Introduction

Most new companies today run their business processes in the cloud. Newer startups and enterprises which realized early enough the direction technology was headed developed their applications for the cloud.

Not all companies were so fortunate. Some built their success decades ago on top of legacy technologies - **monolithic applications** with all components tightly coupled and almost impossible to separate, a nightmare to manage and deployed on super expensive hardware.

If working for an organization which refers to their main business application "the black box”, where nobody knows what happens inside and most logic was never documented, leaving everyone clueless as to what and how things happen from the moment a request enters the application until a response comes out, and you are tasked to convert this business application into a cloud-ready set of applications, then you may be in for a very long and bumpy ride.

Learning Objectives

- Explain what a monolith is.
- Discuss the monolith's challenges in the cloud.
- Explain the concept of microservices.
- Discuss microservices advantages in the cloud.
- Describe the transformation path from a monolith to microservices.

The Legacy Monolith

Although most enterprises believe that the cloud will be the new home for legacy apps, not all legacy apps are a fit for the cloud, at least not yet.

Moving an application to the cloud should be as easy as walking on the beach and collecting pebbles in a bucket and easily carry them wherever needed. A 1000-ton boulder, on the other hand, is not easy to carry at all. This boulder represents the monolith application - sedimented layers of features and redundant logic translated into thousands of lines of code, written in a single, not so modern programming language, based on outdated software architecture patterns and principles.

In time, the new features and improvements added to code complexity, making development more challenging - loading, compiling, and building times increase with every new update. However, there is some ease in administration as the application is running on a single server, ideally a Virtual Machine or a Mainframe.

A **monolith** has a rather expensive taste in hardware. Being a large, single piece of software which continuously grows, it has to run on a single system which has to satisfy its compute, memory, storage, and networking requirements. The hardware of such capacity is both complex and pricey.

Since the entire monolith application runs as a single process, the scaling of individual features of the monolith is almost impossible. It internally supports a hardcoded number of connections and operations. However, scaling the entire application means to manually deploy a new instance of the monolith on another server, typically behind a load balancing appliance - another pricey solution.

During upgrades, patches or migrations of the monolith application - downtimes occur and maintenance windows have to be planned as disruptions in service are expected to impact clients. While there are solutions to minimize downtimes to customers by setting up monolith applications in a highly available active/passive configuration, it may still be challenging for system engineers to keep all systems at the same patch level.

The Modern Microservice

Pebbles, as opposed to the 1000-ton boulder, are much easier to handle. They are carved out of the monolith, separated from one another, becoming distributed components each described by a set of specific characteristics. Once weighed all together, the pebbles make up the weight of the entire boulder. These pebbles represent loosely coupled microservices, each performing a specific business function. All the functions grouped together form the overall functionality of the original monolithic application. Pebbles are easy to select and group together based on color, size, shape, and require minimal effort to relocate when needed. Try relocating the 1000-ton boulder, effortlessly.

Microservices can be deployed individually on separate servers provisioned with fewer resources - only what is required by each service and the host system itself.

Microservices-based architecture is aligned with Event-driven Architecture and Service-Oriented Architecture (SOA) principles, where complex applications are composed of small independent processes which communicate with each other through APIs over a network. APIs allow access by other internal services of the same application or external, third-party services and applications.

Each microservice is developed and written in a modern programming language, selected to be the best suitable for the type of service and its business function. This offers a great deal of flexibility when matching microservices with specific hardware when required, allowing deployments on inexpensive commodity hardware.

Although the distributed nature of microservices adds complexity to the architecture, one of the greatest benefits of microservices is scalability. With the overall application becoming modular, each microservice can be scaled individually, either manually or automated through demand-based autoscaling.

Seamless upgrades and patching processes are other benefits of microservices architecture. There is virtually no downtime and no service disruption to clients because upgrades are rolled out seamlessly - one service at a time, rather than having to re-compile, re-build and re-start an entire monolithic application. As a result, businesses are able to develop and roll-out new features and updates a lot faster, in an agile approach, having separate teams focusing on separate features, thus being more productive and cost-effective.

Newer, more modern enterprises possess the knowledge and technology to build cloud-native applications that power their business.

Unfortunately, that is not the case for established enterprises running on legacy monolithic applications. Some have tried to run monoliths as microservices, and as one would expect, it did not work very well. The lessons learned were that a monolithic size multi-process application cannot run as a microservice and that other options had to be explored. The next natural step in the path of the monolith to microservices transition was refactoring. However, migrating a decades-old application to the cloud through refactoring poses serious challenges and the enterprise faces the refactoring approach dilemma: a "Big-bang" approach or an incremental refactoring.

A so-called "Big-bang" approach focuses all efforts with the refactoring of the monolith, postponing the development and implementation of any new features - essentially delaying progress and possibly, in the process, even breaking the core of the business, the monolith.

An incremental refactoring approach guarantees that new features are developed and implemented as modern microservices which are able to communicate with the monolith through APIs, without appending to the monolith's code. In the meantime, features are refactored out of the monolith which slowly fades away while all, or most its functionality is modernized into microservices. This incremental approach offers a gradual transition from a legacy monolith to modern microservices architecture and allows for phased migration of application features into the cloud.

Once an enterprise chose the refactoring path, there are other considerations in the process. Which business components to separate from the monolith to become distributed microservices, how to decouple the databases from the application to separate data complexity from application logic, and how to test the new microservices and their dependencies, are just a few of the decisions an enterprise is faced with during refactoring.

The refactoring phase slowly transforms the monolith into a cloud-native application which takes full advantage of cloud features, by coding in new programming languages and applying modern architectural patterns. Through refactoring, a legacy monolith application receives a second chance at life - to live on as a modular system adapted to fully integrate with today's fast-paced cloud automation tools and services.

Challenges

The refactoring path from a monolith to microservices is not smooth and without challenges. Not all monoliths are perfect candidates for refactoring, while some may not even "survive" such a modernization phase. When deciding whether a monolith is a possible candidate for refactoring, there are many possible issues to consider.

When considering a legacy Mainframe based system, written in older programming languages - Cobol or Assembler, it may be more economical to just re-build it from the ground up as a cloud-native application. A poorly designed legacy application should be re-designed and re-built from scratch following modern architectural patterns for microservices and even containers. Applications tightly coupled with data stores are also poor candidates for refactoring.

Once the monolith survived the refactoring phase, the next challenge is to design mechanisms or find suitable tools to keep alive all the decoupled modules to ensure application resiliency as a whole.

Choosing runtimes may be another challenge. If deploying many modules on a single physical or virtual server, chances are that different libraries and runtime environment may conflict with one another causing errors and failures. This forces deployments of single modules per servers in order to separate their dependencies - not an economical way of resource management, and no real segregation of libraries and runtimes, as each server also has an underlying Operating System running with its libraries, thus consuming server resources - at times the OS consuming more resources than the application module itself.

Ultimately application containers came along, providing encapsulated lightweight runtime environments for application modules. Containers promised consistent software environments for developers, testers, all the way from Development to Production. Wide support of containers ensured application portability from physical bare-metal to Virtual Machines, but this time with multiple applications deployed on the very same server, each running in their own execution environments isolated from one another, thus avoiding conflicts, errors, and failures. Other features of containerized application environments are higher server utilization, individual module scalability, flexibility, interoperability and easy integration with automation tools.

Success Stories

Although a challenging process, moving from monoliths to microservices is a rewarding journey especially once a business starts to see growth and success delivered by a refactored application system. Below we are listing only a handful of the success stories of companies which rose to the challenge to modernize their monolith business applications. A detailed list of success stories is available at the Kubernetes website: Kubernetes User Case Studies.

- AppDirect - an end-to-end commerce platform provider, started from a complex monolith application and through refactoring was able to retain limited functionality monoliths receiving very few commits, but all new features implemented as containerized microservices.

- box - a cloud storage solutions provider, started from a complex monolith architecture and through refactoring was able to decompose it into microservices.
- Crowdfire - a content management solutions provider, successfully broke down their initial monolith into microservices.
- GolfNow - a technology and services provider, decided to break their monoliths apart into containerized microservices.
- Pinterest - a social media services provider, started the refactoring process by first migrating their monolith API.

Container Orchestration

With container images, we confine the application code, its runtime, and all of its dependencies in a pre-defined format. And, with container runtimes like **runC**, **containerd**, or **rkt** we can use those pre-packaged images, to create one or more containers. All of these runtimes are good at running containers on a single host. But, in practice, we would like to have a fault-tolerant and scalable solution, which can be achieved by creating a single **controller**/**management unit**, after connecting multiple nodes together. This controller/management unit is generally referred to as a **container orchestrator.**

In this chapter, we will explore why we should use container orchestrators, different implementations of container orchestrators, and where to deploy them.

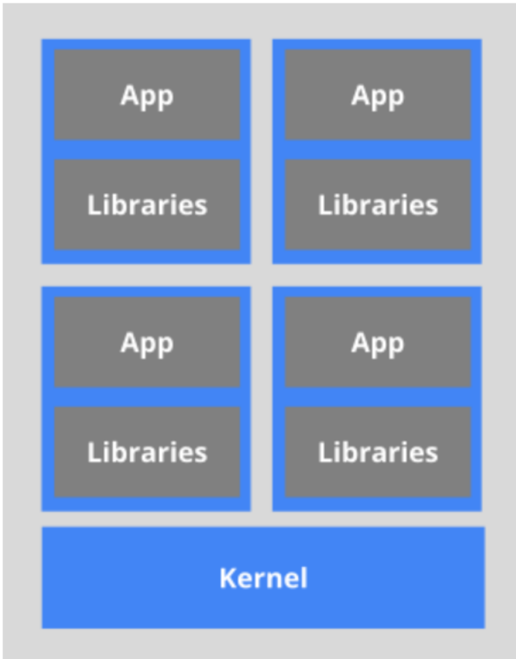
By the end of this chapter, you should be able to:

- Define the concept of container orchestration.
- Explain the reasons for doing container orchestration.
- Discuss different container orchestration options.
- Discuss different container orchestration deployment options.

What Are Containers?

Before we dive into container orchestration, let's review first what containers are.

Containers are application-centric methods to deliver high-performing, scalable applications on any infrastructure of your choice. Containers are best suited to deliver microservices by providing portable, isolated virtual environments for applications to run without interference from other running applications.



Containers

Microservices are lightweight applications written in various modern programming languages, with specific dependencies, libraries and environmental requirements. To ensure that an application has everything it needs to run successfully it is packaged together with its dependencies.

Containers encapsulate microservices and their dependencies but do not run them directly. Containers run container images.

A **container image** bundles the application along with its runtime and dependencies, and a container is deployed from the container image offering an isolated executable environment for the application. Containers can be deployed from a specific image on many platforms, such as workstations, Virtual Machines, public cloud, etc.

