.
    spec:
      containers:
        - name: frontend2 # name of container.
          image: kubernetessuccinctly/remind-me-web:2.0.0
          resources:
            requests:
              cpu: 100m
```

```

        memory: 100Mi
      limits:
        cpu: 100m
        memory: 100Mi
      env:
        - name: GET_HOSTS_FROM # This variable tells the service to
          find service host information from dns service. You can change this value
          to 'env' to query service hosts from environment variables.
          value: dns
        - name: apiUrl # This environment variable is used by the
          application to connect to the API.
          Value: http://remindmeapi-statefulset-0.remind-me-api-
          svc.default.svc.cluster.local
      ports:
        - containerPort: 80
          protocol: TCP
    ---
    # Describe a service that consists of our web app pods.
    apiVersion: v1
    kind: Service
    metadata:
      name: remind-me-web-svc
      labels:
        app: remind-me-web-svc
    spec:
      type: NodePort # type of service.
      ports:
        - port: 8080 # any service in the same namespace can talk to this
          service using this port.
          protocol: TCP
          targetPort: 80 # our web application is running on this port in
          pod. By default, targetPort = port.
          nodePort: 31120 # external users can access this service on port
          31120 using kube-proxy.
      selector:
        app: remindmeweb

```

This configuration will enable your web application to send requests to the back-end stateful service. This will help preserve the reminders across page refreshes and sessions.

Before concluding this discussion, let's discuss an important aspect of services. The sequence of creation of resources is very important when it comes to defining services. If pods exist before services are defined, then the environment variables for the service will not exist within the pods that are part of the service. So in the previous specification, if we were to set the value of the `GET_HOSTS_FROM` environment variable to `env`, and create the `remind-me-web-svc` web application first and the `remind-me-api-svc` second, then the resolution might fail. This is because the existing pods won't get the environment variables that contain the details of the service `remind-me-api-svc`. In such cases, you will need to recreate containers by scaling the

pod count to 0 and then increasing the count. As a rule, always define and apply service resources before creating other resources.

Summary

In this chapter, we discussed how we can build stateful microservices applications on Kubernetes. Kubernetes abstracts persistent storage volumes from applications and distributes the responsibility to provision and apply persistence capabilities between developers and operations. Abstractions like these make Kubernetes an excellent collaborative platform for both developers and operations.

In the next chapter, we will discuss an important aspect of every enterprise-grade application, which is performing smooth upgrades and scaling applications on demand.

Chapter 5 Upgrades and Scaling

In the previous chapter, we covered how we can develop stateful microservices with persistent volumes. Our application is living happily on production systems, and our users want it to support many other features. However, since the users are too dependent on the application, they don't want any downtime during upgrades. With a set of new features, it is now time for our next deployment. Deploying multiple times to a nonproduction system is not an issue. The challenge comes when your application is already in production, and you have multiple changes to be deployed. You need to take care of many aspects at this point, like application downtime, storage backup and restore, and testing the deployment. However, this isn't new to the software world, and we already have multiple tried-and-tested strategies in place to overcome these issues.

Fortunately, Kubernetes supports many of the best practices out of the box, which you can leverage to accomplish this task.

Application upgrades

Application upgrades are an essential aspect of modern software applications, and it has become critical that this happens with zero downtime and issues. As enterprises are learning the importance of DevOps automation, many strategies/frameworks are getting built, which helps in performing application upgrades. Let's discuss some of these strategies concerning Kubernetes.

Rolling updates

Rolling updates ensures that your application pods are updated incrementally to eliminate downtime. However, this method doesn't consider scenarios where you have breaking changes in the update. It just updates the pod instances one by one, and all requests coming to your application can go to either of the old or new instances. There are a few strategies to implement selective routing of incoming traffic to different versions of an application. We will discuss one such strategy later in this chapter.

The following diagram illustrates the rolling update scenario.

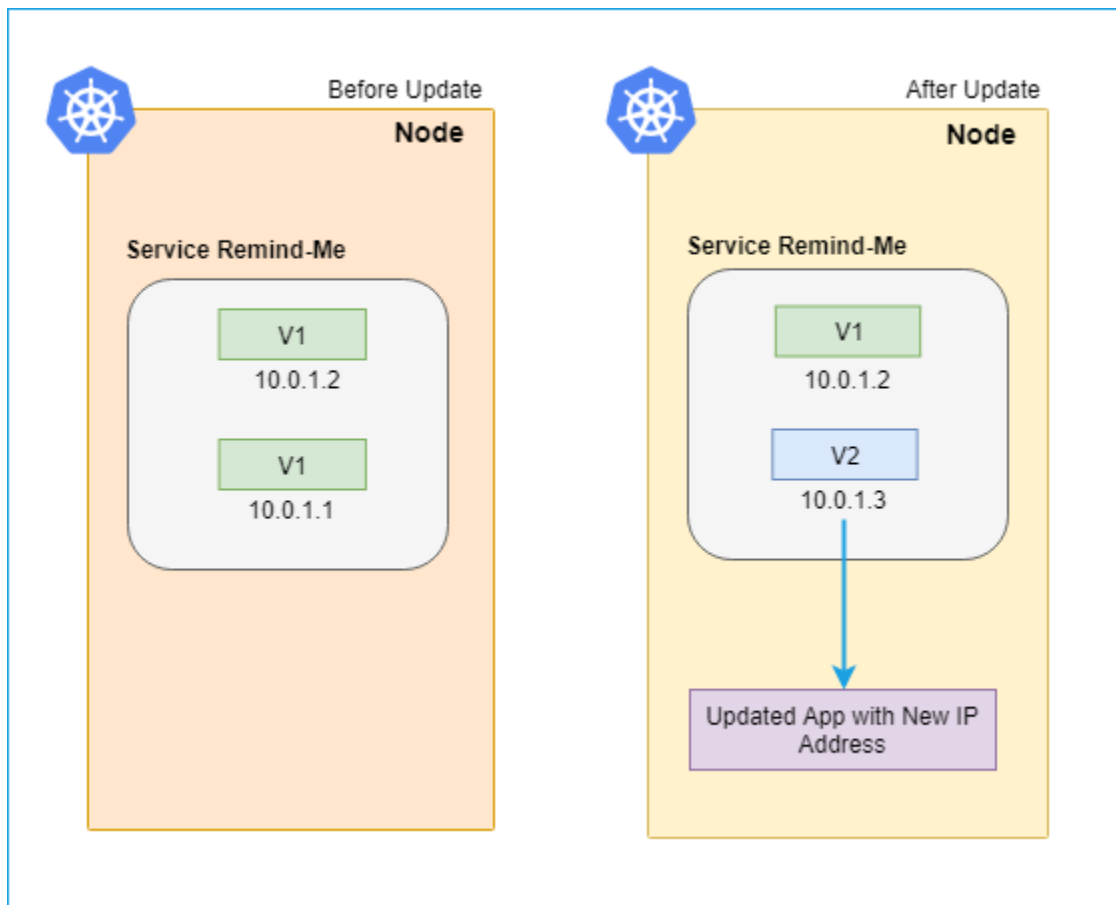


Figure 44: Rolling update

In our API application (**remind-me-service**), we have added support for deleting reminders so that reminders that have already been processed can be removed from the database. We have published a new image version of our service to Docker Hub, and now it's time to upgrade our stateful set to use the updated image. To update the application image in the cluster, we will use the **kubectl set image** command.

Code Listing 56: Update container image

```
kubectl set image statefulsets/remindmeapi-statefulset
backend=kubernetes Succinctly/remind-me-web:2.0.0
```



Note: In the previous command, *backend* is the name of the container. Check the *servicespec.yaml* that we previously applied to build our API for more details.

You can verify the upgrade status of your pods by using the following command.

Code Listing 57: Get pods

```
kubectl get pods
```

We can also use the **rollout status** command to check the status of our deployment.

Code Listing 58: Check rollout status

```
kubectl rollout status statefulsets/remindmeapi-statefulset
```

In the event of an issue in the deployment, **kubectl** provides an easy way to roll back to the last successful deployment.

Code Listing 59: Rollback deployment

```
kubectl rollout undo statefulsets/remindmeapi-statefulset
```

If we have multiple updates to be deployed at the same time, we can also create a new deployment specification and apply that to our cluster. While creating the new specification, we need to specify the deployment strategy (in the **.spec.strategy** attribute) appropriately. There are two types of strategies available:

- **Rolling update strategy:** Ensures pods are updated in a rolling update manner.
- **Recreate strategy:** Ensures all pods are destroyed before new ones are created.



Note: In the rolling update strategy, you can also specify *maxUnavailable* (the maximum number of pods that can be unavailable during the update process in absolute or percentage terms) and *maxSurge* (the maximum number of pods that can be created over the desired number in absolute or percentage terms) to control the update process. The recreate strategy is more suited for the nonproduction environment.

[This is the updated specification](#) that we will use to deploy the new version of the Remind Me service.

To apply this configuration to your cluster, execute the following command in your shell.

Code Listing 60: Updating application in cluster

```
kubectl create -f https://raw.githubusercontent.com/Kubernetes-Succinctly/remind-me-v3/master/kube-manifests/servicespec.yaml --record
```

The flag **record** appended to the end of the command saves the **kubectl** command that you used to apply the update. You will be able to see the recorded commands as the reason for change when you check the rollout history of a deployment. The following is the output of the command that you executed previously.

```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl apply -f https://raw.githubusercontent.com/Kubernetes-Succinctly/remind-me-v3/master/kube-manifests/servicespec.yaml --record
persistentvolume "remind-me-api-pv-volume" configured
persistentvolumeclaim "remind-me-api-pv-claim" configured
secret "api-secret" configured
service "remind-me-api-svc" configured
statefulset.apps "remindmeapi-statefulset" configured
```

Figure 45: Output cluster service upgrade

You can also run the `kubectl get pods` command to see the rolling update in action.

```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
remindmeapi-statefulset-0           0/1     ContainerCreating   0           2s
remindmeapi-statefulset-1           1/1     Running             0           8s
remindmeapi-statefulset-2           1/1     Running             0           13s
remindmeweb-deployment-fbc8d4f57-6bt6x 1/1     Running             0           15m
remindmeweb-deployment-fbc8d4f57-m9t8w 1/1     Running             0           15m
```

Figure 46: Output `kubectl get pods`

As you can see in the output, only one of the three pods is in the **ContainerCreating** state for your service, while the other two are in the **Running** state.

The spec template has optional fields that can be used to define the rolling strategy and tweak based on the requirements.



Tip: The default value of `.Spec.Strategy` is `rollingupdates`, and therefore, it is an optional attribute.

Now, we can check the status of deployment using the following command.

Code Listing 61: Deployment status