What Is Container Orchestration?

In Development (Dev) environments, running containers on a single host for development and testing of applications may be an option. However, when migrating to Quality Assurance (QA) and Production (Prod) environments, that is no longer a viable option because the applications and services need to meet specific requirements:

- Fault-tolerance
- On-demand scalability
- Optimal resource usage
- Auto-discovery to automatically discover and communicate with each other
- Accessibility from the outside world
- Seamless updates/rollbacks without any downtime.

Container orchestrators are tools which group systems together to form clusters where containers' deployment and management is automated at scale while meeting the requirements mentioned above.

Container Orchestrators

With enterprises containerizing their applications and moving them to the cloud, there is a growing demand for container orchestration solutions. While there are many solutions available, some are mere re-distributions of well-established container orchestration tools, enriched with features and, sometimes, with certain limitations in flexibility.

Although not exhaustive, the list below provides a few different container orchestration tools and services available today:

- **Amazon Elastic Container Service**
  Amazon Elastic Container Service (ECS) is a hosted service provided by Amazon Web Services (AWS) to run Docker containers at scale on its infrastructure.
- **Azure Container Instances**
  Azure Container Instance (ACI) is a basic container orchestration service provided by Microsoft Azure.
- **Azure Service Fabric**
  Azure Service Fabric is an open source container orchestrator provided by Microsoft Azure.

- **Kubernetes**
  Kubernetes is an open source orchestration tool, started by Google, part of the Cloud Native Computing Foundation (CNCF) project.
- **Marathon**
  Marathon is a framework to run containers at scale on Apache Mesos.
- **Nomad**
  Nomad is the container orchestrator provided by HashiCorp.
- **Docker Swarm**
  Docker Swarm is a container orchestrator provided by Docker, Inc. It is part of Docker Engine.

We have explored different container orchestrators in another edX MOOC, *Introduction to Cloud Infrastructure Technologies* (*LFS151x*). We highly recommend that you take *LFS151x*.

Why Use Container Orchestrators?

Although we can manually maintain a couple of containers or write scripts for dozens of containers, orchestrators make things much easier for operators especially when it comes to managing hundreds and thousands of containers running on a global infrastructure.

Most container orchestrators can:

- Group hosts together while creating a cluster
- Schedule containers to run on hosts in the cluster based on resources availability
- Enable containers in a cluster to communicate with each other regardless of the host they are deployed to in the cluster
- Bind containers and storage resources
- Group sets of similar containers and bind them to load-balancing constructs to simplify access to containerized applications by creating a level of abstraction between the containers and the user
- Manage and optimize resource usage

- Allow for implementation of policies to secure access to applications running inside containers.

With all these configurable yet flexible features, container orchestrators are an obvious choice when it comes to managing containerized applications at scale. In this course, we will explore **Kubernetes**, one of the most in-demand container orchestration tools available today.

Where to Deploy Container Orchestrators?

Most container orchestrators can be deployed on the infrastructure of our choice - on bare metal, Virtual Machines, on-premise, or the public cloud. Kubernetes, for example, can be deployed on a workstation, with or without a local hypervisor such as Oracle VirtualBox, inside a company's data center, in the cloud on AWS Elastic Compute Cloud (EC2) instances, Google Compute Engine (GCE) VMs, DigitalOcean Droplets, OpenStack, etc.

There are turnkey solutions which allow Kubernetes clusters to be installed, with only a few commands, on top of cloud Infrastructures-as-a-Service, such as GCE, AWS EC2, Docker Enterprise, IBM Cloud, Rancher, VMware, Pivotal, and multi-cloud solutions through IBM Cloud Private and StackPointCloud.

Last but not least, there is the managed container orchestration as-a-Service, more specifically the managed Kubernetes as-a-Service solution, offered and hosted by the major cloud providers, such as Google Kubernetes Engine (GKE), Amazon Elastic Container Service for Kubernetes (Amazon EKS), Azure Kubernetes Service (AKS), IBM Cloud Kubernetes Service, DigitalOcean Kubernetes, Oracle Container Engine for Kubernetes, etc. These shall be explored in one of the later chapters.

Kubernetes

In this chapter, we will explain what **Kubernetes** is, its features, and the reasons why you should use it. We will explore the evolution of Kubernetes from **Borg**, which is a cluster manager created by Google.

We will also talk about the **Cloud Native Computing Foundation (CNCF)**, which currently hosts the Kubernetes project, along with other cloud-native projects, like Prometheus, Fluentd, rkt, containerd, etc.

By the end of this chapter, you should be able to:

- Define Kubernetes.
- Explain the reasons for using Kubernetes.
- Discuss the features of Kubernetes.
- Discuss the evolution of Kubernetes from Borg.
- Explain what the Cloud Native Computing Foundation does.

What Is Kubernetes?

According to the [Kubernetes website](),

"Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."

Kubernetes comes from the Greek word **κυβερνήτης**, which means *helmsman* or *ship pilot*. With this analogy in mind, we can think of Kubernetes as the pilot on a ship of containers.

Kubernetes is also referred to as **k8s**, as there are 8 characters between *k* and *s*.

Kubernetes is highly inspired by the Google Borg system, a container orchestrator for its global operations for more than a decade. It is an open source project written in the Go language and licensed under the [Apache License, Version 2.0]().

Kubernetes was started by Google and, with its v1.0 release in July 2015, Google donated it to the Cloud Native Computing Foundation (CNCF). We will talk more about CNCF later in this chapter.

New Kubernetes versions are released in 3 months cycles. The current stable version is 1.14 (as of May 2019).

From Borg to Kubernetes

According to the abstract of Google's Borg paper, published in 2015,

"Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines".

For more than a decade, Borg has been Google's secret, running its worldwide containerized workloads in production. Services we use from Google, such as Gmail, Drive, Maps, Docs, etc., they are all serviced using Borg.

Some of the initial authors of Kubernetes were Google employees who have used Borg and developed it in the past. They poured in their valuable knowledge and experience while designing Kubernetes. Some of the features/objects of Kubernetes that can be traced back to Borg, or to lessons learned from it, are:

- API servers
- Pods
- IP-per-Pod
- Services
- Labels.

We will explore all of them, and more, in this course.

Kubernetes Features I

Kubernetes offers a very rich set of features for container orchestration. Some of its fully supported features are:

- **Automatic bin packing**
  Kubernetes automatically schedules containers based on resource needs and constraints, to maximize utilization without sacrificing availability.
- **Self-healing**
  Kubernetes automatically replaces and reschedules containers from failed nodes. It kills and restarts containers unresponsive to health checks, based on existing rules/policy. It also prevents traffic from being routed to unresponsive containers.
- **Horizontal scaling**
  With Kubernetes applications are scaled manually or automatically based on CPU or custom metrics utilization.
- **Service discovery and Load balancing**
  Containers receive their own IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set.

Kubernetes Features II

Some other fully supported Kubernetes features are:

- **Automated rollouts and rollbacks**
  Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, constantly monitoring the application's health to prevent any downtime.
- **Secret and configuration management**
  Kubernetes manages secrets and configuration details for an application separately from the container image, in order to avoid a re-build of the respective image. Secrets consist of confidential information passed to the application without revealing the sensitive content to the stack configuration, like on GitHub.
- **Storage orchestration**
  Kubernetes automatically mounts software-defined storage (SDS) solutions to containers from local storage, external cloud providers, or network storage systems.
- **Batch execution**
  Kubernetes supports batch execution, long-running jobs, and replaces failed containers.

There are many other features besides the ones we just mentioned, and they are currently in alpha/beta phase. They will add great value to any Kubernetes deployment once they become stable features. For example, support for role-based access control (RBAC) is stable as of the Kubernetes 1.8 release.

Why Use Kubernetes?

In addition to its fully-supported features, Kubernetes is also portable and extensible. It can be deployed in many environments such as local or remote Virtual Machines, bare metal, or in public/private/hybrid/multi-cloud setups. It supports and it is supported by many 3rd party open source tools which enhance Kubernetes' capabilities and provide a feature-rich experience to its users.

Kubernetes' architecture is modular and pluggable. Not only that it orchestrates modular, decoupled microservices type applications, but also its architecture follows decoupled microservices patterns. Kubernetes' functionality can be extended by writing custom resources, operators, custom APIs, scheduling rules or plugins.

For a successful open source project, the community is as important as having great code. Kubernetes is supported by a thriving community across the world. It has more than 2,000 contributors, who, over time, have pushed over 77,000 commits. There are meet-up groups in different cities and countries which meet regularly to discuss Kubernetes and its ecosystem. There are *Special Interest Groups* (SIGs), which focus on special topics, such as scaling, bare metal, networking, etc. We will talk more about them in our last chapter, *Kubernetes Communities*.

Kubernetes Users

With just a few years since its debut, many enterprises of various sizes run their workloads using Kubernetes. It is a solution for workload management in banking, education, finance and investments, gaming, information technology, media and streaming, online retail, ridesharing, telecommunications, and many other industries. There are numerous user case studies and success stories on the Kubernetes website:

- BlaBlaCar
- BlackRock
- Box

- eBay
- Haufe Group
- Huawei
- IBM
- ING
- Nokia
- Pearson
- Wikimedia

Cloud Native Computing Foundation (CNCF)

CNCF hosts a multitude of projects, with more to be added in the future. CNCF provides resources to each of the projects, but, at the same time, each project continues to operate independently under its pre-existing governance structure and with its existing maintainers. Projects within CNCF are categorized based on achieved status: Sandbox, Incubating, and Graduated. At the time this course was created, there were six projects with Graduated status and many more still Incubating and in the Sandbox.

Graduated projects:

- Kubernetes for container orchestration
- Prometheus for monitoring
- Envoy for service mesh
- CoreDNS for service discovery
- containerd for container runtime
- Fluentd for logging

Incubating projects:

- rkt and CRI-O for container runtime
- Linkerd for service mesh

- etcd for key/value store
- gRPC for remote procedure call (RPC)
- CNI for networking API
- Harbor for registry
- Helm for package management
- Rook and Vitess for cloud-native storage
- Notary for security
- TUF for software updates
- NATS for messaging
- Jaeger and OpenTracing for distributed tracing
- Open Policy Agent for policy.

There are many projects in the CNCF Sandbox geared towards metrics, monitoring, identity, scripting, serverless, nodeless, edge, etc.

As we can see, the CNCF projects cover the entire lifecycle of a cloud-native application, from its execution using container runtimes, to its monitoring and logging. This is very important to meet the CNCF goal.