```
kubectl rollout status deployments/remindmeweb-deployment
```

To verify whether our service is updated successfully, execute the command `kubectl proxy` and then navigate to the following URL in the browser:

<http://localhost:8001/api/v1/namespaces/default/pods/remindmeapi-statefulset-0/proxy/index.html>

This URL will take you to the [Open API specification](#) of the API (v2), which looks like the following. Notice the delete API we have introduced in this version, as compared to the previous version deployed in chapter 4.

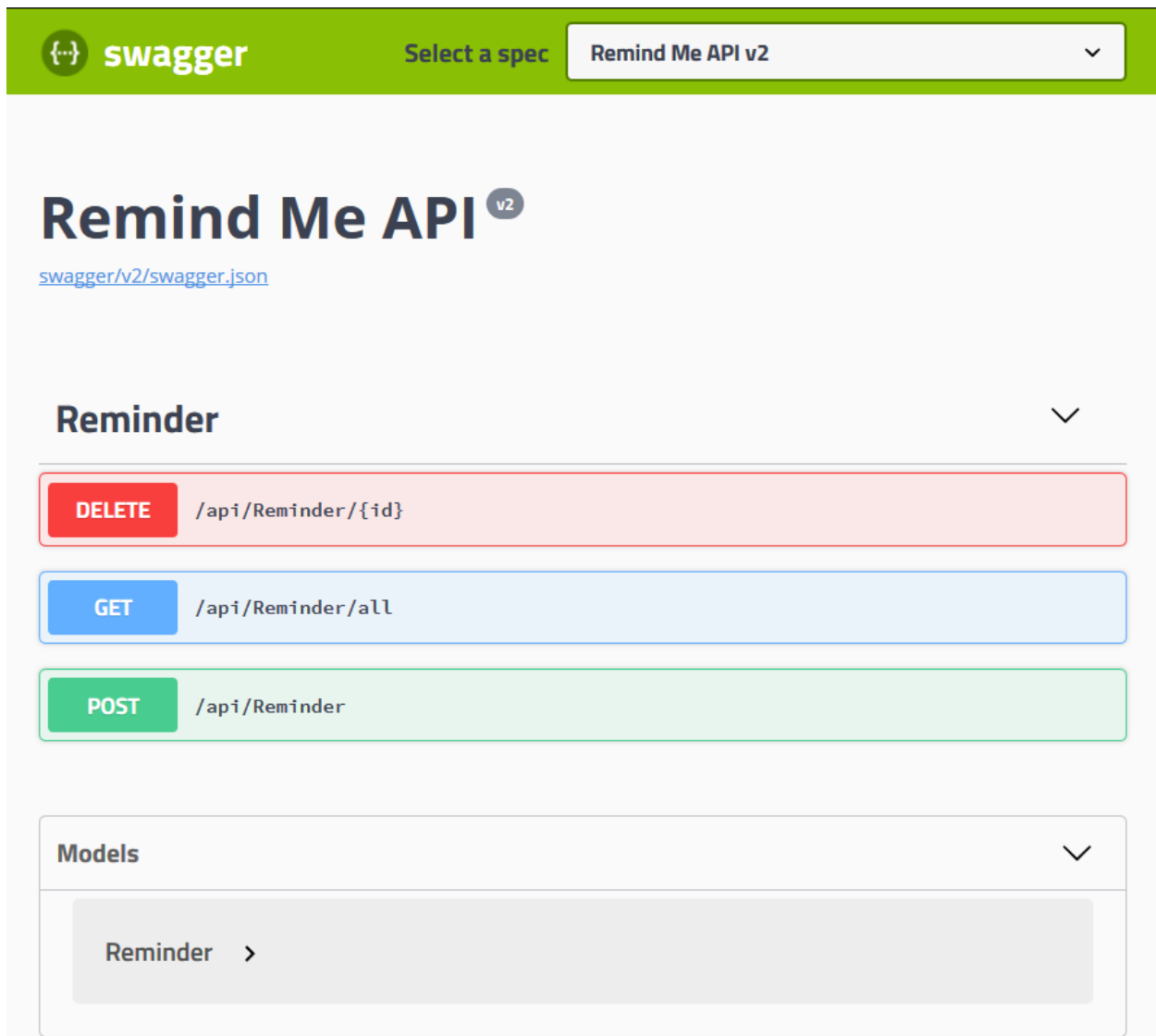


Figure 47: Remind Me API v2 Open API Definition

Congratulations, your cluster is now updated with the latest version of the application. We will use the new version of our API to later build a job that processes the reminders and deletes the processed reminders from the database.

To view the history of all the rollouts you performed on a certain resource, you can execute the following command.

Code Listing 62: Deployment history

```
kubectl rollout history statefulsets/remindmeapi-statefulset
```

The length of history that you want to keep can be controlled by setting the revision history limit.

Blue-green deployments

We have seen how we can specify the recreate strategy to avoid partial update scenarios. However, it may impact application availability for a longer duration, resulting in application downtime. There is a popular strategy that can be used for zero-downtime upgrades, known as blue-green deployments. In the case of blue-green deployments, a new version (green) is created for your application, and the older version (blue) continues to run on the cluster. Once you are done testing the new (green) version of the application, you can switch your endpoints to the new version. This strategy ensures minimum downtime for your application. However, in the case of storage-level breaking changes, you need to take care of migration and data updates for the new deployment as well. This method is costly to begin with, since it involves two sets of deployments, but it ensures that your application availability is maintained in the case of breaking changes.



Tip: You can follow the same steps and infrastructure to set up your green deployment, or you can change it entirely, since it is a separate deployment. This helps in upgrading and degrading infrastructure with minimum downtime.

The following diagram illustrates the blue-green deployment scenario.

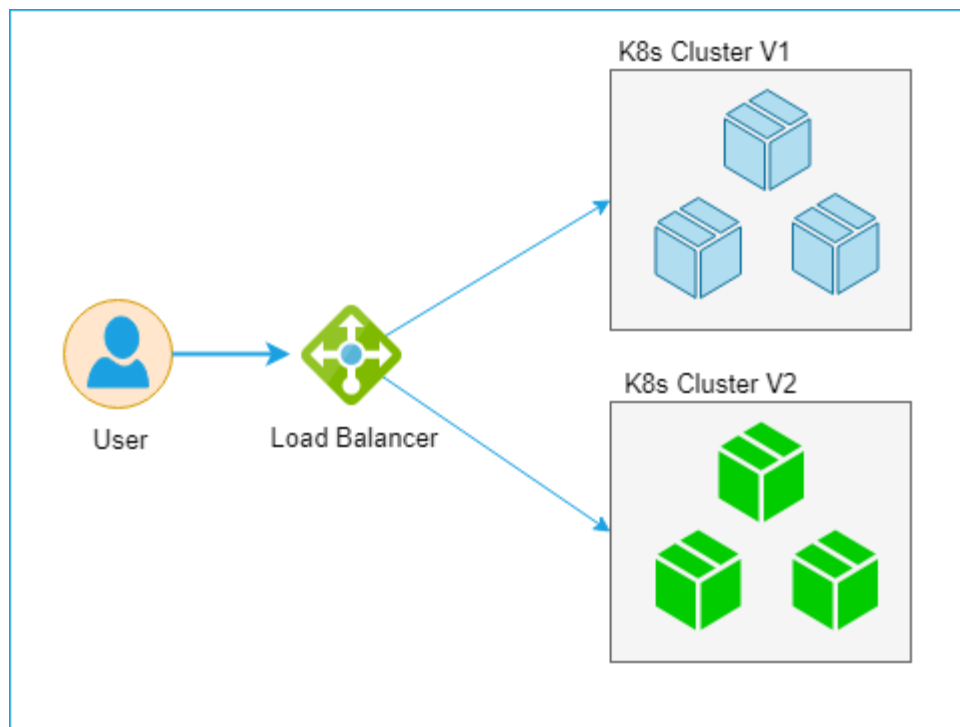


Figure 48: Blue-green deployment

The K8s Cluster V2, which is a green deployment, is thoroughly tested before being attached to the load balancer. Once the green deployment is attached, the blue one is removed from the load balancer, and the updated version of the application is made available to the end users.

This can be set up in a simple manner using Kubernetes services. If we consider the previous scenario, to update our stateful set with the new image, we can leverage the blue-green setup as well. We will create a new stateful set with different selectors, name, and version, and deploy it to the cluster green stateful set. To test it, we can expose a new service and test it separately. Once we are satisfied, we just have to point the existing service to the new selectors, and the existing (blue) stateful set will be replaced with the new (green) setup.

Scaling

Scaling (or scalability) is an essential aspect of modern applications. It refers to how your infrastructure adapts to an increasing or decreasing load by increasing or decreasing the number of instances running your applications, or upgrading the infrastructure to more capable VMs. To save cost, autoscaling, which is the process of scaling an application automatically based on metrics such as user load, has become a de facto feature of most applications.

Scaling can be categorized in two types:

- Horizontal scaling: Add a higher number of resources to your existing infrastructure.
- Vertical scaling: Increase the configuration of your existing infrastructure.

Vertical scaling usually requires manual intervention and is not preferred, due to higher costs and possibility of underutilization. Horizontal scaling, on the other hand, is widely used to make sure your application can handle the spikes in load and can return to a normal state once the load decreases.

In Kubernetes, we have multiple ways to define autoscaling rules for the cluster:

- Using the **kubectl autoscale** command.
- Horizontal pod autoscaling.

kubectl autoscale follows the imperative command style, while horizontal pod autoscaling follows the declarative style, in which you can control the scaling operation via a YAML configuration.

kubectl autoscale

To configure autoscaling for your cluster using the **kubectl autoscale** command, execute the following command in your shell.

Code Listing 63: kubectl autoscale

```
kubectl autoscale deployment remindmeweb-deployment --min=2 --max=4 --cpu-percent=60
```

The command instructs the cluster to have a minimum of two pod instances and to increase the pod instance count when CPU load goes beyond 60 percent, but not to go beyond four instances. You can specify the replication set, replication controller, or deployment in the **autoscale** command.

As you can see, autoscaling is very easy and straightforward to configure. However, in enterprise scenarios, you might want to scale your applications using other metrics apart from CPU utilization. You can achieve this with the Horizontal Pod Autoscaler YAMLs.

Horizontal Pod Autoscaler (HPA)

The Horizontal Pod Autoscaler can also scale the deployment, replication controller, or replica set objects, but cannot be used for objects that cannot be scaled, like daemon sets. By default, it has support for scaling by CPU percentage (v1), but it can be configured to scale on other metrics, like request rate or memory utilization (v2-beta). It can also be configured to scale based on custom metrics.

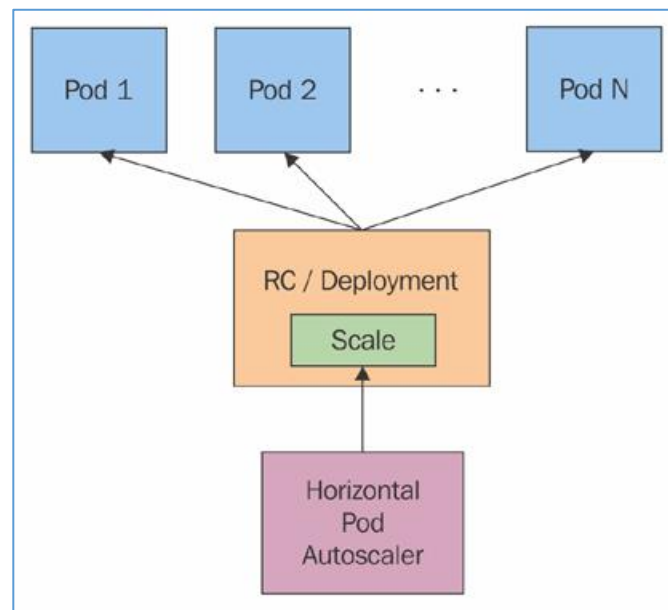


Figure 49: Horizontal Pod Autoscaler. Source: [Kubernetes.io](https://kubernetes.io)

As depicted in Figure 49, the Horizontal Pod Autoscaler doesn't scale the instances by itself. Instead, it controls the instance count, provided there exists a resource like deployment or replication controller. It fetches the metric at a fixed duration (the default is 30 seconds, but can be changed with the flag `--horizontal-pod-autoscaler-sync-period`) from resource metrics API or custom metrics API, and takes appropriate action.

The standard APIs used by the Horizontal Pod Autoscaler are **metrics.k8s.io**, **custom.metrics.k8s.io**, and **external.metrics.k8s.io**. The **metrics.k8s.io** API is usually provided by **metrics-server**, which needs to be launched separately.

For leveraging the full feature set of the Horizontal Pod Autoscaler (v2), the following conditions must be met:

- The API aggregation layer is enabled. This is an additional layer that needs to be enabled explicitly, as it is not part of core Kubernetes APIs. Please refer to the [documentation](#) for more details on how to enable the aggregation layer.

- **Metrics-server** is enabled as a cluster add-on. Please refer to the [documentation](#) for more details.
- The **--horizontal-pod-autoscaler-use-rest-clients** is true or unset. Setting this to **false** switches to Heapster-based autoscaling, which is deprecated (Heapster enables collecting performance metrics for a cluster).
- **custom.metrics.k8s.io** and **external.metrics.k8s.io** are provided by the adaptor API server and must be enabled by the respective providers.

HPA commands

HPA is directly supported by **kubectl**. The following are the set of commands supported by HPA. Create a config file **hpaspec.yaml** with type of **HorizontalPodAutoscaler** and apply it to your cluster using the following commands.

Code Listing 64: Horizontal Pod Autoscaler

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: remindmeautoscaler
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: remindmeweb-deployment
  minReplicas: 2
  maxReplicas: 6
  targetCPUUtilizationPercentage: 60
---
```

Now, you can apply the YAML using the following command.