CNCF and Kubernetes

For Kubernetes, the Cloud Native Computing Foundation:

- Provides a neutral home for the Kubernetes trademark and enforces proper usage
- Provides license scanning of core and vendor code
- Offers legal guidance on patent and copyright issues
- Creates open source learning curriculum, training, and certification for both Kubernetes administrators and application developers

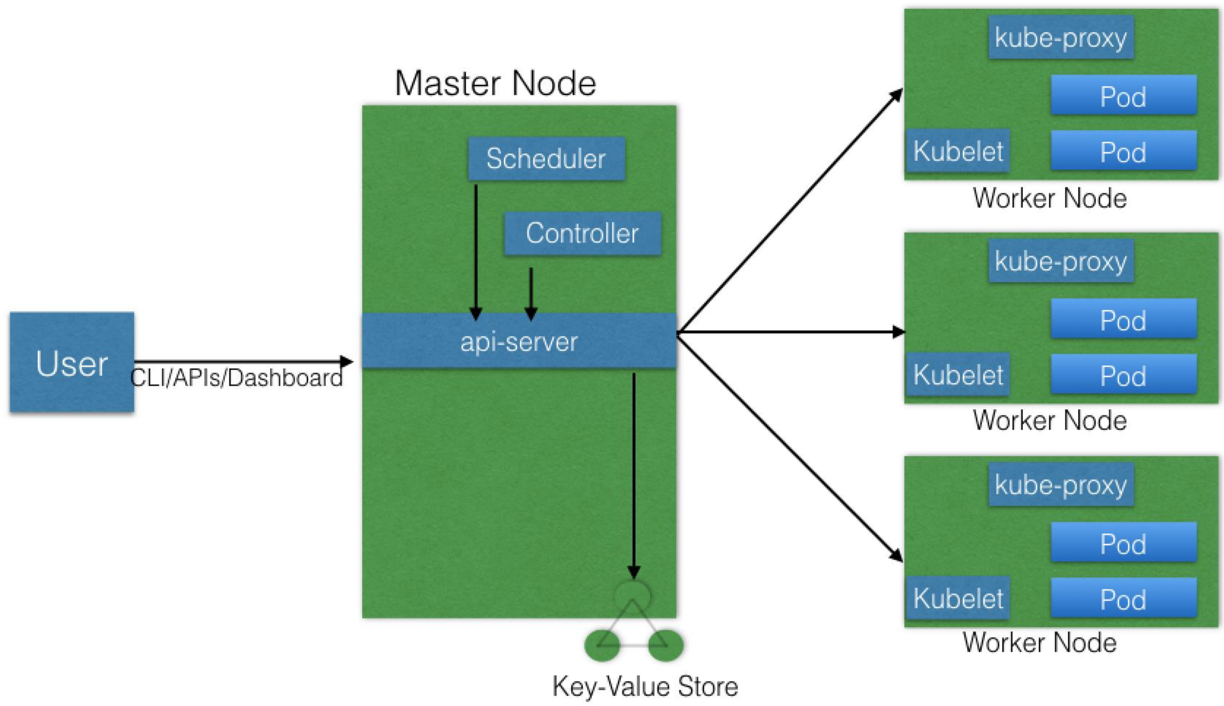
- Manages a software conformance [working group]
- Actively markets Kubernetes
- Supports ad hoc activities
- Funds conferences and meetup events.

Learning Objectives

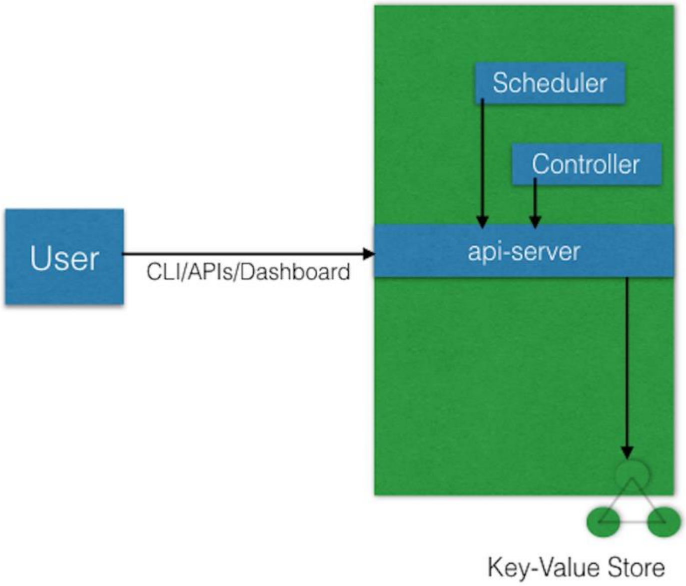
By the end of this chapter, you should be able to:

- Discuss the Kubernetes architecture.
- Explain the different components for master and worker nodes.
- Discuss about cluster state management with etcd.
- Review the Kubernetes network setup requirements.

Kubernetes Architecture



Master Node



Kubernetes Master Node

It is important to keep the control plane running at all costs. Losing the control plane may introduce downtimes, causing service disruption to clients, with possible loss of business. To ensure the control plane's fault tolerance, master node replicas are added to the cluster, configured in High-Availability (HA) mode. While only one of the master node replicas actively manages the cluster, the control plane components stay in sync across the master node replicas. This type of configuration adds resiliency to the cluster's control plane, should the active master node replica fail.

To persist the Kubernetes cluster's state, all cluster configuration data is saved to etcd. However, **etcd** is a distributed key-value store which only holds cluster state related data, no client workload data. etcd is configured on the master node (stacked) or on its dedicated host (external) to reduce the chances of data store loss by decoupling it from the control plane agents.

When stacked, HA master node replicas ensure etcd resiliency as well. Unfortunately, that is not the case of external etcds, when the etcd hosts have to be separately replicated for HA mode configuration.

Master Node Components

A master node has the following components:

- API server
- Scheduler
- Controller managers
- etcd.

In the next few sections, we will discuss them in more detail.

Master Node Components: API Server

All the administrative tasks are coordinated by the **kube-apiserver**, a central control plane component running on the master node. The API server intercepts RESTful calls from users, operators and external agents, then validates and processes them. During processing the API server reads the Kubernetes cluster's current state from the etcd, and after a call's execution, the resulting state of the Kubernetes cluster is saved in the distributed key-value data store for persistence. The API server is the only master plane component to talk to the etcd data store, both to read and to save Kubernetes cluster state information from/to it - acting as a middle-man interface for any other control plane agent requiring to access the cluster's data store.

The API server is highly configurable and customizable. It also supports the addition of custom API servers, when the primary API server becomes a proxy to all secondary custom API servers and routes all incoming RESTful calls to them based on custom defined rules.

Master Node Components: Scheduler

The role of the **kube-scheduler** is to assign new objects, such as pods, to nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new object's requirements. The scheduler obtains from etcd, via the API server, resource usage data for each worker node in the cluster. The scheduler also receives from the API server the new object's requirements which are part of its configuration data. Requirements may include constraints that users and operators set, such as scheduling work on a node labeled with **disk==ssd** key/value pair. The scheduler also takes into account Quality of Service (QoS) requirements, data locality, affinity, anti-affinity, taints, toleration, etc.

The scheduler is highly configurable and customizable. Additional custom schedulers are supported, then the object's configuration data should include the name of the custom scheduler expected to make the scheduling decision for that particular object; if no such data is included, the default scheduler is selected instead.

A scheduler is extremely important and quite complex in a multi-node Kubernetes cluster. In a single-node Kubernetes cluster, such as the one explored later in this course, the scheduler's job is quite simple.

Master Node Components: Controller Managers

The **controller managers** are control plane components on the master node running controllers to regulate the state of the Kubernetes cluster. Controllers are watch-loops continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from etcd data store via the API server). In case of a mismatch corrective action is taken in the cluster until its current state matches the desired state.

The **kube-controller-manager** runs controllers responsible to act when nodes become unavailable, to ensure pod counts are as expected, to create endpoints, service accounts, and API access tokens.

The **cloud-controller-manager** runs controllers responsible to interact with the underlying infrastructure of a cloud provider when nodes become unavailable, to manage storage volumes when provided by a cloud service, and to manage load balancing and routing.

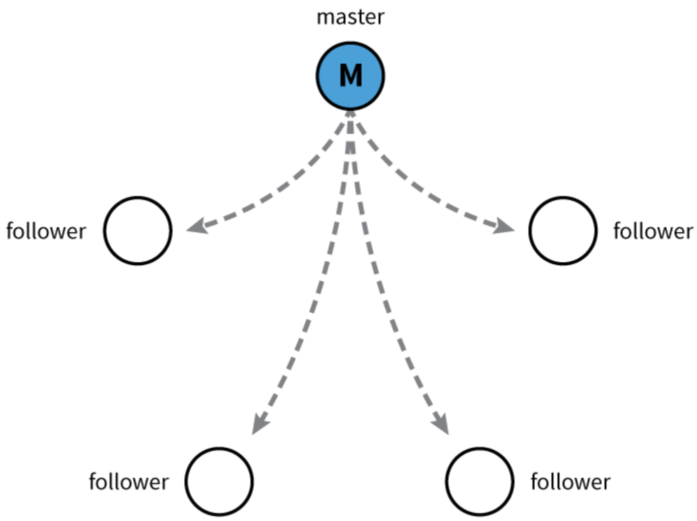
Master Node Components: etcd

etcd is a distributed key-value data store used to persist a Kubernetes cluster's state. New data is written to the data store only by appending to it, data is never replaced in the data store. Obsolete data is compacted periodically to minimize the size of the data store.

Out of all the control plane components, only the API server is able to communicate with the etcd data store.

etcd's CLI management tool provides backup, snapshot, and restore capabilities which come in handy especially for a single etcd instance Kubernetes cluster - common in Development and learning environments. However, in Stage and Production environments, it is extremely important to replicate the data stores in HA mode, for cluster configuration data resiliency.

Some Kubernetes cluster bootstrapping tools, by default, provision stacked etcd master nodes, where the data store runs alongside and shares resources with the other control plane components on the same master node. For data store isolation from the control plane components, the bootstrapping process can be configured for an external etcd, where the data store is provisioned on a dedicated separate host, thus reducing the chances of an etcd failure. Both stacked and external etcd configurations support HA configurations. etcd is based on the Raft Consensus Algorithm which allows a collection of machines to work as a coherent group that can survive the failures of some of its members. At any given time, one of the nodes in the group will be the master, and the rest of them will be the followers. Any node can be treated as a master.

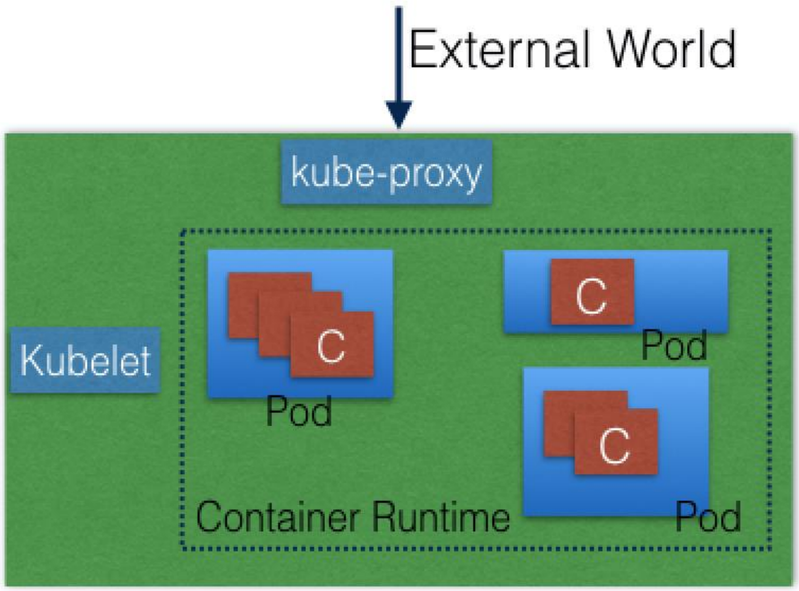


Master and Followers

etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.

Worker Node

A **worker node** provides a running environment for client applications. Though containerized microservices, these applications are encapsulated in Pods, controlled by the cluster control plane agents running on the master node. Pods are scheduled on worker nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world. A Pod is the smallest scheduling unit in Kubernetes. It is a logical collection of one or more containers scheduled together. We will explore them further in later chapters.



Kubernetes Worker Node

Also, to access the applications from the external world, we connect to worker nodes and not to the master node. We will dive deeper into this in future chapters.

Worker Node Components

A worker node has the following components:

- Container runtime
- kubelet
- kube-proxy
- Addons for DNS, Dashboard, cluster-level monitoring and logging.

In the next few sections, we will discuss them in more detail.

Worker Node Components: Container Runtime

Although Kubernetes is described as a "container orchestration engine", it does not have the capability to directly handle containers. In order to run and manage a container's lifecycle, Kubernetes requires a **container runtime** on the node where a

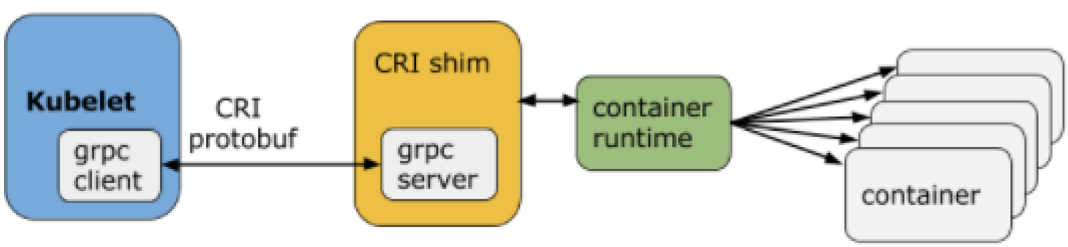
Pod and its containers are to be scheduled. Kubernetes supports many container runtimes:

- Docker - although a container platform which uses **containerd** as a container runtime, it is the most widely used container runtime with Kubernetes
- CRI-O - a lightweight container runtime for Kubernetes, it also supports Docker image registries
- containerd - a simple and portable container runtime providing robustness
- rkt - a pod-native container engine, it also runs Docker images
- rktlet - a Kubernetes Container Runtime Interface (CRI) implementation using **rkt**.

Worker Node Components: kubelet

The **kubelet** is an agent running on each node and communicates with the control plane components from the master node. It receives Pod definitions, primarily from the API server, and interacts with the container runtime on the node to run containers associated with the Pod. It also monitors the health of the Pod's running containers.

The kubelet connects to the container runtime using Container Runtime Interface (CRI). CRI consists of protocol buffers, gRPC API, and libraries.



Container Runtime Interface
(Retrieved from blog.kubernetes.io)

As shown above, the kubelet acting as grpc client connects to the CRI shim acting as grpc server to perform container and image operations. CRI implements two services: **ImageService** and **RuntimeService**. The **ImageService** is responsible for all the image-related operations, while the **RuntimeService** is responsible for all the Pod and container-related operations.

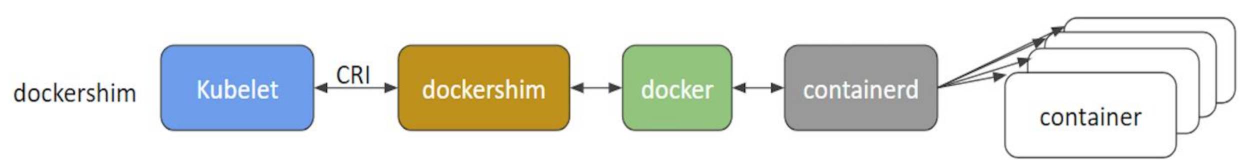
Container runtimes used to be hard-coded in Kubernetes, but with the development of CRI, Kubernetes is more flexible now and uses different container runtimes without the need to recompile. Any container runtime that implements CRI can be used by Kubernetes to manage Pods, containers, and container images.

In the next section, we will discuss some of the CRI shims.

Worker Node Components: kubelet - CRI shims

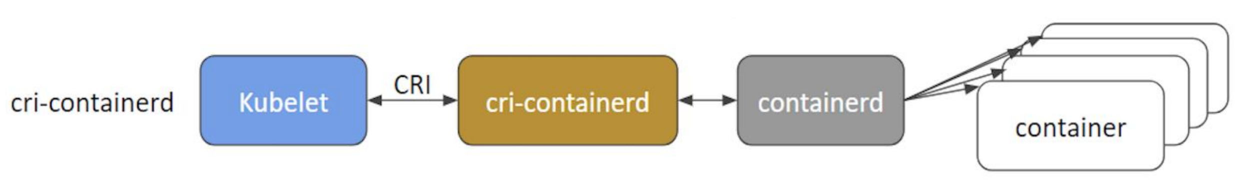
Below you will find some examples of CRI shims:

- **dockershim**
  With dockershim, containers are created using Docker installed on the worker nodes. Internally, Docker uses containerd to create and manage containers.



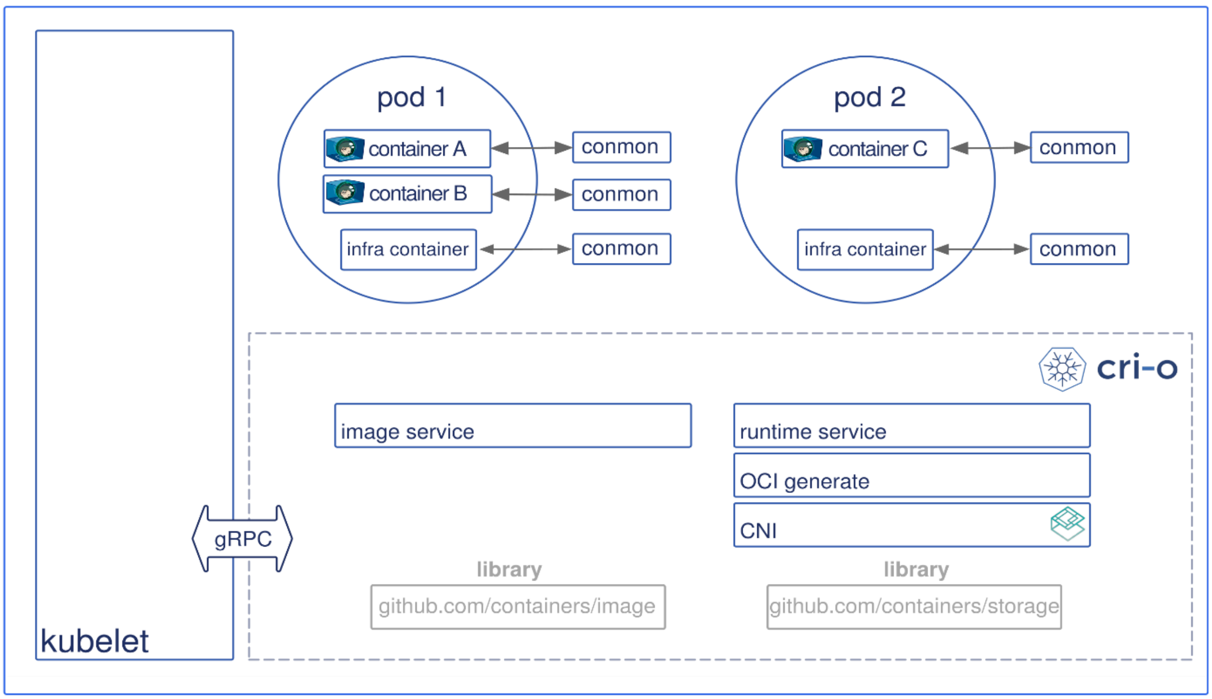
dockershim
(Retrieved from blog.kubernetes.io)

- **cri-containerd**
  With cri-containerd, we can directly use Docker's smaller offspring containerd to create and manage containers.



cri-containerd
(Retrieved from blog.kubernetes.io)

- **CRI-O**
  CRI-O enables using any Open Container Initiative (OCI) compatible runtimes with Kubernetes. At the time this course was created, CRI-O supported runC and Clear Containers as container runtimes. However, in principle, any OCI-compliant runtime can be plugged-in.



CRI-O
(Retrieved from cri-o.io)

Worker Node Components: kube-proxy

The **kube-proxy** is the network agent which runs on each node responsible for dynamic updates and maintenance of all networking rules on the node. It abstracts the details of Pods networking and forwards connection requests to Pods.

We will explore Pod networking in more detail in later chapters.

Worker Node Components: Addons

Addons are cluster features and functionality not yet available in Kubernetes, therefore implemented through 3rd-party pods and services.

- **DNS** - cluster DNS is a DNS server required to assign DNS records to Kubernetes objects and resources
- **Dashboard** - a general purposed web-based user interface for cluster management
- **Monitoring** - collects cluster-level container metrics and saves them to a central data store
- **Logging** - collects cluster-level container logs and saves them to a central log store for analysis.

Networking Challenges

Decoupled microservices based applications rely heavily on networking in order to mimic the tight-coupling once available in the monolithic era. Networking, in general, is not the easiest to understand and implement. Kubernetes is no exception - as a containerized microservices orchestrator is needs to address 4 distinct networking challenges:

- Container-to-container communication inside Pods
- Pod-to-Pod communication on the same node and across cluster nodes
- Pod-to-Service communication within the same namespace and across cluster namespaces
- External-to-Service communication for clients to access applications in a cluster.

All these networking challenges must be addressed before deploying a Kubernetes cluster. Next, we will see how we solve these challenges.