Code Listing 65: HPA Autoscale

```
kubectl apply -f hpaspec.yaml
```

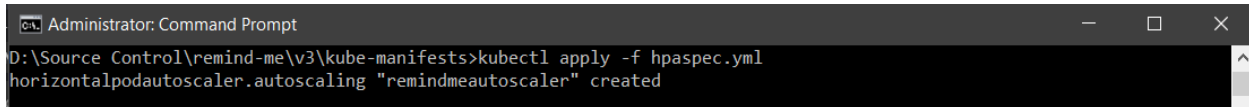


Figure 50: Output create HPA

You can also list the autoscalers active in your cluster using the following command.

Code Listing 66: Get HPA

```
kubectl get hpa
```



```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl get hpa
NAME                REFERENCE                TARGETS      MINPODS  MAXPODS  REPLICAS  AGE
remindmeautoscaler  Deployment/remindmeweb-deployment  <unknown>/60%  2        6        2         1m
```

Figure 51: Output get HPA

Alternatively, if you want to delete the already created autoscalers, then the following command can be used.

Code Listing 67: Delete HPA

```
kubectl delete hpa <hpaname>
```

```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl delete hpa remindmeautoscaler
horizontalpodautoscaler.autoscaling "remindmeautoscaler" deleted
```

Figure 52: Output delete HPA

This will delete the scheduled autoscale with a defined name from the Kubernetes cluster.



Note: You can also prevent frequent pod creation and deletion caused by the thrashing by setting the `--horizontal-pod-autoscaler-downscale-delay` flag. Its default value is five minutes. This flag sets the metric cool-down time to prevent frequent fluctuations.

The HPA works great for scaling your cluster on different metrics, but it has a few shortcomings. Let's discuss one of those, as well.

Limitation

One of the limitations of HPA is that it doesn't support rolling updates with the replication controller. This is because a new replication controller is created during the rolling update, and HPA remains bound to the old replication controller. This can be overcome by using a deployment resource instead of a replication controller, which will ensure that both autoscaling and rolling updates are supported.

Summary

In this chapter, we learned how we could update our applications deployed in a Kubernetes cluster. We covered how we can use existing, established techniques like rolling updates and blue-green deployment to update our applications. We also covered different strategies for scheduling autoscaling in a Kubernetes cluster using `kubectl` and HPA.

In the next chapter, we'll discuss the last functional piece of our application—jobs.

Chapter 6 Jobs

In the previous chapter, we successfully deployed and updated our application to a Kubernetes cluster. As we are progressing, we have a requirement to send scheduled reminder mail to our customers. In modern applications, this is usually achieved with background tasks, as they don't require direct user interaction. This can be accomplished in Kubernetes using jobs and cron jobs. Jobs and cron jobs create one or more pods and ensure that they are successfully terminated.

Jobs track the successful completion of pods by tracking their exit and making sure that the defined number of pods are executed successfully. If any pod fails during the execution, the job makes sure that a new pod is created to complete the task and monitors it through to successful completion. This is very useful for background scenarios where there is no direct monitoring, and you can rely on the system to complete the task.

Job specification

Jobs also follow the same declarative style specification as other Kubernetes components. The following sample specification file, called Jobspec.yaml, creates a job for our **remindme** service. This job will print simulated email messages in the console and delete the reminders that are processed by the job using the **delete** API we created in Chapter 5.

Code Listing 68: Remindme job

```
apiVersion: batch/v1
kind: Job
metadata:
  name: remindmejob
spec:
  template:
    spec:
      containers:
        - name: remindmejob
          image: kubernetes Succinctly/remind-me-job:1.0.0
          env:
            - name: GET_HOSTS_FROM # This variable tells the service to
              find service host information from dns service. You can change this value
              to 'env' to query service hosts from environment variables.
              value: dns
            - name: apiUrl # This environment variable is used by the
              application to connect to the API.
              value: http://remindmeapi-statefulset-0.remind-me-api-
              svc.default.svc.cluster.local
          restartPolicy: Never
      backoffLimit: 4
```

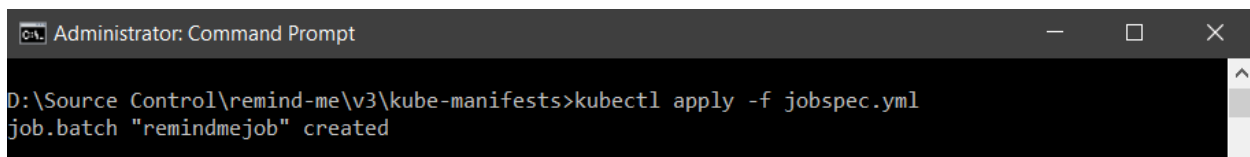
The specification asks Kubernetes to create a job named **remindmejob** using the given image. The **Job** specification requires **apiversion**, **kind**, and **metadata**.

The **spec.template** is exactly same as the pod template we used in Chapter 3. It defines the pod details and supports a **backofflimit** number, which is the same as a number of retries in case of failure. Once this number is reached, the job is moved to a failed state. The default value for this is **6**. The job can also be terminated by setting the **.spec.activeDeadlineSeconds** field, which ensures that the job completes within the specified time (or it fails).

You can apply the create job specification to your cluster using the familiar **kubectl** command.

Code Listing 69: Applying job to cluster

```
kubectl apply -f jobspec.yaml
```



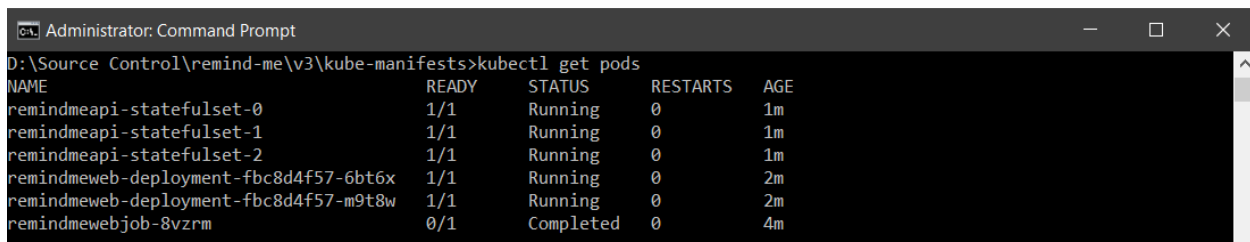
```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl apply -f jobspec.yaml
job.batch "remindmejob" created
```

Figure 53: Output create job

After a successful execution, you can execute the following command to check the status of the job provisioned in your cluster.

Code Listing 70: Get job details

```
kubectl get pods
```



```
Administrator: Command Prompt
D:\Source Control\remind-me\v3\kube-manifests>kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
remindmeapi-statefulset-0           1/1     Running   0           1m
remindmeapi-statefulset-1           1/1     Running   0           1m
remindmeapi-statefulset-2           1/1     Running   0           1m
remindmeweb-deployment-fbc8d4f57-6bt6x 1/1     Running   0           2m
remindmeweb-deployment-fbc8d4f57-m9t8w 1/1     Running   0           2m
remindmewebjob-8vzrm                0/1     Completed 0           4m
```

Figure 54: Output get pods

Job types

There are mainly three types of jobs in Kubernetes: nonparallel jobs, parallel jobs with a fixed completion count, and parallel jobs with a work queue. Let's discuss them in a little more detail.

Nonparallel jobs

These are single-execution jobs, and only one pod is created. If there is an error during the execution, then a new pod is created to complete the task. Otherwise, the job is considered complete once the pod terminates successfully.

Parallel jobs with a fixed completion count

These jobs require a `.spec.completions` value in the spec template. Multiple pods get created, not necessarily simultaneously, but equal to the value of `.spec.completions`. The job is considered complete when the same number of pods terminate successfully.

The default value of `.spec.completions` is 1.

Parallel jobs with a work queue

These jobs require a `.spec.parallelism` value in the spec template. Multiple pods get created simultaneously, equal to the value of `.spec.parallelism`, and the job is considered complete when the same amount of pods terminate successfully. The default value of `.spec.parallelism` is 1.



Note: *Once a single pod has exited, all other pods in the job must also be in the exiting phase.*

Jobs are great for one-time background processing, but for recurring tasks, we have the concept of cron jobs in Kubernetes.

Cron jobs

We now understand how we can process a task in the background in Kubernetes. However, in our case, we want to schedule a job to send emails periodically. Enter cron jobs. Cron jobs are like background jobs that are used to schedule low-priority tasks and background tasks, like sending emails or purging logs.

We will create a cron job for sending email periodically to users at midnight. A cron job is used to create jobs on a scheduled basis. It uses the popular **cron** format to specify schedules.

A **cron** expression is specified as a string of five numbers in the following order:

- Minute (0-59)
- Hour (0-23)
- Day of month (1-31)
- Month (1-12)
- Day of week (0-6)

We can also use a * or ? character to denote any value. Some examples of cron expressions are:

- 5 0 * 8 * - At 00:05 in August.
- 15 14 1 * * - At 14:15 on day-of-month 1.
- 0 0 * * 1-5 - At midnight on every day of the week, Monday–Friday.

Let's create our config file **cronjobspec.yaml** to define the cron job that will run every minute and send the reminder email. Notice the **kind** property.

Code Listing 71: CronJob

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: remindmejob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          restartPolicy: OnFailure
          containers:
            - name: remindmejob
              image: kubernetes Succinctly/remind-me-job:1.0.0
              env:
                - name: GET_HOSTS_FROM # This variable tells the service to
                  find service host information from dns service. You can change this value
                  to 'env' to query service hosts from environment variables.
                  value: dns
                - name: apiUrl # This environment variable is used by the
                  application to connect to the API.
                  value: http://remindmeapi-statefulset-0.remind-me-api-
                    svc.default.svc.cluster.local
          ---
```

You can apply the **cronjobspec.yaml** to the cluster using the following command.

Code Listing 72: Apply cron job to cluster

```
kubectl create -f cronjobspec.yaml
```

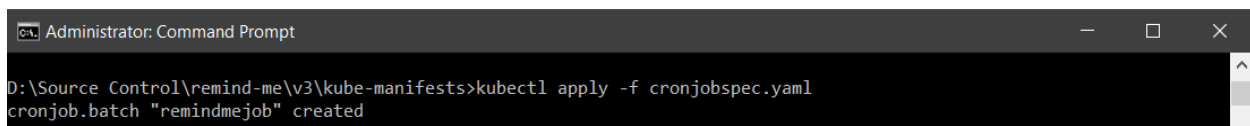


Figure 55: Output cron job created

We can also use the imperative style to achieve the same.

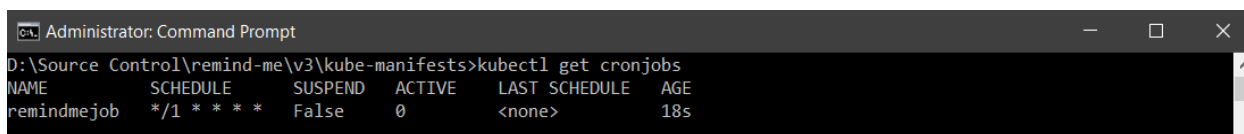
Code Listing 73: Declare and apply cron job to cluster

```
kubectl run remindmejob --schedule="*/1 * * * *" --restart=OnFailure --
image=kubernetessuccinctly/reminde-me-job:1.0.0
```

After it is successfully scheduled, we can check the status of the job using the following command.

Code Listing 74: Check job status

```
kubectl get cronjobs
```



NAME	SCHEDULE	SUSPEND	ACTIVE	LAST SCHEDULE	AGE
remindmejob	*/1 * * * *	False	0	<none>	18s

Figure 56: Output job status

Similarly, if we want to delete the job, we can use the following command:

Code Listing 75: Delete job