Container-to-Container Communication Inside Pods

Making use of the underlying host operating system's kernel features, a container runtime creates an isolated network space for each container it starts. On Linux, that isolated network space is referred to as a **network namespace**. A network namespace is shared across containers, or with the host operating system.

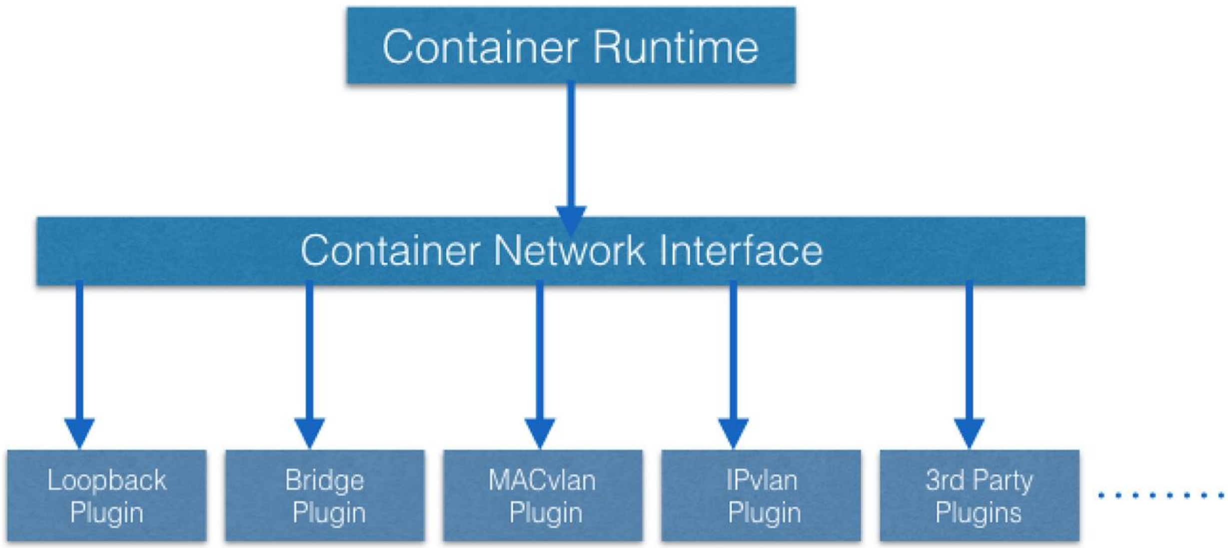
When a Pod is started, a network namespace is created inside the Pod, and all containers running inside the Pod will share that network namespace so that they can talk to each other via localhost.

Pod-to-Pod Communication Across Nodes

In a Kubernetes cluster Pods are scheduled on nodes randomly. Regardless of their host node, Pods are expected to be able to communicate with all other Pods in the cluster, all this without the implementation of Network Address Translation (NAT). This is a fundamental requirement of any networking implementation in Kubernetes.

The Kubernetes network model aims to reduce complexity, and it treats Pods as VMs on a network, where each VM receives an IP address - thus each Pod receiving an IP address. This model is called "**IP-per-Pod**" and ensures Pod-to-Pod communication, just as VMs are able to communicate with each other.

Let's not forget about containers though. They share the Pod's network namespace and must coordinate ports assignment inside the Pod just as applications would on a VM, all while being able to communicate with each other on **localhost** - inside the Pod. However, containers are integrated with the overall Kubernetes networking model through the use of the Container Network Interface (CNI) supported by CNI plugins. CNI is a set of a specification and libraries which allow plugins to configure the networking for containers. While there are a few core plugins, most CNI plugins are 3rd-party Software Defined Networking (SDN) solutions implementing the Kubernetes networking model. In addition to addressing the fundamental requirement of the networking model, some networking solutions offer support for Network Policies. Flannel, Weave, Calico are only a few of the SDN solutions available for Kubernetes clusters.



Container Network Interface (CNI)

The container runtime offloads the IP assignment to CNI, which connects to the underlying configured plugin, such as Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested container runtime.

For more details, you can explore the Kubernetes documentation.

Pod-to-External World Communication

For a successfully deployed containerized applications running in Pods inside a Kubernetes cluster, it requires accessibility from the outside world. Kubernetes enables external accessibility through **services**, complex constructs which encapsulate networking rules definitions on cluster nodes. By exposing services to the external world with **kube-proxy**, applications become accessible from outside the cluster over a virtual IP.

We will have a complete chapter dedicated to this, so we will dive into this later.

Installing Kubernetes

Learning Objectives

By the end of this chapter, you should be able to:

- Discuss the different Kubernetes configuration options.
- Discuss infrastructure considerations before installing Kubernetes.
- Discuss infrastructure choices for a Kubernetes deployment.
- Review Kubernetes installation tools and resources.

Kubernetes Configuration

Kubernetes can be installed using different configurations. The four major installation types are briefly presented below:

- **All-in-One Single-Node Installation**
  In this setup, all the master and worker components are installed and running on a single-node. While it is useful for learning, development, and testing, it should not be used in production. Minikube is one such example, and we are going to explore it in future chapters.
- **Single-Node etcd, Single-Master and Multi-Worker Installation**
  In this setup, we have a single-master node, which also runs a single-node etcd instance. Multiple worker nodes are connected to the master node.
- **Single-Node etcd, Multi-Master and Multi-Worker Installation**
  In this setup, we have multiple-master nodes configured in HA mode, but we have a single-node etcd instance. Multiple worker nodes are connected to the master nodes.
- **Multi-Node etcd, Multi-Master and Multi-Worker Installation**
  In this mode, etcd is configured in clustered HA mode, the master nodes are all configured in HA mode, connecting to multiple worker nodes. This is the most advanced and recommended production setup.

Infrastructure for Kubernetes Installation

Once we decide on the installation type, we also need to make some infrastructure-related decisions, such as:

- Should we set up Kubernetes on bare metal, public cloud, or private cloud?
- Which underlying OS should we use? Should we choose RHEL, CoreOS, CentOS, or something else?
- Which networking solution should we use?
- And so on.

Explore the Kubernetes documentation for details on choosing the right solution. Next, we will take a closer look at these solutions.

Localhost Installation

These are only a few localhost installation options available to deploy single- or multi-node Kubernetes clusters on our workstation/laptop:

- Minikube - single-node local Kubernetes cluster
- Docker Desktop - single-node local Kubernetes cluster for Windows and Mac
- CDK on LXD - multi-node local cluster with LXD containers.

Minikube is the preferred and recommended way to create an all-in-one Kubernetes setup locally. We will be using it extensively in this course.

On-Premise Installation

Kubernetes can be installed on-premise on VMs and bare metal.

- **On-Premise VMs**
  Kubernetes can be installed on VMs created via Vagrant, VMware vSphere, KVM, or another Configuration Management (CM) tool in conjunction with a hypervisor software.

There are different tools available to automate the installation, such as Ansible or kubeadm.

- **On-Premise Bare Metal**
Kubernetes can be installed on on-premise bare metal, on top of different operating systems, like RHEL, CoreOS, CentOS, Fedora, Ubuntu, etc. Most of the tools used to install Kubernetes on VMs can be used with bare metal installations as well.

Cloud Installation

Kubernetes can be installed and managed on almost any cloud environment:

- **Hosted Solutions**
With Hosted Solutions, any given software is completely managed by the provider. The user pays hosting and management charges. Some of the vendors providing hosted solutions for Kubernetes are:

- Google Kubernetes Engine (GKE)
- Azure Kubernetes Service (AKS)
- Amazon Elastic Container Service for Kubernetes (EKS)
- DigitalOcean Kubernetes
- OpenShift Dedicated
- Platform9
- IBM Cloud Kubernetes Service.

- **Turnkey Cloud Solutions**
Below are only a few of the Turnkey Cloud Solutions, to install Kubernetes with just a few commands on an underlying IaaS platform, such as:

- Google Compute Engine (GCE)
- Amazon AWS (AWS EC2)
- Microsoft Azure (AKS).

- **Turnkey On-Premise Solutions**
  The On-Premise Solutions install Kubernetes on secure internal private clouds with just a few commands:

- GKE On-Prem by Google Cloud
- IBM Cloud Private
- OpenShift Container Platform by Red Hat.

Kubernetes Installation Tools/Resources

While discussing installation configuration and the underlying infrastructure, let's take a look at some useful tools/resources available:

- **kubeadm**
  kubeadm is a first-class citizen on the Kubernetes ecosystem. It is a secure and recommended way to bootstrap a single- or multi-node Kubernetes cluster. It has a set of building blocks to setup the cluster, but it is easily extendable to add more features. Please note that kubeadm does not support the provisioning of hosts.
- **kubespray**
  With kubespray (formerly known as kargo), we can install Highly Available Kubernetes clusters on AWS, GCE, Azure, OpenStack, or bare metal. Kubespray is based on Ansible, and is available on most Linux distributions. It is a Kubernetes Incubator project.
- **kops**
  With kops, we can create, destroy, upgrade, and maintain production-grade, highly-available Kubernetes clusters from the command line. It can provision the machines as well. Currently, AWS is officially supported. Support for GCE is in beta, and VMware vSphere in alpha stage, and other platforms are planned for the future. Explore the kops project for more details.
- **kube-aws**
  With kube-aws we can create, upgrade and destroy Kubernetes clusters on AWS from the command line. Kube-aws is also a Kubernetes Incubator project.

If the existing solutions and tools do not fit our requirements, then we can install Kubernetes from scratch (although a dated link from Kubernetes v1.12, it is still a valid solution).

It is worth checking out the *Kubernetes The Hard Way* GitHub project by Kelsey Hightower, which shares the manual steps involved in bootstrapping a Kubernetes cluster.

Minikube – Single Node Cluster

Introduction

As we mentioned in the previous chapter, Minikube is the easiest and most recommended way to run an all-in-one Kubernetes cluster locally on our workstations. In this chapter, we will explore the requirements to install Minikube locally on our workstation, together with the installation instructions to set it up on local Linux, macOS, and Windows operating systems.

Minikube

Learning Objectives

By the end of this chapter, you should be able to:

- Discuss Minikube.
- Install Minikube on local Linux, macOS, and Windows workstation.

- Verify the local installation.

Requirements for Running Minikube

Minikube is installed and runs directly on a local Linux, macOS, or Windows workstation. However, in order to fully take advantage of all the features Minikube has to offer, a Type-2 Hypervisor should be installed on the local workstation, to run in conjunction with Minikube. This does not mean that we need to create any VMs with guest operating systems with this Hypervisor.