```
kubectl delete cronjob remindmejob
```

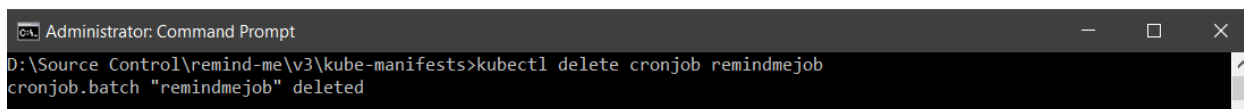


Figure 57: Output delete cron job

The previous command will delete the **reminderwebjob** from our cluster.



Note: The *startingDeadlineSeconds* field is optional. It stands for the deadline, in seconds, for starting the job if it misses its scheduled time for any reason. After the deadline, the cron job does not start the job. Jobs that do not meet their deadline in this way count as failed jobs. If this field is not specified, the jobs have no deadline. The cronjob controller counts how many missed schedules happen for a cron job ([source](#)).

The cron job is a powerful feature that is heavily utilized for background processing and ensures your application can handle the increased load.

Summary

In this chapter, we learned about creating and deploying background jobs to the Kubernetes cluster. We also learned about scheduling the jobs as cron jobs for the cluster.

In the next chapter, we will discuss how you can gather infrastructure and application logs when you are running a cluster at scale. We will also study some of the available solutions for application and cluster monitoring.

Chapter 7 Monitoring and Logging

In the previous chapter, we covered how we can schedule background jobs in Kubernetes. Once the deployment is complete, the next task that comes to mind is monitoring. How will we ensure that our cluster is working smoothly? What happens in case of an error? We'll answer these questions in this chapter.

Monitoring consists of many subtasks that need to be performed to have an effective monitoring solution. In the case of Kubernetes, our task becomes more difficult because there are so many components that can go wrong. We need to make sure that our master node, worker node, and pod/containers all are working correctly. We also must keep an eye on resource utilization by a different component. There are many available solutions that help us achieve this task, such as:

- Kubernetes Dashboard.
- Monitoring through Prometheus.
- Centralized logging through Elasticsearch, Fluentd, and Kibana (EFK) stack.

Let's look at each of them in detail.

Kubernetes Dashboard

Kubernetes Dashboard is our primary web interface for managing Kubernetes clusters. You can manage cluster components including applications deployed in the cluster through Kubernetes Dashboard. Most common cluster metrics can be viewed, and further diagnosis or troubleshooting can be done using the Dashboard UI.

Installation

To deploy the dashboard, execute the following command.

Code Listing 76: Install dashboard

```
kubectl apply -f
https://raw.githubusercontent.com/kubernetes/dashboard/master/src/deploy/re
commended/kubernetes-dashboard.yaml
```

If you are seeking access from the same machine the dashboard is installed on, you need to first execute the following command.

Code Listing 77: kubectl proxy

```
kubectl proxy
```


Now we can access the dashboard at: <http://localhost:8001/api/v1/namespaces/kube-system/services/https:kubernetes-dashboard:/proxy/>.

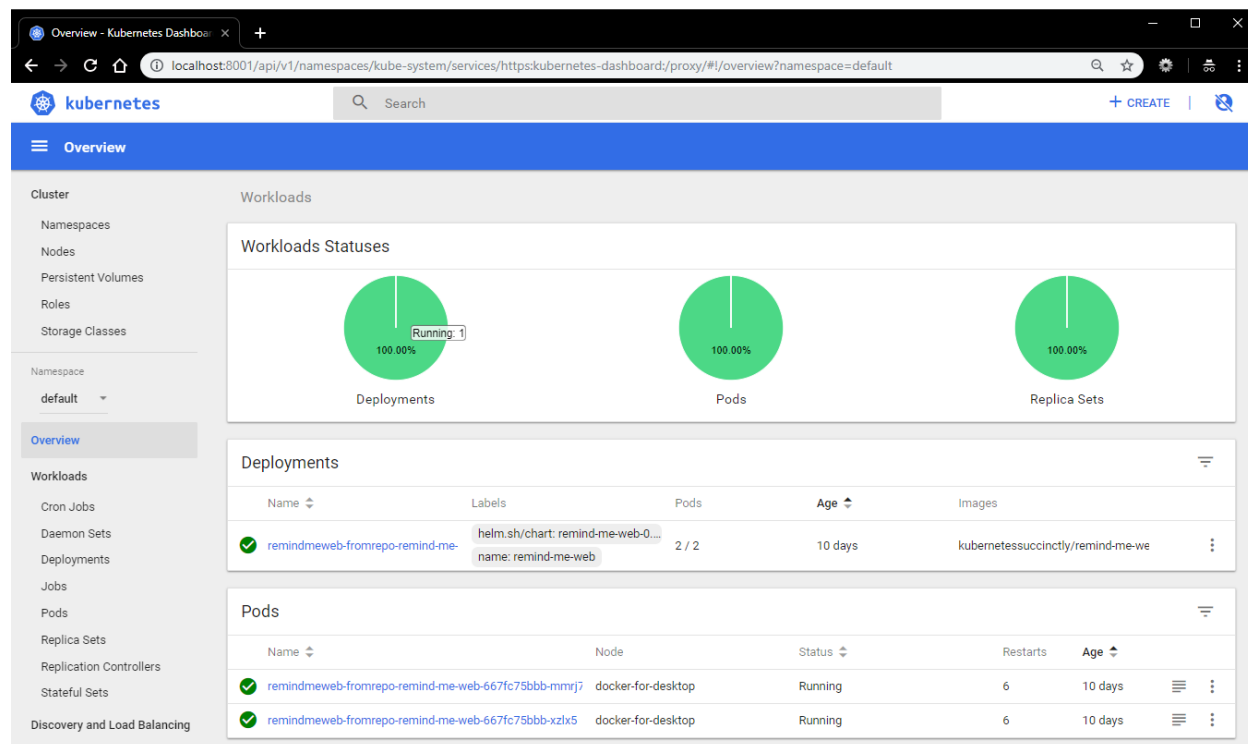


Figure 58: Kubernetes Dashboard

Monitoring through Prometheus

Prometheus is an open-source monitoring and alerting tool that was first built by SoundCloud. It is well integrated with other open-source projects like Grafana and Kubernetes. It follows a pull-based approach for metrics collection. In a pull-based system, the monitoring system fetches metrics from the application or services.

Although the Prometheus stack consists of many components, the Prometheus server is the main component for scraping and storing time-series data.

Installation

[Installing Prometheus](#) is straightforward for a Kubernetes cluster. It is always good practice to segregate your deployed components into namespaces, as you do for your code bases. Let's create a new namespace called **monitoring** for our Prometheus installation.

Code Listing 78: Create namespace

```
kubectl create namespace monitoring
```

Now we need to assign read permission to this namespace for our cluster so that the Prometheus server can access Kubernetes APIs for fetching metrics data. We will create a cluster-role for this purpose using the following configuration.

Code Listing 79: Cluster role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: prometheus
rules:
- apiGroups: [""]
  resources:
    - nodes
    - nodes/proxy
    - services
    - endpoints
    - pods
  verbs: ["get", "list", "watch"]
- apiGroups:
  - extensions
  resources:
    - ingresses
  verbs: ["get", "list", "watch"]
- nonResourceURLs: ["/metrics"]
  verbs: ["get"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: prometheus
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: prometheus
subjects:
- kind: ServiceAccount
  name: default
  namespace: monitoring
---
```

Let's save this configuration in a file and name it **cluster-permission.yaml**. Apply this configuration to our cluster using the following command.

Code Listing 80: Apply permission

```
kubectl create -f cluster-permission.yaml
```

To separate our nonsensitive configuration from pods and components, we can create a **ConfigMaps**, which will have all the configuration information, such as environment variables, command line arguments, port numbers, and other configurations of pods and containers. It will have all the configuration required by the Prometheus server to discover pods and services. It will also have all the information regarding metrics collection and Kubernetes certificates.



Note: ConfigMaps should only be used for nonsensitive configuration, while for the sensitive configuration, you should use secrets.

You can use the sample config map specification from the *Kubernetes Succinctly* [GitHub location](#). Apply the configuration to our cluster using the following command.

Code Listing 81: Apply Prometheus configuration

```
kubectl create -f https://raw.githubusercontent.com/Kubernetes-Succinctly/Manifest/master/prometheus-config.yaml -n monitoring
```

Finally, we will create a Prometheus deployment by mounting the previous configuration in **/etc/Prometheus**. Let's name this specification file **prometheus.yaml**.

Code Listing 82: Prometheus Deployment

```
apiVersion: extensions/v1
kind: Deployment
metadata:
  name: prometheus-deployment
  namespace: monitoring
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: prometheus-server
    spec:
      containers:
        - name: prometheus
          image: prom/prometheus:v2.1.0
          args:
            - "--config.file=/etc/prometheus/prometheus.yml"
            - "--storage.tsdb.path=/prometheus/"
          ports:
            - containerPort: 9090
```

```

    volumeMounts:
      - name: prometheus-config-volume
        mountPath: /etc/prometheus/
      - name: prometheus-storage-volume
        mountPath: /prometheus/
    volumes:
      - name: prometheus-config-volume
        configMap:
          defaultMode: 420
          name: prometheus-server-conf

      - name: prometheus-storage-volume
        emptyDir: {}
---

```

You can apply the YAML using the following command.

Code Listing 83: Create Prometheus deployment

```
kubectl create -f prometheus.yaml -n monitoring
```

If you can encounter any errors in the setup, you can refer to the [Prometheus documentation](#) for more details. If everything succeeds, we have a working version of Prometheus deployed in our cluster.

Connecting Prometheus

The easiest way to connect to Prometheus is by exposing it as a service with **nodePort**. Let's create the file *prometheus-service.yaml* with the following configuration and expose Prometheus on Kubernetes node ports on port 31112.

Code Listing 84: Exposing service

```

apiVersion: v1
kind: Service
metadata:
  name: prometheus-service
spec:
  selector:
    app: prometheus-server
  type: NodePort
  ports:
    - port: 8080
      targetPort: 9090
      nodePort: 31112

```

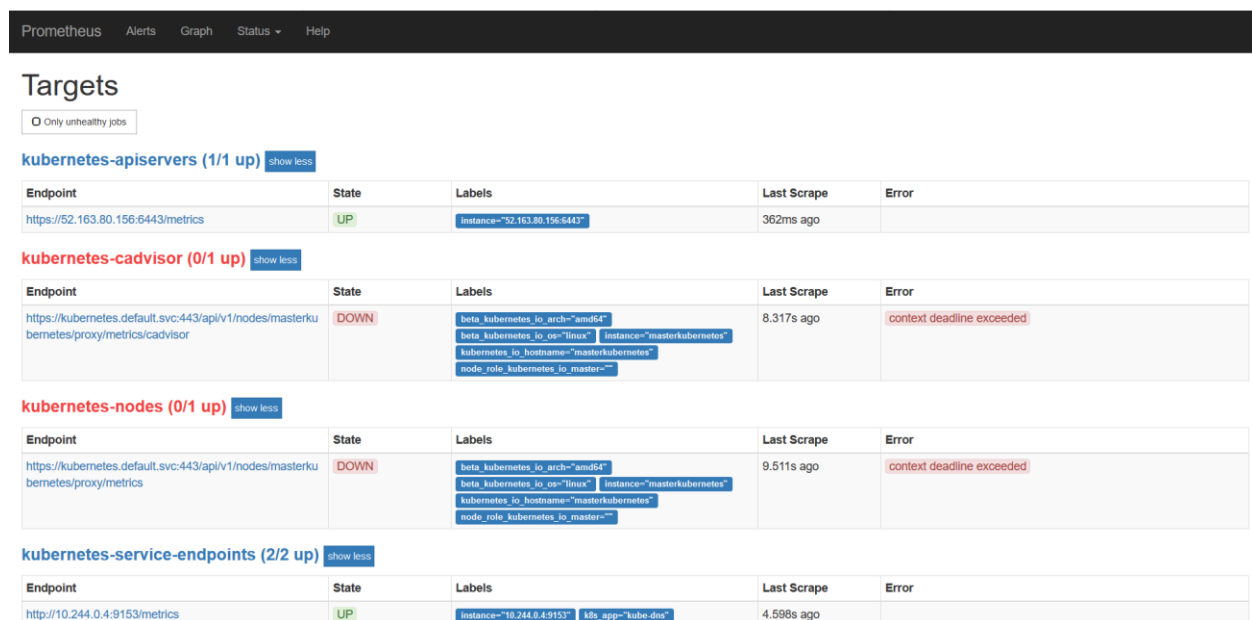
Create the service using the following command.

Code Listing 85: Expose Prometheus as a service

```
kubectl create -f prometheus-service.yaml -n monitoring
```

If successful, we can access Prometheus at the following URL: <http://<node-ip>:31112/>.