Minikube builds all its infrastructure as long as the Type-2 Hypervisor is installed on our workstation. Minikube invokes the Hypervisor to create a single VM which then hosts a single-node Kubernetes cluster. Thus we need to make sure that we have the necessary hardware and software required by Minikube to build its environment. Below we outline the requirements to run Minikube on our local workstation:

- kubectl
  `kubectl` is a binary used to access and manage any Kubernetes cluster. It is installed separately from Minikube. Since we will install **`kubectl`** after the Minikube installation, we may see warnings during the Minikube initialization - safe to disregard for the time being, but do keep in mind that we will have to install **`kubectl`** to be able to manage the Kubernetes cluster. We will explore **`kubectl`** in more detail in future chapters.
- Type-2 Hypervisor
- On Linux VirtualBox or KVM
- On macOS VirtualBox, HyperKit, or VMware Fusion
- On Windows VirtualBox or Hyper-V

  NOTE: Minikube supports a **`--vm-driver=none`** option that runs the Kubernetes components directly on the host OS and not inside a VM. With this option a Docker installation is required and a Linux OS on the local workstation, but no hypervisor installation. If you use **`--vm-driver=none`**, be sure to specify a bridge network for Docker. Otherwise, it might change between network restarts, causing loss of connectivity to your cluster.
- VT-x/AMD-v virtualization must be enabled on the local workstation in BIOS

- Internet connection on first Minikube run - to download packages, dependencies, updates and pull images needed to initialize the Minikube Kubernetes cluster. Subsequent runs will require an internet connection only when new Docker images need to be pulled from a container repository or when deployed containerized applications need it. Once an image has been pulled it can be reused without an internet connection.

In this chapter, we use VirtualBox as hypervisor on all three operating systems - Linux, macOS, and Windows, to allow Minikube to provision the VM which hosts the single-node Kubernetes cluster.

Read more about Minikube from the official Kubernetes documentation or GitHub.

Installing minikube on Linux

Let's learn how to install Minikube v1.0.1 on Ubuntu Linux 18.04 LTS with VirtualBox v6.0 specifically.

NOTE: For other versions, the installation steps may vary! Check the Minikube installation!

Install the VirtualBox hypervisor
Add the source repository for the **bionic** distribution (Ubuntu 18.04), download and register the public key, update and install:

$ sudo bash -c 'echo "deb https://download.virtualbox.org/virtualbox/debian bionic contrib" >> /etc/apt/sources.list'
$ wget -q https://www.virtualbox.org/download/oracle_vbox_2016.asc -O- | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install -y virtualbox-6.0

Install Minikube
We can download the latest release from the Minikube release page. At the time the course was written, the latest Minikube release was v1.0.1. Once downloaded, we need to make it executable and add it to our **`PATH`**:

$ curl -Lo minikube https://storage.googleapis.com/minikube/releases/v1.0.1/minikube-linux-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/

NOTE: Replacing **/v1.0.1/** with **/latest/** will always download the latest version.

Start Minikube
We can start Minikube with the **minikube start** command (disregard "Unable to read.../docker/config..." and "No matching credentials..." warnings):

```
$ minikube start
minikube v1.0.1 on linux (amd64)
Downloading Minikube ISO ...
142.88 MB / 142.88 MB
[============================================] 100.00% 0s
Downloading Kubernetes v1.14.1 images in the background ...
Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
"minikube" IP address is 192.168.99.100
Configuring Docker as the container runtime ...
Version of container runtime is 18.06.3-ce
Waiting for image downloads to complete ...
Preparing Kubernetes environment ...
Downloading kubeadm v1.14.1
Downloading kubelet v1.14.1
Pulling images required by Kubernetes v1.14.1 ...
Launching Kubernetes v1.14.1 using kubeadm ...
Waiting for pods: apiserver proxy etcd scheduler controller dns
Configuring cluster permissions ...
Verifying component health .....
kubectl is now configured to use "minikube"
For best results, install
kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/
Done! Thank you for using minikube!
```

Check the status
With the **minikube status** command, we display the status of Minikube:

```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

Stop minikube
With the **minikube stop** command, we can stop Minikube:

```
$ minikube stop
```

```
Stopping "minikube" in virtualbox ...
"minikube" stopped.
```

Installing Minikube on macOS

Let's learn how to install Minikube v1.0.1 on Mac OS X with VirtualBox v6.0 specifically.

NOTE: For other versions, the installation steps may vary! Check the Minikube installation!

Although VirtualBox is the default hypervisor for Minikube, on Mac OS X we can configure Minikube at startup to use another hypervisor, with the `--vm-driver=xhyve` or `=hyperkit` start option.

Install the VirtualBox hypervisor for OS X hosts
Download and install the `.dmg` package.

Install Minikube
We can download the latest release from the Minikube release page. At the time the course was written, the latest Minikube release was v1.0.1. Once downloaded, we need to make it executable and add it to our `PATH`:

```
$ curl -Lo
minikube https://storage.googleapis.com/minikube/releases/v1.0.1/minikube-darwin-amd64 && chmod +x minikube && sudo mv minikube /usr/local/bin/
```

NOTE: Replacing `/v1.0.1/` with `/latest/` will always download the latest version.

Start Minikube
We can start Minikube with the `minikube start` command (disregard "Unable to read.../docker/config..." and "No matching credentials..." warnings):

```
$ minikube start
minikube v1.0.1 on darwin (amd64)
Downloading Kubernetes v1.14.1 images in the background ...
Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
Downloading Minikube ISO ...
142.88 MB / 142.88 MB
[============================================] 100.00% 0s
"minikube" IP address is 192.168.99.100
```

```
Configuring Docker as the container runtime ...
Version of container runtime is 18.06.3-ce
Waiting for image downloads to complete ...
Preparing Kubernetes environment ...
Downloading kubeadm v1.14.1
Downloading kubelet v1.14.1
Pulling images required by Kubernetes v1.14.1 ...
Launching Kubernetes v1.14.1 using kubeadm ...
Waiting for pods: apiserver proxy etcd scheduler controller dns
Configuring cluster permissions ...
Verifying component health .....
kubectl is now configured to use "minikube"
For best results, install
kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/
Done! Thank you for using minikube!
```

Check the status
With the `minikube status` command, we display the status of Minikube:

```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

Stop minikube
With the `minikube stop` command, we can stop Minikube:

```
$ minikube stop
Stopping "minikube" in virtualbox ...
"minikube" stopped.
```

Installing Minikube on Windows

Let's learn how to install Minikube 1.0.1 on Windows 10 with VirtualBox v6.0.6 specifically.

NOTE: For other versions, the installation steps may vary! Check the Minikube installation!

NOTE: Windows support is currently in experimental phase, and you may encounter issues during installation.

Install the VirtualBox hypervisor for Windows hosts

Download and install the **`.exe`** package.

NOTE: Make sure Hyper-V is disabled (if prior installed and used) while running VirtualBox.

Install Minikube

We can download the latest release from the Minikube release page. At the time the course was written, the latest Minikube release was v1.0.1. Once downloaded, we need to make sure it is added to our **`PATH`**.

There are two **`.exe`** packages available to download for Windows found under Minikube v1.0.1:

- **`minikube-windows-amd64.exe`** which requires to be added to the **`PATH:`** manually
- **`minikube-installer.exe`** which automatically adds the executable to the **`PATH`**.

Download and install the **`minikube-installer.exe`** package found under Minikube v1.0.1.

Start Minikube

We can start Minikube using the **`minikube start`** command (disregard the "Unable to read...docker\\config..." and "No matching credentials..." warnings). Open the PowerShell using the *Run as Administrator* option and execute the following command:

```
PS C:\WINDOWS\system32> minikube start
minikube v1.0.1 on windows (amd64)
Downloading Kubernetes v1.14.1 images in the background ...
Creating virtualbox VM (CPUs=2, Memory=2048MB, Disk=20000MB) ...
Downloading Minikube ISO ...
 0 B / 142.88 MB [---------------------------------------------
-------]   0.00%
 142.88 MB / 142.88 MB
[============================================] 100.00% 0s
"minikube" IP address is 192.168.99.100
Configuring Docker as the container runtime ...
Version of container runtime is 18.06.3-ce
Waiting for image downloads to complete ...
Preparing Kubernetes environment ...
Downloading kubeadm v1.14.1
Downloading kubelet v1.14.1
Pulling images required by Kubernetes v1.14.1 ...
```

```
Launching Kubernetes v1.14.1 using kubeadm ...
Waiting for pods: apiserver proxy etcd scheduler controller dns
Configuring cluster permissions ...
Verifying component health .....
kubectl is now configured to use "minikube"
For best results, install
kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/
Done! Thank you for using minikube!
```

Check the status
We can see the status of Minikube using the **`minikube status`** command. Open the PowerShell using the *Run as Administrator* option and execute the following command:

```
PS C:\WINDOWS\system32> minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

Stop Minikube
We can stop Minikube using the **`minikube stop`** command. Open the PowerShell using the *Run as Administrator* option and execute the following command:

```
PS C:\WINDOWS\system32> minikube stop
Stopping "minikube" in virtualbox ...
"minikube" stopped.
```

Minikube CRI-O

According to the CRI-O website,

"CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes."

Start Minikube with CRI-O as container runtime, instead of Docker, with the following command:

```
$ minikube start --container-runtime=cri-o
minikube v1.0.1 on linux (amd64)
Downloading Kubernetes v1.14.1 images in the background ...
Tip: Use 'minikube start -p <name>' to create a new cluster, or 'minikube delete' to
```

delete this one.
Restarting existing virtualbox VM for "minikube" ...
Waiting for SSH access ...
"minikube" IP address is 192.168.99.100
Configuring CRI-O as the container runtime ...
Version of container runtime is 1.13.5
Waiting for image downloads to complete ...
Preparing Kubernetes environment ...
Pulling images required by Kubernetes v1.14.1 ...
Relaunching Kubernetes v1.14.1 using kubeadm ...
Waiting for pods: apiserver etcd scheduler controller
Updating kube-proxy configuration ...
Verifying component health ......
kubectl is now configured to use "minikube"
For best results, install kubectl: https://kubernetes.io/docs/tasks/tools/install-kubectl/
Done! Thank you for using minikube!

Let's login via ssh into the Minikube's VM:

```
$ minikube ssh
```

```
          _             _
    _         _ ( )           ( )
 ___ ___  (_)  ___  (_)| |/')  _   _ | |_      __
/' _ ` _ `\| |/' _ `\| || , <  ( ) ( )| '_`\  /'__`\
| ( ) ( ) || || ( ) || || |\`\ | (_) || |_) )(  ___/
(_) (_) (_)(_)(_) (_)(_)(_) (_)`\___/'(_,__/'`\____)

$ _
```

NOTE: If you try to list containers using the `docker` command, it will not produce any results, because Docker is not running containers:

```
$ sudo docker container ls
```

```
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker
daemon running?
```

List the containers created via CRI-O container runtime with the following command:

```
$ sudo runc list
ID
   PID          STATUS       BUNDLE

                 CREATED                          OWNER
1090869caeea44cb179d31b70ba5b6de96f10a8a5f4286536af5dac1c4312030  366
1    running   /run/containers/storage/overlay-
containers/1090869caeea44cb179d31b70ba5b6de96f10a8a5f4286536af5dac1c43
12030/userdata  2019-04-18T20:03:02.199284303Z  root
1e9f8dce6d535b67822e744204098060ff92e574780a1809adbda48ad8605d06
   3614        running     /run/containers/storage/overlay-
containers/1e9f8dce6d535b67822e744204098060ff92e574780a1809adbda
48ad8605d06/userdata   2019-04-18T20:03:02.129881761Z   root
1edcfc78bca52be153cc9f525d9fc64be75ccea478897004a5032f37c6c4c9dc
   3812        running     /run/containers/storage/overlay-
containers/1edcfc78bca52be153cc9f525d9fc64be75ccea478897004a5032
f37c6c4c9dc/userdata   2019-04-18T20:03:02.740669541Z   root
...
```

Installing Minikube (Demo)

LinuxFoundationXLFS
158x-V000100_DTH.r

Ch 6 - Installing
Minikube-en.txt

Accessing Minikube

Any healthy running Kubernetes cluster can be accessed via any one of the following methods:

- **Command Line Interface (CLI) tools and scripts**

- **Web-based User Interface (Web UI) from a web browser**
- **APIs from CLI or programmatically**

These methods are applicable to all Kubernetes clusters.

Accessing Minikube: Command Line Interface (CLI)

`kubectl` is the **Kubernetes Command Line Interface (CLI) client** to manage cluster resources and applications. It can be used standalone, or part of scripts and automation tools. Once all required credentials and cluster access points have been configured for **`kubectl`** it can be used remotely from anywhere to access a cluster.

In later chapters, we will be using **`kubectl`** to deploy applications, manage and configure Kubernetes resources.

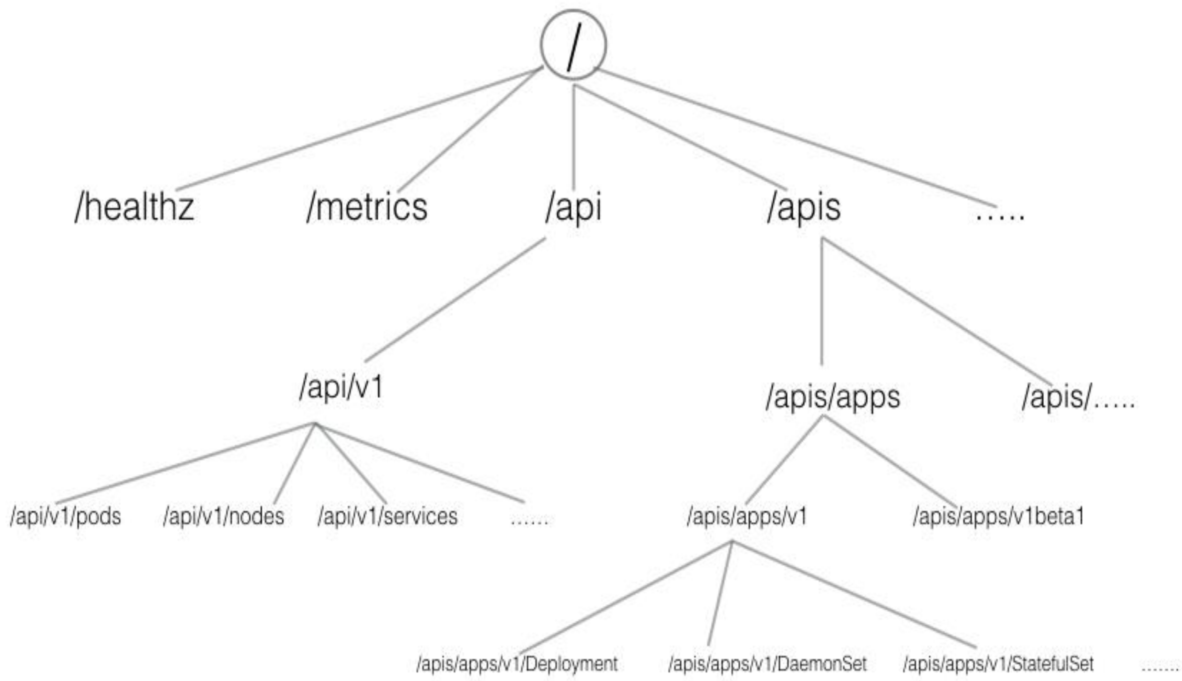
Accessing Minikube: Web-based User Interface (Web UI)

The Kubernetes Dashboard provides a **Web-Based User Interface (Web UI)** to interact with a Kubernetes cluster to manage resources and containerized applications. In one of the later chapters, we will be using it to deploy a containerized application.

Accessing Minikube: APIs

As we know, Kubernetes has the **API server**, and operators/users connect to it from the external world to interact with the cluster. Using both CLI and Web UI, we can connect to the API server running on the master node to perform different operations. We can directly connect to the API server using its API endpoints and send commands to it, as long as we can access the master node and have the right credentials.

Below, we can see a part of the HTTP API space of Kubernetes:



HTTP API Space of Kubernetes

HTTP API space of Kubernetes can be divided into three independent groups:

- **Core Group (`/api/v1`)**
  This group includes objects such as Pods, Services, nodes, namespaces, configmaps, secrets, etc.
- **Named Group**
  This group includes objects in **`/apis/$NAME/$VERSION`** format. These different API versions imply different levels of stability and support:
  Alpha level - it may be dropped at any point in time, without notice. For example, **`/apis/batch/v2alpha1`**.
  Beta level - it is well-tested, but the semantics of objects may change in incompatible ways in a subsequent beta or stable release. For example, **`/apis/certificates.k8s.io/v1beta1`**.
  Stable level - appears in released software for many subsequent versions. For example, **`/apis/networking.k8s.io/v1`**.

- **System-wide**
  This group consists of system-wide API endpoints, like **`/healthz`**, **`/logs`**, **`/metrics`**, **`/ui`**, etc.

  We can either connect to an API server directly via calling the respective API endpoints or via the CLI/Web UI.

  Next, we will see how we can access the Minikube environment we set up in the previous chapter.

kubectl

`kubectl` is generally installed before installing Minikube, but we can also install it after. Once installed, **`kubectl`** receives its configuration automatically for Minikube Kubernetes cluster access. However, in other Kubernetes cluster setups, we may need to configure the cluster access points and certificates required by **`kubectl`** to access the cluster.

There are different methods that can be used to install **`kubectl`**, which are mentioned in the Kubernetes documentation. For best results, it is recommended to keep **`kubectl`** at the same version with the Kubernetes run by Minikube - at the time the course was written the latest stable release was **v1.14.1**. Next, we will look at a few steps to install it on Linux, macOS, and Windows systems.

Installing kubectl on Linux

To install **`kubectl`** on Linux, follow the instruction below:

Download the latest stable `kubectl` binary, make it executable and move it to the `PATH`:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

NOTE: To download and setup a specific version of **`kubectl`** (such as v1.14.1), issue the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.14.1/bin/linux/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

Installing kubectl on macOS

There are two ways to install **kubectl** on macOS: manually and using the Homebrew package manager. Next, we will provide instructions for both methods.

To manually install `kubectl`, download the latest stable kubectl binary, make it executable and move it to the `PATH` with the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/darwin/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

NOTE: To download and setup a specific version of **kubectl** (such as v1.14.1), issue the following command:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.14.1/bin/darwin/amd64/kubectl && chmod +x kubectl && sudo mv kubectl /usr/local/bin/
```

To install `kubectl` with Homebrew package manager, issue the following command:

```
$ brew install kubernetes-cli
```

Installing kubectl on Windows

To install **kubectl**, we can download the binary directly or use **curl** from the CLI. Once downloaded the binary needs to be added to the **PATH**.

Direct download link for v1.14.1 binary (just click below):

https://storage.googleapis.com/kubernetes-release/release/v1.14.1/bin/windows/amd64/kubectl.exe

NOTE: Obtain the latest **kubectl** stable release version number from the link below, and if needed, edit the download link for the binary from above:

https://storage.googleapis.com/kubernetes-release/release/stable.txt

Use the `curl` command (if installed) from the CLI:

```
$ curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.14.1/bin/windows/amd64/kubectl.exe
```

Once downloaded, move the `kubectl` binary to the `PATH`.

kubectl Configuration File

To access the Kubernetes cluster, the **`kubectl`** client needs the master node endpoint and appropriate credentials to be able to interact with the API server running on the master node. While starting Minikube, the startup process creates, by default, a configuration file, **`config`**, inside the **`.kube`** directory (often referred to as the **`dot-kube-config`** file), which resides in the user's **`home`** directory. The configuration file has all the connection details required by **`kubectl`**. By default, the **`kubectl`** binary parses this file to find the master node's connection endpoint, along with credentials. To look at the connection details, we can either see the content of the **`~/.kube/config`** file (on Linux) or run the following command:

```
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/client.crt
    client-key: /home/student/.minikube/client.key
```

Once **kubectl** is installed, we can get information about the Minikube cluster with the **kubectl cluster-info** command:

```
$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
KubeDNS is running at https://192.168.99.100:8443//api/v1/namespaces/kube-system/services/kube-dns:dns/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
```

You can find more details about the **kubectl** command line options here.

Although for the Kubernetes cluster installed by Minikube the **~/.kube/config** file gets created automatically, this is not the case for Kubernetes clusters installed by other tools. In other cases, the config file has to be created manually and sometimes re-configured to suit various networking and client/server setups.