On the page available at the link, navigate to **Targets** under the **Status** tab, and you will see the status of the different components within your cluster. In the following screenshot, we can see different types of status, depending on component health.



The screenshot shows the Prometheus web interface with the 'Targets' tab selected. The interface includes a navigation bar with 'Prometheus', 'Alerts', 'Graph', 'Status', and 'Help'. Below the navigation bar, there's a 'Targets' section with a filter 'Only unhealthy jobs'. The main content area displays four target groups: 'kubernetes-apiservers (1/1 up)', 'kubernetes-cadvisor (0/1 up)', 'kubernetes-nodes (0/1 up)', and 'kubernetes-service-endpoints (2/2 up)'. Each group has a table with columns for 'Endpoint', 'State', 'Labels', 'Last Scrape', and 'Error'.

Endpoint	State	Labels	Last Scrape	Error
https://52.163.80.156:6443/metrics	UP	instance="52.163.80.156:6443"	362ms ago	
https://kubernetes.default.svc:443/api/v1/nodes/masterkubernetes/proxy/metrics/cadvisor	DOWN	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="masterkubernetes" kubernetes_io_hostname="masterkubernetes" node_role_kubernetes_io="master"	8.317s ago	context deadline exceeded
https://kubernetes.default.svc:443/api/v1/nodes/masterkubernetes/proxy/metrics	DOWN	beta_kubernetes_io_arch="amd64" beta_kubernetes_io_os="linux" instance="masterkubernetes" kubernetes_io_hostname="masterkubernetes" node_role_kubernetes_io="master"	9.511s ago	context deadline exceeded
http://10.244.0.4:9153/metrics	UP	instance="10.244.0.4:9153" k8s_app="kubernetes"	4.598s ago	

Figure 59: Prometheus Targets tab

Centralized logging

By default, you have **kubectl** logs, which are written to **stdout**. We can fetch these logs using the following command.

Code Listing 86: Get pod logs

```
kubectl logs <podname>
```

However, since Kubernetes is a distributed system, this will not be enough for a production environment in which multiple pods are running the same or different applications. Moreover, the logs are not persisted anywhere and can easily be lost in the case of an error, which defeats the purpose. We will need central persistent storage to store and analyze the logs. Also, monitoring statistics will need to be persisted outside of the container for future analysis.



Tip: You can use `kubectl logs --help` to get other commands for getting different types of logs, like container logs. The command will only search pods in the default namespace. If you want to search in other namespaces, you will have to specify the namespace name using the `-n <namespace>` flag.

Centralized logging is a critical piece in a distributed application. With Kubernetes, you also have more complexity regarding the node, pod, and cluster, which makes centralized logging even more essential. There is no Docker-like driver support in Kubernetes for central logging, but there are open-source solutions available that can help you achieve it. The most popular of them is the EFK stack, which stands for Elasticsearch, Fluentd, and Kibana.

Elasticsearch

Elasticsearch is a distributed, RESTful search and analytics engine that helps you store and query data in a fast and reliable manner. It is schema-less, but you can define your index.

Elasticsearch can also be used as a managed service. There are many cloud providers that support it, like AWS, Azure, and GCP. In our scenario, we are going to deploy it as a separate service within the cluster. In complex systems, the Elasticsearch itself might be deployed in a separate Kubernetes cluster due to its vast use cases. If you have a search capability within your cluster, you most likely already have or need this component setup in your cluster. It supports schema-less indexed storage of your data, which can be easily queried via REST or natively supported APIs.

Kibana

Kibana is a visualization tool that can be integrated with Elasticsearch to query stored data and visualize it for different use cases, including application monitoring. It supports visualization in chart, graph, and table format. Once it's set up, it can be easily integrated with Elasticsearch, and allows for data to be accessed in the simple, browser-based interface. It should be configured with the same Elasticsearch version; otherwise, it won't work.



Note: Make sure your versions of Elasticsearch and Kibana are compatible.

Kibana provides a quick and easy way to explore your Elasticsearch data, which is very useful with large data sets, such as logs. A Kibana dashboard looks like the following screenshot.

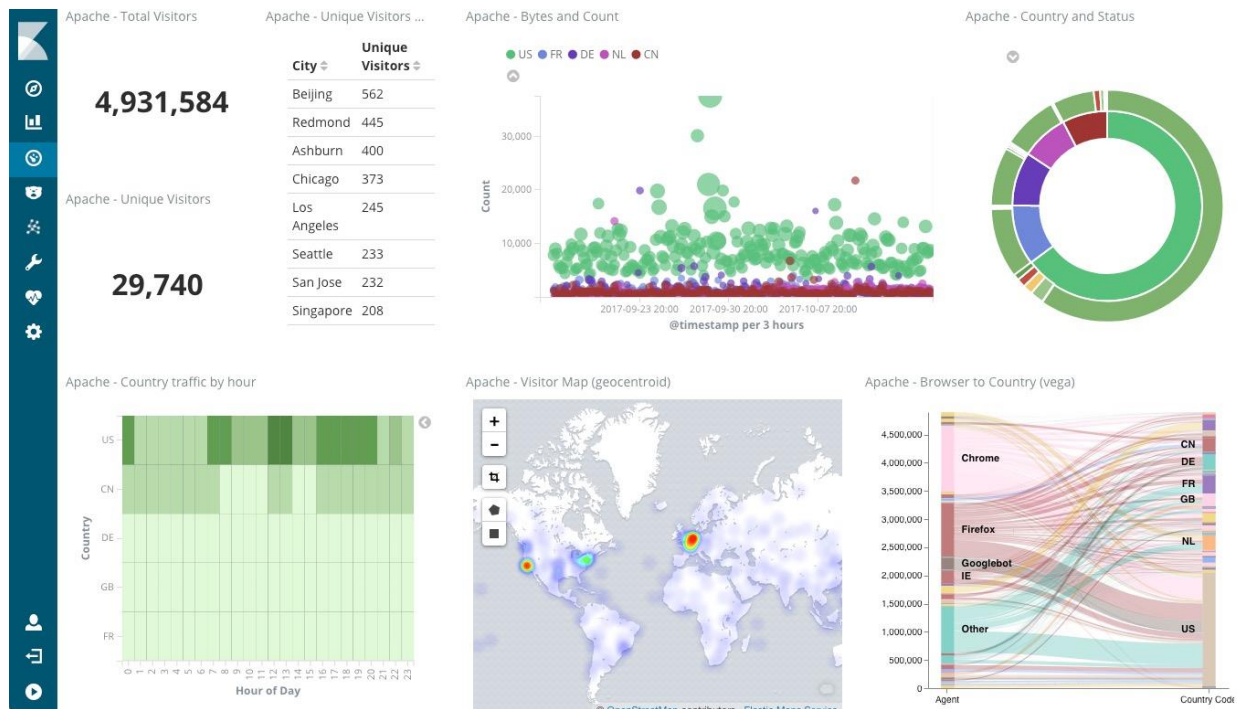


Figure 60: Kibana dashboard ([Source](#))

Fluentd

[Fluentd](#) is a unified data aggregator that is used for collecting data from different sources, in this case, logs from pods, nodes, etc., and sending them to Elasticsearch. Fluentd supports collecting data from different data sources and produces a common JSON output, which enables the downstream layer to process it easily without modifying the source systems. As it is open source and supports pluggable architecture, it has lots of pluggable components available to extend its functionality and use with a wide variety of systems.

Fluentd supports both in-memory and file-based buffering, allowing it to provide both desired scalability and performance. In our scenario, Fluentd will run on each node, aggregating logs from all components and sending them to Elasticsearch for storage and analysis.

Installation

We can deploy EFK as an add-on to our cluster using configurations provided by the official [Kubernetes repository](#).



Note: The EFK add-on should not be used as-is in production. It should be configured with proper security and storage configurations to have a reliable and secure setup. Please refer to the [Kubernetes repository](#) for more details.

This will install Elasticsearch and Kibana as services, while **fluentd** will be installed as a daemon set on each node. Application logs will be collected by **fluentd**, which will send these

logs to Elasticsearch. You can easily access these logs using Kibana. Alternatively, since Elasticsearch is an independent service, you can use any other tool to fetch and analyze logs. The following diagram explains the setup in more detail.

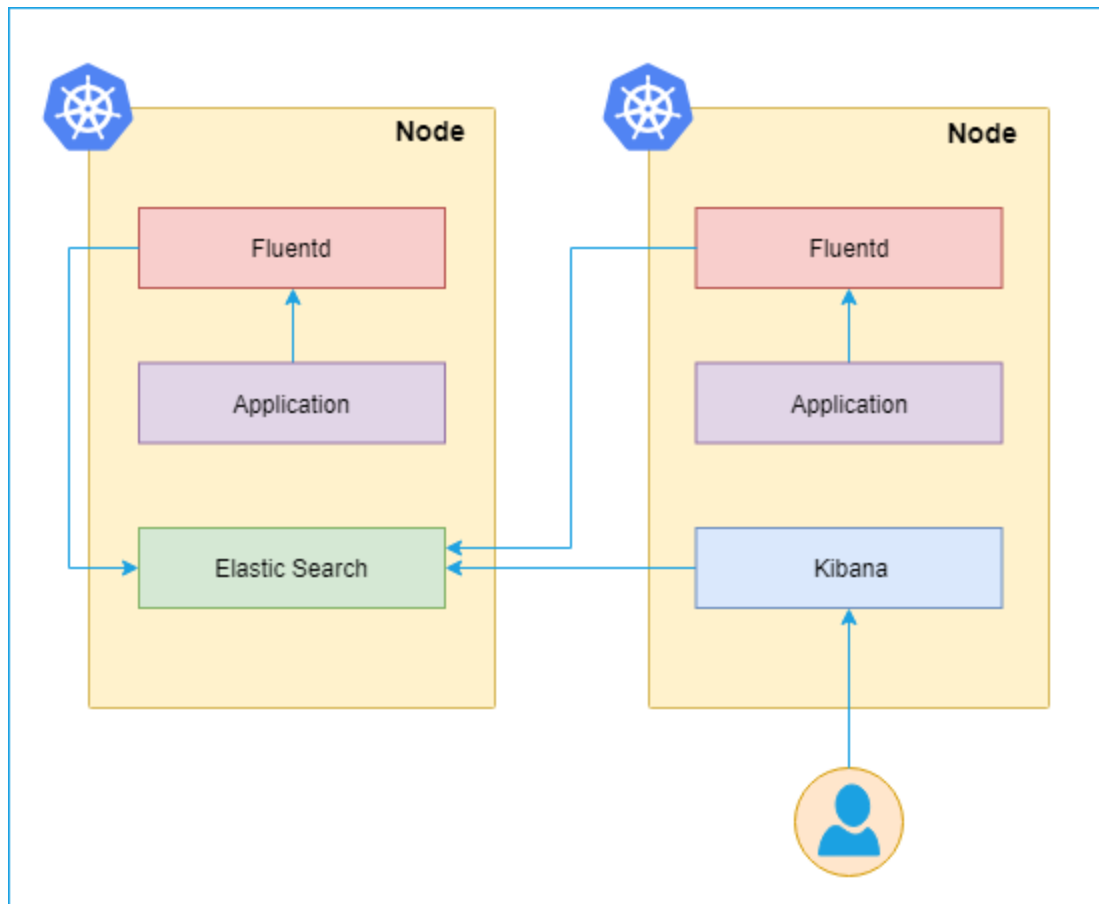


Figure 61: EFK Stack Deployment

The diagram shows a two-node cluster where **fluentd** is installed as a daemon set in each node. Elasticsearch and Kibana can be installed on the same node or separate nodes. Once the entire stack is installed, we need to configure Kibana and add an index, which needs to fetch from Elasticsearch. The index, which you configure in Elasticsearch, needs to be added here so that you can access it from Kibana.

Summary

In this chapter, we have covered monitoring Kubernetes clusters using Prometheus. We have seen how we can leverage the EFK stack to have central logging implemented in our Kubernetes cluster. Now let's look at how we can package the application using Helm to efficiently manage production deployments.

Chapter 8 Packaging Applications with Helm

After deploying applications again and again in the course of this book, you might have realized that writing YAML specs for each component—such as deployments, replica sets, and services that make up the cluster—is difficult. If you are working on more extensive applications, this problem becomes even harder to solve. Enter Helm, a package manager for Kubernetes that makes it easy for developers to package, configure, and deploy applications on Kubernetes clusters.

Introduction

You probably interact with package managers in your day-to-day life, such as Python's *pip*, Debian's *apt*, and Microsoft's *NuGet*. Just like the other package managers, Helm can perform the following activities for you:

- Fetch software packages from repositories.
- Configure software deployments.
- Install software and its dependencies.
- Upgrade software packages to the version that you need.

Helm is made up of two components: a command line interface (CLI), **helm**, and a server component, **tiller**. As a developer, you will interact with the **helm** CLI, while the **tiller** component will run on the Kubernetes cluster, and will manage the deployment of your software packages.

The software packages in the world of Helm are called *charts*, and you can find several well-known charts at the [official Helm chart repository](#). Let's investigate the components that make up a chart.

The anatomy of a chart

A typical chart is composed of several YAML configuration files. Here's how a typical chart looks.

Code Listing 87: Chart Files

```
package-name/  
charts/  
templates/  
Chart.yaml  
LICENSE
```

```
README.md
requirements.yaml
values.yaml
```

Helm recognizes only the files and directories shown in the previous code listing, and it will leave the other files in the directory unprocessed. Therefore, the names of the files and directories are important. The following are the various components that make up a chart:

- **values.yaml:** This file contains default configuration values for the chart.
- **requirements.yaml:** This file contains the list of dependencies of the chart.
- **charts/:** This directory contains managed chart dependencies that you don't want to link to, or can't link to, from an internet-accessible source. It is recommended to use the *requirements.yaml* configuration for dependencies if you can.
- **templates/:** This directory contains template files that are referenced by the *values.yaml* file to define the configuration. Since programming language Go has been used to build Helm, *values.yaml* uses the [Go template format](#).
- **Chart.yaml:** This file contains the necessary metadata about the chart, such as the name, version, website, and maintainer details. You can define metadata for your chart such as Kubernetes version, Helm version, and other constraints in this file, which will be useful to the users of this chart.
- **LICENSE:** This file contains the chart licensing information.
- **README.md:** This file contains information necessary for the users to use the chart.

Using the `helm` command, you can install a chart from your local directory or from a chart repository. You can also supply a compressed version of the chart directory in the form of a `tar.gz` (or `.tgz`) file to the `helm` command, which is the format also used by the helm repository for sending charts to the client.

By default, Helm installs charts from the default [chart repository](#). A chart repository lists the location of charts and their related metadata. This repository is called **stable** and you can find it on [GitHub](#). There is another repository called **incubator** located in the same repository that contains all the charts that are not yet ready for promotion to stable. In addition to these repositories, you can also find many charts at the [Bitnami Helm charts repository](#) that are not present in the official repo.

Installation

To install Helm and Tiller on your developer machine, follow the instructions from the [official documentation](#). The easiest way to install Helm is to execute the following command via Homebrew on your macOS.

Code Listing 88: Install Helm on macOS

```
brew install kubernetes-helm
```

Likewise, on your Windows machine, you can install it through [Chocolatey](#) using the following command.

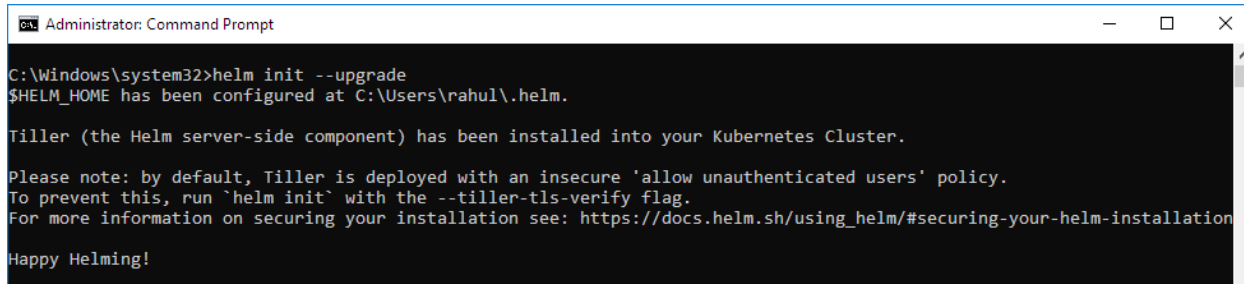
Code Listing 89: Install Helm on Windows

```
choco install kubernetes-helm
```

After installing Helm, you can execute the following command from your terminal to install Tiller.

Code Listing 90: Install tiller

```
helm init --upgrade
```



```
Administrator: Command Prompt
C:\Windows\system32>helm init --upgrade
$HELM_HOME has been configured at C:\Users\rahul\.helm.

Tiller (the Helm server-side component) has been installed into your Kubernetes Cluster.

Please note: by default, Tiller is deployed with an insecure 'allow unauthenticated users' policy.
To prevent this, run `helm init` with the --tiller-tls-verify flag.
For more information on securing your installation see: https://docs.helm.sh/using_helm/#securing-your-helm-installation

Happy Helming!
```

Figure 62: Output Tiller install

This command applies some configurations on your local system and deploys Tiller on your local Kubernetes cluster in the **kube-system** namespace. You can inspect the deployment by executing the following command in your terminal.

Code Listing 91: Describe deployment tiller

```
kubectl describe deployment tiller-deploy -n kube-system
```



Note: Helm will become a single-service component from version 3, and there will be no Tiller component as part of Helm installation.

You can check the version of Helm installed on your system by running the **version** command.

Code Listing 92: Check version

```
helm version
```

This command will output the versions of both the Helm client and the Tiller running on the cluster.

Creating charts

To understand Helm in detail, let's create a Helm chart for our application. Let's create a chart for the v1 **remind-me-web** application that we created earlier. Execute the following command to create a chart named **remind-me-web**.

Code Listing 93: Create chart

```
helm create remind-me-web
```

This command will create a new folder named `remind-me-web` with the following directory structure. Note that we have deliberately removed some of the unnecessary files from the generated chart template.

Code Listing 94: Chart Template

```
remind-me-web
|
| .helmignore
| Chart.yaml
| values.yaml
|
+---charts
\---templates
    deployment.yaml
    NOTES.txt
    service.yaml
    _helpers.tpl
```

Let's talk briefly about the files present in the template folder:

- **deployment.yaml:** This file contains the YAML definition for your deployment object.
- **service.yaml:** This file contains the YAML definition for your service object.
- **_helpers.tpl:** This is a template that consists of partials and helper functions that can be used in the **deployment** and **service** definitions.
- **NOTES.txt:** The contents of this file are printed on the console after the chart is successfully deployed. You should write the next steps in this file that the user needs to take after deploying your chart.

The default chart template is configured to deploy **nginx** on a cluster. For this exercise, we will modify the default chart to deploy our service. Remember that we created a service and deployment for provisioning the [first version for our application](#).

Let's start by defining the service in our chart. The most important part of the chart is the contents in the `templates/` directory. All Kubernetes objects, such as deployments and services, are defined in the manifests present in this directory. The easiest way to translate a manifest to the chart is to dump the manifest in this directory and use the **helm install** command to deploy the chart on a cluster.

Helm takes all files present in this directory through a [Go Template](#) rendering engine to substitute parameters present in templates with actual values to generate a valid definition. Open the file **service.yaml** present in the **templates** folder to define our service, and replace the code with the following code listing.

Code Listing 95: Define service

```
apiVersion: v1
kind: Service
metadata:
  name: {{ include "remind-me-web.fullname" . }} # remind-me-web.fullname
  is defined in /templates/_helpers.tpl
  labels:
    name: {{ include "remind-me-web.name" . }}
    helm.sh/chart: {{ include "remind-me-web.chart" . }}
spec:
  type: NodePort
  ports:
    - port: {{ .Values.service.port }} # Select port from parameter
      values defined in Values.yaml
      targetPort: {{ .Values.service.targetPort }}
      protocol: TCP
      nodePort: {{ .Values.service.nodePort }}
  selector:
    name: {{ include "remind-me-web.name" . }}
---
```

Helm defines three objects that you can use within your templates to provide configurable values for your chart attributes: **.Release**, **.Chart**, and **.Value**. The **.Release** object gets values from command line. The **.Chart** object is used to provide information about the chart, such as the name and version. You can set values for this object in the Chart.yaml file. The **.Value** object exposes custom configurations for a chart. The values for this object are stored in the Values.yaml file.

Now, let's move on to define the configuration for our deployment. Open the **/templates/deployment.yaml** file in your editor and change the code in the file to the code in the following listing.

Code Listing 96: Define configuration

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: {{ include "remind-me-web.fullname" . }}
  labels:
    name: {{ include "remind-me-web.name" . }}
    helm.sh/chart: {{ include "remind-me-web.chart" . }}
spec:
  replicas: {{ .Values.replicaCount }} # substitute value of replica
  count from the value in Values.yaml file.
  selector:
    matchLabels:
      name: {{ include "remind-me-web.name" . }}
  template:
```

```

metadata:
  labels:
    name: {{ include "remind-me-web.name" . }}
spec:
  containers:
    - name: {{ .Chart.Name }} # the substitution value of this
parameter is present in Chart.yaml file.
      image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
      ports:
        - name: http
          containerPort: 80
          protocol: TCP
---

```

To keep the chart configurable by the end user, we have used several parameters in our definitions. Let's define the default values of these parameters in the Values.yaml file. Modify the code in the file to reflect the following code listing.

Code Listing 97: Define default values

```

replicaCount: 2

nameOverride: ""
fullnameOverride: ""

service:
  port: 80
  targetPort: 80
  nodePort: 31120

image:
  repository: "kubernetes Succinctly/remind-me-web"
  tag: "1.0.0"

```

As a good practice, let's now modify the code in the Chart.yaml file to add metadata for our chart to help the users. Modify the code to reflect the following code listing.

Code Listing 98: Metadata

```

apiVersion: v1
appVersion: "1.0"
description: Helm chart for remind-me-web v1
name: remind-me-web
version: 0.1.0

```

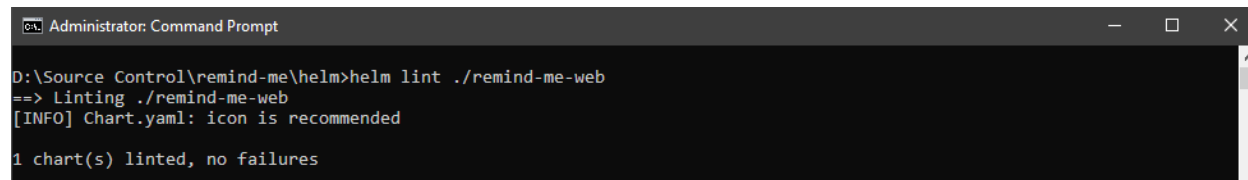
Our chart is now ready for deployment to our cluster. In the next section, we will deploy our chart to our cluster and perform some common operations on it.

Deploying charts

Now that our chart is ready, let's validate whether our chart is appropriately formed by processing it through a linter. Execute the following command in your terminal to validate your chart.

Code Listing 99: Validate chart

```
helm lint ./remind-me-web
```



```
Administrator: Command Prompt
D:\Source Control\remind-me\helm>helm lint ./remind-me-web
==> Linting ./remind-me-web
[INFO] Chart.yaml: icon is recommended
1 chart(s) linted, no failures
```

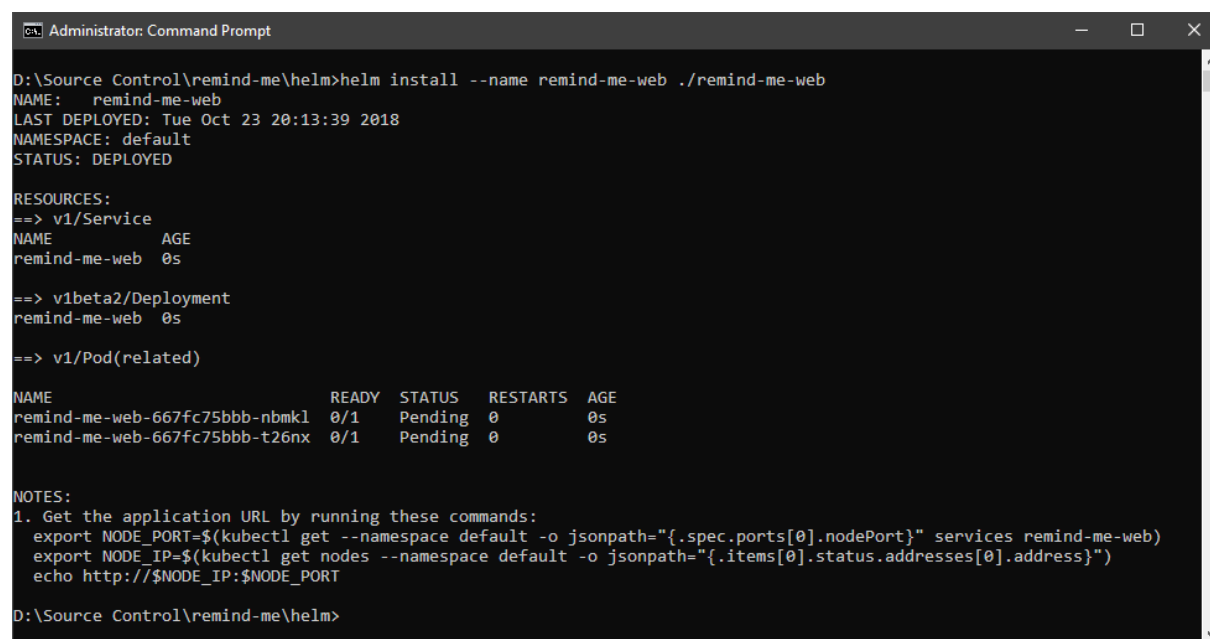
Figure 63: Output validate chart

Now that we have a green signal from the linter, let's move on to deploy this chart on the local cluster with this command.

Code Listing 100: Deploy chart

```
helm install --name remind-me-web ./remind-me-web
```

In this command, you can see that we have provided the value of the **.Release.Name** parameter in the command. We have used this parameter in the `/templates/_help.tpl` template and the `NOTES.txt` file.



```
Administrator: Command Prompt
D:\Source Control\remind-me\helm>helm install --name remind-me-web ./remind-me-web
NAME: remind-me-web
LAST DEPLOYED: Tue Oct 23 20:13:39 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME          AGE
remind-me-web 0s

==> v1beta2/Deployment
remind-me-web 0s

==> v1/Pod(related)

NAME                                READY  STATUS   RESTARTS  AGE
remind-me-web-667fc75bbb-nbmkl      0/1    Pending  0          0s
remind-me-web-667fc75bbb-t26nx      0/1    Pending  0          0s

NOTES:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services remind-me-web)
  export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT

D:\Source Control\remind-me\helm>
```

Figure 64: Installing chart

Your service should not be available on <http://localhost:31120>. Congratulations! You have successfully deployed your first chart on a live cluster.

You can modify values in your **Deployment** template or the **Values** template and reinstall the chart to make any modifications to your deployment.

Debugging charts

Writing charts can be tricky because the templates are rendered on Tiller, which in turn sends the generated templates to the Kubernetes API server for processing. If you want to check the generated value of an expression, then comment out the expression that you want to test and execute the following command.

Code Listing 101: Check chart expression

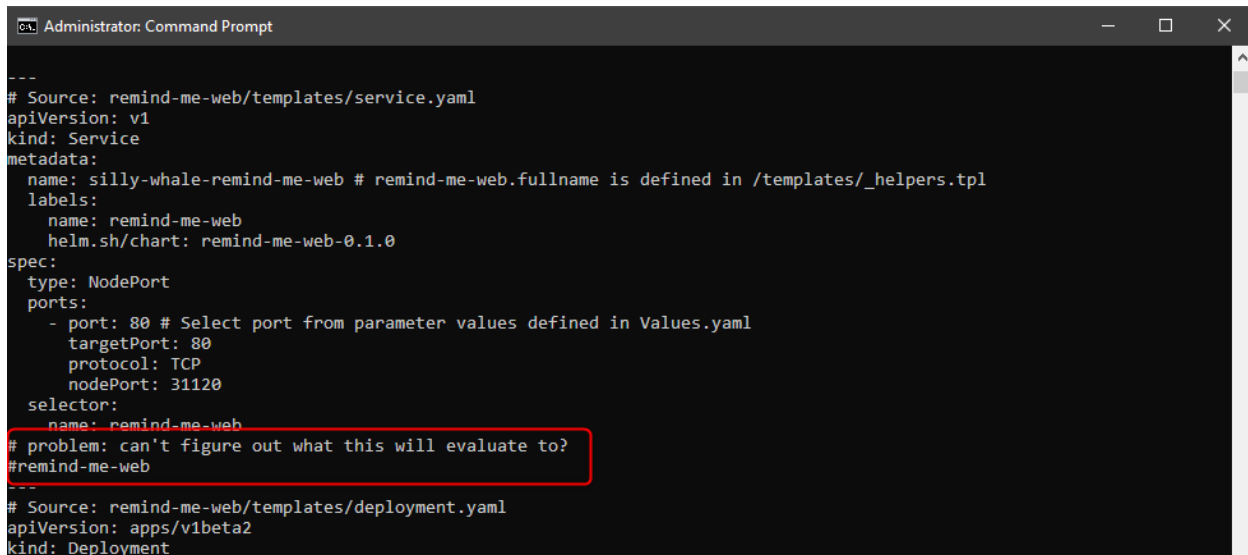
```
helm install ./remind-me-web --dry-run --debug
```

This command will return the manifest with the comment intact. For example, if we want to evaluate the value of the following complex expression in the helper's template, we can copy the relevant section, paste it in either the **deployment** or **service** template, and execute the previous command.

Code Listing 102: Update template

```
apiVersion: v1
kind: Service
... removed content for brevity
# problem: can't figure out what this will evaluate to?
# {{- default .Chart.Name .Values.nameOverride | trunc 63 | trimSuffix "-"
" -}}
```

After executing the **helm install** command with the **dry-run** and **debug** flags, you will be able to see the evaluated value of the expression in comments.



```
Administrator: Command Prompt

---
# Source: remind-me-web/templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: silly-whale-remind-me-web # remind-me-web.fullname is defined in /templates/_helpers.tpl
  labels:
    name: remind-me-web
    helm.sh/chart: remind-me-web-0.1.0
spec:
  type: NodePort
  ports:
    - port: 80 # Select port from parameter values defined in Values.yaml
      targetPort: 80
      protocol: TCP
      nodePort: 31120
  selector:
    name: remind-me-web
# problem: can't figure out what this will evaluate to?
#remind-me-web
---
# Source: remind-me-web/templates/deployment.yaml
apiVersion: apps/v1beta2
kind: Deployment
```

Figure 65: Output Helm install debug

Let's now prepare our chart for broader distribution by packaging it.

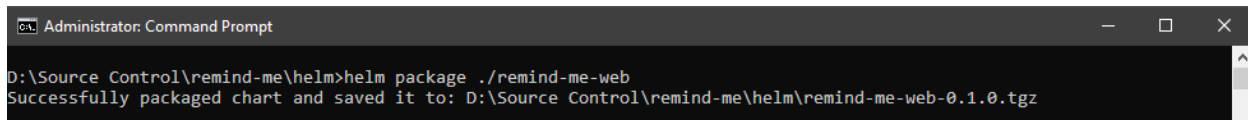
Packaging charts

To make our charts available to our friends, colleagues, and family, we need to package it in the form of a TAR file. The following command will package your chart and generate a zipped TAR file in the current working directory.

Code Listing 103: Package chart

```
helm package ./remind-me-web
```

The output of this command will tell you the path where the package was generated.



```
Administrator: Command Prompt

D:\Source Control\remind-me\helm>helm package ./remind-me-web
Successfully packaged chart and saved it to: D:\Source Control\remind-me\helm\remind-me-web-0.1.0.tgz
```

Figure 66: Output package chart

Helm supports discovering and installing charts from HTTP server. It reads the list of available charts from an index file hosted on the server and downloads charts from the location mentioned in the index file for a chart. You can also use Helm command **helm serve** to serve charts from your local system.

To create the index file for your charts, execute the following command in your terminal at the same working directory where your charts are located.

Code Listing 104: Create Index

```
helm repo index .
```

We copied the index file and the **remind-me-web** chart to a [GitHub repository](https://raw.githubusercontent.com/Kubernetes-Succinctly/helmrepo/master/) that we can now use as a HTTP server for our charts. Execute the following command in your terminal to add the Helm repository to your system.

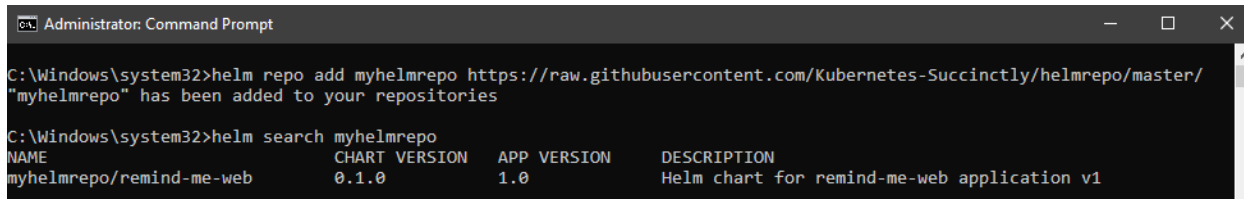
Code Listing 105: Add repository

```
helm repo add myhelmrepo https://raw.githubusercontent.com/Kubernetes-Succinctly/helmrepo/master/
```

After adding the **repo**, you can search for charts available in the repository by executing the following command.

Code Listing 106: Search chart

```
helm search myhelmrepo
```



```
C:\Windows\system32>helm repo add myhelmrepo https://raw.githubusercontent.com/Kubernetes-Succinctly/helmrepo/master/
"myhelmrepo" has been added to your repositories

C:\Windows\system32>helm search myhelmrepo
NAME                CHART VERSION  APP VERSION     DESCRIPTION
myhelmrepo/remind-me-web  0.1.0          1.0             Helm chart for remind-me-web application v1
```

Figure 67: Output search repository

Let's now install the chart that we just added from the repository rather than the local system. Clear the existing deployment from your system and execute the following command in your terminal.

Code Listing 107: Installing chart

```
helm install --name remindmeweb-fromrepo myhelmrepo/remind-me-web
```

```
Administrator: Command Prompt

C:\Windows\system32>helm install --name remindmeweb-fromrepo myhelmrepo/remind-me-web
NAME: remindmeweb-fromrepo
LAST DEPLOYED: Wed Oct 24 22:14:44 2018
NAMESPACE: default
STATUS: DEPLOYED

RESOURCES:
==> v1/Service
NAME                                AGE
remindmeweb-fromrepo-remind-me-web  1s

==> v1beta2/Deployment
remindmeweb-fromrepo-remind-me-web  1s

==> v1/Pod(related)

NAME                                READY  STATUS             RESTARTS  AGE
remindmeweb-fromrepo-remind-me-web-667fc75bbb-mm7j7  0/1    ContainerCreating  0          0s
remindmeweb-fromrepo-remind-me-web-667fc75bbb-xz1x5  0/1    Pending            0          1s

NOTES:
1. Get the application URL by running these commands:
  export NODE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].nodePort}" services remindmeweb-fromrepo-remind-me-web)
  export NODE_IP=$(kubectl get nodes --namespace default -o jsonpath="{.items[0].status.addresses[0].address}")
  echo http://$NODE_IP:$NODE_PORT

C:\Windows\system32>
```

Figure 68: Output install chart

Your web application should now be humming happily on <http://localhost:31120>. In a larger application, your charts will have dependencies on other charts. If the application dependencies are not available in an inaccessible repository, you can copy them under the chart's directory in your Helm chart. The preferred method is to declare your dependencies in the `/requirements.yaml` file in your chart. If your application requires **MySQL** to be present in the cluster as well, then write the following code in your `requirements.yaml` file.



Note: The repository name can either be a complete URL to the chart repository or an alias, such as `@stable`.

You can define **dependencies** in the following way.

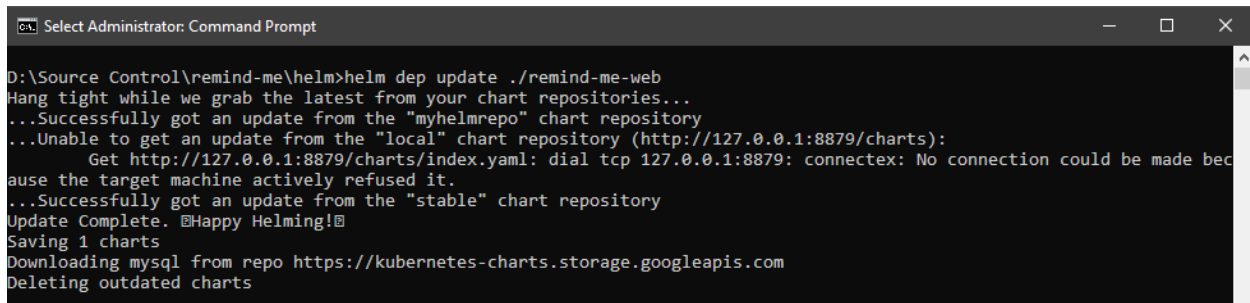
Code Listing 108: Defining dependencies

```
dependencies:
- name: mysql
  version: 0.10.2
  repository: '@stable'
```

To download the dependency package in your chart, execute the following command, which will download the **MySQL** chart inside the `/charts` directory of your chart.

Code Listing 109: Installing dependencies

```
helm dep update ./remind-me-web
```



```
D:\Source Control\remind-me\helm>helm dep update ./remind-me-web
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "myhelmrepo" chart repository
...Unable to get an update from the "local" chart repository (http://127.0.0.1:8879/charts):
Get http://127.0.0.1:8879/charts/index.yaml: dial tcp 127.0.0.1:8879: connectex: No connection could be made because the target machine actively refused it.
...Successfully got an update from the "stable" chart repository
Update Complete. @Happy Helming!@
Saving 1 charts
Downloading mysql from repo https://kubernetes-charts.storage.googleapis.com
Deleting outdated charts
```

Figure 69: Output update chart

Next, package the chart again and deploy it to your repository. Now, when you install the chart, you will see that MySQL objects will be deployed to your cluster as well.

Summary

In this chapter, we discussed how we can package our applications using Helm. Helm makes it easy for everyone to discover and deploy your applications on their clusters without writing their own specifications.

Finally, let us look at some of the exciting projects coming up in the Kubernetes ecosystem and see how we can get involved with the Kubernetes community.

Chapter 9 Next Steps

Kubernetes is one of the fastest-growing technologies in the DevOps world. In the world of containers, Kubernetes has become as significant as Docker. As covered in Chapter 2, Docker has introduced inbuilt support for Kubernetes. However, like any other technology, it is still evolving, and is under continuous development.

In this book, we have covered the primary components to deploy applications in Kubernetes. If you are interested in learning more or want to know what new stuff is available, you can explore some of the topics presented in the following sections.

Advanced Kubernetes networking

At the core of Kubernetes is its networking model. In Chapter 2, we saw that before adding nodes to our cluster, we need to have a **pod-network** add-on deployed in the cluster, which extends the Kubernetes existing networking to provide better pod-to-pod communication. Similarly, the container-to-container interaction can happen using pods and localhost communication. There is also the services concept in Kubernetes, which gives you the capability to interact with the external world.

Kubernetes follows a different networking model than Docker, where every pod has an IP address through which it can communicate with another component. This helps improve communication between the component and load balancing.

To learn more, you can start with official Kubernetes [documentation](#).

Kubernetes and serverless

Serverless is the next big thing in compute resources in which you don't even bother about the executing machine or OS, and just focus on running your code and paying only for resources you consume. If you compare it to traditional cloud models, it comes under the PaaS (Platform-as-a-Service) model, but provides more abstraction regarding VMs and operating systems, and enables you to focus more on your application (the idea of cloud). Now, since we have applications moving from traditional deployment to container-based deployment, we have serverless container services coming up, like Amazon Elastic Container Service and Azure Container Instances, which provide serverless infrastructure to run your containers.

For a simple application, this is good enough, but when you have complex containers and services, and they need constant interaction and monitoring, you need an orchestrator to manage them—enter Kubernetes.

Due to immense popularity and maturity, Kubernetes (after all, Google has been running it for a long time) is becoming a de facto orchestrator for managing containers and running applications smoothly at a large scale.

Kubernetes automation

We saw in Chapter 3 that we could easily deploy our application to Kubernetes clusters using the `kubectl` tool. We can also perform updates and manage different components of a cluster using the same command. However, when you have a complex cluster with many services and components, doing all these tasks manually becomes cumbersome and error-prone. Like any other DevOps technology, you can also automate Kubernetes cluster deployment using open-source tools and scripts. You can also use good old Jenkins (a popular open-source automation server), with Helm as a package manager, and automate your cluster deployment.

There are many open-source and paid automation tools available that help in automation of Kubernetes cluster. Some of the other popular Kubernetes automation tools are Keel and Buddy.

Keel

[Keel](#) is an open-source Kubernetes automation tool that helps in automating deployment updates. It is deployed in the Kubernetes cluster and manages subsequent cluster upgrades automatically by monitoring version updates in your repository. It is a self-hosted solution without any dependencies and doesn't require any specific storage. It depends on configuration files within the application or Helm charts to manage the cluster.

Buddy

[Buddy](#) is another continuous-delivery tool that can be used to automate cluster updates. You can perform the following tasks using Buddy:

- Managing configuration changes of a K8s deployment.
- Deploying code changes.
- Managing Docker file updates.
- Building Docker images and pushing them to the Docker registry.
- Applying new images to your K8s cluster.

Kubernetes and Windows

We all know that open source is the future of technology, and many companies have started embracing it wholeheartedly, including Microsoft. However, many enterprises have invested heavily in .NET and Windows, and it might not be immediately possible for them to migrate to other technologies. There are many advantages these enterprises can leverage if they move their deployment to container-based, even if the core technology stack is legacy.

To make Kubernetes available for all, the Kubernetes team is working tirelessly to have Kubernetes available for Windows Server containers. As of this book writing, the Kubernetes 1.9 release has introduced beta support for Windows Server containers, which can be used to have Windows-based containers deployed on Kubernetes.

General availability of Windows Server containers may happen very soon, and we expect it will be a popular and exciting feature.

Kubernetes Kustomize

Kubernetes [Kustomize](#) is a template-free configuration customization for Kubernetes.

In Kubernetes deployment, YAML are the main files that contain all the cluster configuration. To deploy the cluster in a different environment, say testing, staging, or production, usually a copy of the configuration file is maintained for each environment. Since each configuration has its own set of labels or resource limits, it quickly becomes difficult to maintain all these files, and a change made in one file needs to be manually propagated across all the YAMLs.

You can always maintain a standard copy and replace environment-specific values via scripts or tools, but it involves custom processing and the effort of scripting and understanding the existing configuration. Kustomize provides a solution to this problem by reading the configuration file and the Kubernetes API resource files it references, and then generating an entirely new resource to standard output. This output can then be directly utilized to build or update the cluster.

Kubernetes security

Security is one of the critical aspects of any software deployment, and Kubernetes is no exception. While Kubernetes by default has many inbuilt security mechanisms, there are many best practices you need to follow to make sure your cluster and your application running inside that cluster are both secure and free from malicious access. Some of these are explained in the following sections.

Restricting access to Kubernetes API

First and foremost, you should have all your cluster APIs exposed via SSL protocol only. With free SSL certificates sites like [LetsEncrypt](#), you can easily have one procured for your cluster, or you can buy one from the certificate providers. Moreover, Kubernetes supports authentication for its APIs, and you can use any authentication pattern to make sure your APIs are accessed by authorized personnel only.

Restricting kubelet

You can control both node and containers using the kubelet HTTPs endpoint, which is exposed unauthenticated by default. However, you can easily disable that using `--anonymous-auth=false` while starting the kubelet.

Controlling resource usage and access

Kubernetes gives you the flexibility to limit resource consumption for a cluster using the [resource quotas and limit ranges](#).

Resource quotas limit the aggregate resource consumption by the Kubernetes namespace. They restrict the number of resources and objects that can be created in a namespace to avoid having an uneven distribution of resources. This helps in maintaining the stability of the cluster.

Kubernetes on cloud

In the previous chapter, we saw how we could package our application using Helm, which helps us automate repeated deployment tasks. In our journey, we have seen many critical phases in a Kubernetes deployment, starting from installation, monitoring, logging, etc. In the world of cloud and Platform-as-a-Service, it seems redundant to do all these tasks manually and risk running into bottlenecks as the application grows. Like all other services, we have Kubernetes offerings also available from many cloud providers, including Microsoft, Google, and Amazon. They provide managed services for Kubernetes, which take care of most of the cluster management operations and leave you focusing only on your application.

Kubernetes and AWS

AWS provides support for both managed and unmanaged Kubernetes service. You can use EC2 instances and run your own Kubernetes cluster, or choose its managed service, Amazon Elastic Container Service for Kubernetes. You can use the **Kops** tool to deploy your Kubernetes cluster on AWS EC2 automatically.

Amazon Elastic Container Service for Kubernetes ([EKS](#)) is a fully managed offering from Amazon to deploy, manage, and scale containerized applications. Amazon EKS provides a managed control pane that runs across different geography to ensure high availability and scalability. You can think of it as a Kubernetes master. It ensures that all your components and resources inside the cluster are running as expected.

Amazon EKS also provides integration with existing AWS services, like IAM (AWS Identity and Access Management) and VPC (AWS Virtual Private Cloud) to ensure better security and accessibility. It provides inbuilt logging through integration with the AWS CloudTrail service (a logging and monitoring service), which also helps in cluster monitoring. Amazon EKS also supports all **kubectl** commands.

Kubernetes and Azure

Microsoft Azure also supports both managed and unmanaged Kubernetes services. You can use Azure Virtual Machine with accelerated networking to create your Kubernetes cluster, or choose its managed service, called Azure Kubernetes Service.

Microsoft's Azure Kubernetes Service ([AKS](#)) is a managed Kubernetes service that handles most of the complexity of managing and operating the cluster. Many of the components of Kubernetes are abstracted by AKS, including the most essential Kubernetes master. It also takes care of monitoring and central logging for you. Since the master node is abstracted, you are not charged for it, and you only pay for worker node(s) in your cluster. Also, it supports most of the **kubectl** commands, so you don't have to learn a new one for AKS. Installing AKS is relatively easy; you can set it up using CLI or the Azure portal, and even automate it like other Azure services.

Kubernetes and GCP

Like AWS and Azure, Google Cloud Platform ([GCP](#)) also supports setting up Kubernetes clusters in GCP Compute Engine instances. It also has a managed service, Google Kubernetes Engine. Since Kubernetes was originally a Google product, and Google is still the major contributor to it, it has slightly better integration with Kubernetes than other cloud providers.

Google Kubernetes Engine (GKE) is a fully managed service to run your containers in a Kubernetes cluster without worrying about its operational maintenance. It provides inbuilt logging, security, monitoring, autoscale, and dashboard capabilities.

Kubernetes communities

Although introduced by Google, Kubernetes is now backed by the Cloud Native Computing Foundation ([CNCF](#)). Being open source has helped it become one of the popular orchestrator systems. As a result, like other open-source systems, it has a wide community base that keeps on growing day by day. CNCF ensures that it remains vendor-neutral and is compatible with other OSS products like Prometheus so that the developer community gets the most benefit from it. From its website: *“Cloud native computing uses an open source software stack to deploy applications as microservices, packaging each part into its container, and dynamically orchestrating those containers to optimize resource utilization.”*

The comprehensive list of Kubernetes working groups and communities can be found on [GitHub](#).

Summary

In this chapter, we have covered Kubernetes’s upcoming releases like Kubernetes Kustomize, Windows Server containers, and Kubernetes automation tools. We have also briefly covered managed cloud offerings in Kubernetes and touched on various Kubernetes security concepts. Being the leader in container orchestration, Kubernetes is growing every day, and it has now become one of the default skills for a developer or DevOps engineer.

Kubernetes is a vast platform, but we have tried to cover it in a precise and interactive manner. In this short title, we discussed some of the core components of Kubernetes. It’s essential that we keep ourselves up to date with the latest developments in the wonderful container world, as it is very likely to replace the traditional deployment model very soon. What containers are doing today to the traditional deployment models is what cloud did to the bare metal deployments a decade back, and as developers, it’s our responsibility to stay on top of modern technology trends. We wish you success on your learning journey.