Installing kubectl CLI Client (Demo)

Installing kubectl CLI Client

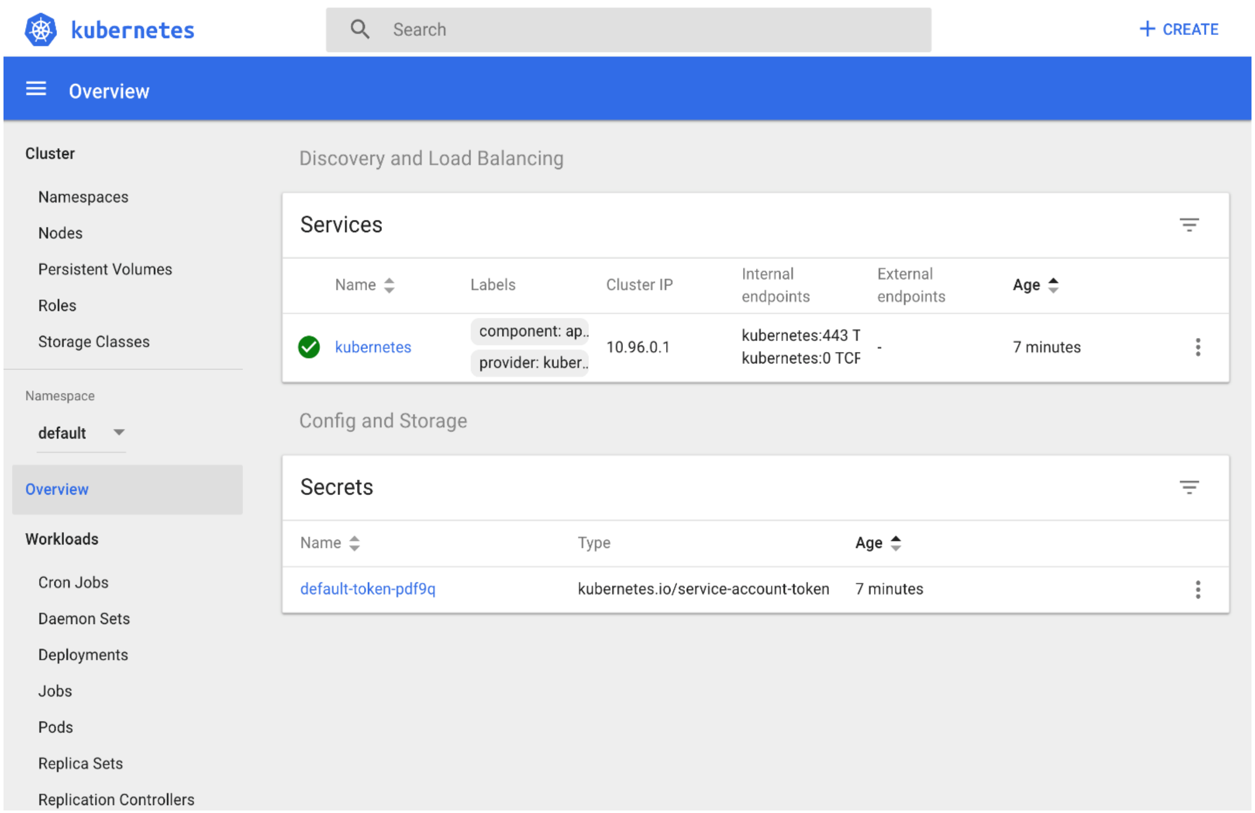
LinuxFoundationXLFS
158x-V000200_DTH.r

Ch 7 - Installing
kubectl CLI client-en.t

Kubernetes Dashboard

As mentioned earlier, the Kubernetes Dashboard provides a web-based user interface for Kubernetes cluster management. To access the dashboard from Minikube, we can use the **minikube dashboard** command, which opens a new tab on our web browser, displaying the Kubernetes Dashboard:

```
$ minikube dashboard
```



Kubernetes Dashboard

NOTE: In case the browser is not opening another tab and does not display the Dashboard as expected, verify the output in your terminal as it may display a link for the Dashboard (together with some Error messages). Copy and paste that link in a new tab of your browser. Depending on your terminal's features you may be able to just click or right-click the link to open directly in the browser. The link may look similar to:

`http://127.0.0.1:37751/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/`

Chances are that the only difference is the PORT number, which above is 37751. Your port number may be different.

After a logout/login or a reboot of your workstation the normal behavior should be expected (where the **`minikube dashboard`** command directly opens a new tab in your browser displaying the Dashboard).

The 'kubectl proxy' Command

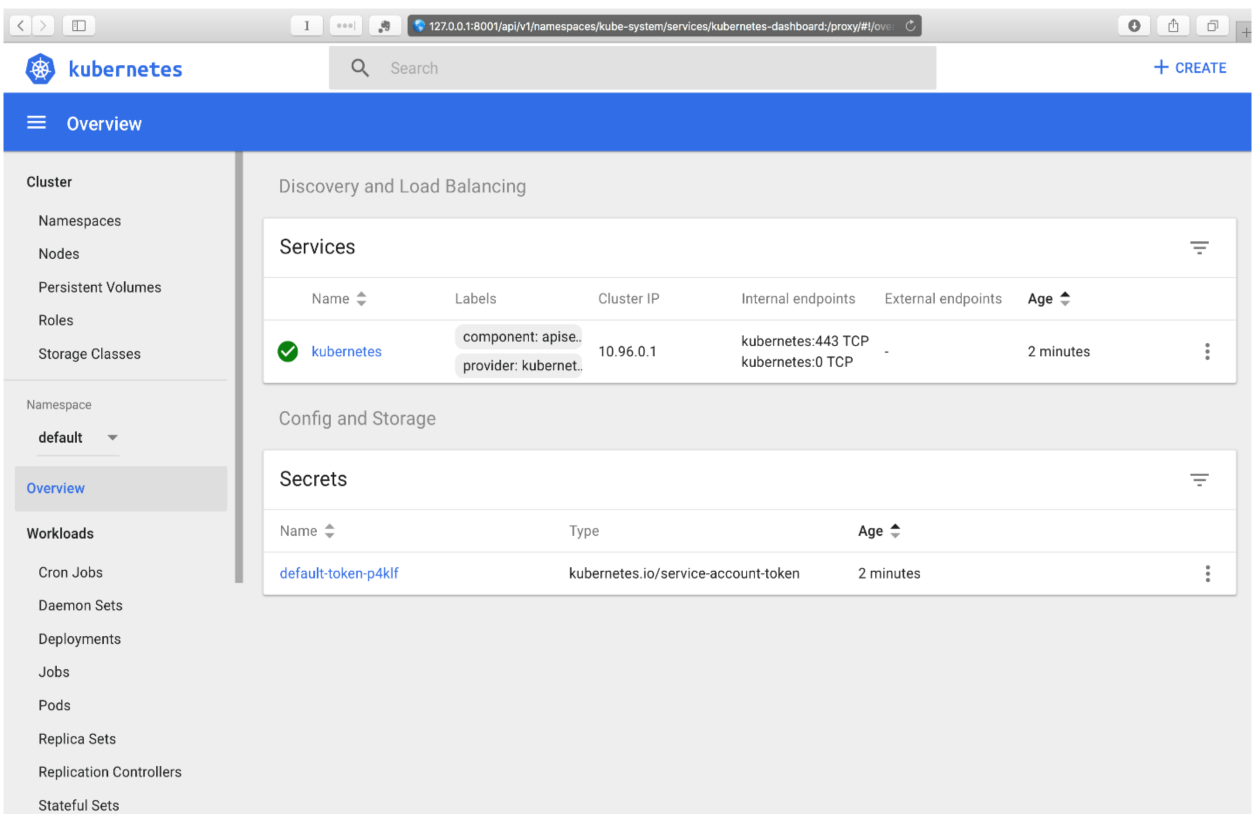
Issuing the **`kubectl proxy`** command, **`kubectl`** authenticates with the API server on the master node and makes the Dashboard available on a slightly different URL than the one earlier, this time through the proxy port 8001.

First, we issue the **`kubectl proxy`** command:

```
$ kubectl proxy
Starting to serve on 127.0.0.1:8001
```

It locks the terminal for as long as the proxy is running. With the **proxy** running we can access the **Dashboard** over the new URL (just click on it below - it should work on your workstation). Once we stop the proxy (with CTRL + C) the Dashboard is no longer accessible.

http://127.0.0.1:8001/api/v1/namespaces/kube-system/services/kubernetes-dashboard:/proxy/#!/overview?namespace=default



Kubernetes Dashboard over the proxy

APIs - with 'kubectl proxy'

When **`kubectl proxy`** is running, we can send requests to the API over the **`localhost`** on the proxy port 8001 (from another terminal, since the proxy locks the first terminal):

```
$ curl http://localhost:8001/
{
 "paths": [
   "/api",
   "/api/v1",
   "/apis",
   "/apis/apps",
   ......
   ......
   "/logs",
   "/metrics",
   "/openapi/v2",
   "/version"
 ]
}
```

With the above **`curl`** request, we requested all the API endpoints from the API server. Clicking on the link above (in the curl command), it will open the same listing output in a browser tab.

We can explore every single path combination with **`curl`** or in a browser, such as:

http://localhost:8001/api/v1

http://localhost:8001/apis/apps/v1

http://localhost:8001/healthz

http://localhost:8001/metrics

APIs - without 'kubectl proxy'

When not using the **`kubectl proxy`**, we need to authenticate to the API server when sending API requests. We can authenticate by providing a **Bearer Token** when issuing a **`curl`**, or by providing a set of **keys** and **certificates**.

A **Bearer Token** is an **access token** which is generated by the authentication server (the API server on the master node) and given back to the client. Using that token, the client can connect back to the Kubernetes API server without providing further authentication details, and then, access resources.

Get the token:

```
$ TOKEN=$(kubectl describe secret -n kube-system $(kubectl get
secrets -n kube-system | grep default | cut -f1 -d ' ') | grep -
E '^token' | cut -f2 -d':' | tr -d '\t' | tr -d " ")
```

Get the API server endpoint:

```
$ APISERVER=$(kubectl config view | grep https | cut -f 2- -d
":" | tr -d " ")
```

Confirm that the `APISERVER` stored the same IP as the Kubernetes master IP by issuing the following 2 commands and comparing their outputs:

```
$ echo $APISERVER
https://192.168.99.100:8443

$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443 ...
```

Access the API server using the `curl` command, as shown below:

```
$ curl $APISERVER --header "Authorization: Bearer $TOKEN" --
insecure
{
 "paths": [
   "/api",
   "/api/v1",
   "/apis",
   "/apis/apps",
   ......
   ......
   "/logs",
   "/metrics",
   "/openapi/v2",
   "/version"
 ]
}
```

Instead of the **`access token`**, we can extract the client certificate, client key, and certificate authority data from the **`.kube/config`** file. Once extracted, they are encoded and then passed with a **`curl`** command for authentication. The new **`curl`** command looks similar to:

```
$ curl $APISERVER --cert encoded-cert --key encoded-key --cacert encoded-ca
```

Accessing the Cluster with Dashboard and Query APIs with CLI (Demo)

LinuxFoundationXLFS158x-V000300_DTH.r

Ch 7 - v02 - Accessing the cluster

Kubernetes Building Blocks

Introduction

In this chapter, we will explore the **Kubernetes object model** and discuss some of its fundamental building blocks, such as **Pods**, **ReplicaSets**, **Deployments**, **Namespaces**, etc. We will also discuss the essential role **Labels** and Selectors play in a microservices driven architecture as they group decoupled objects together.

Learning Objectives

By the end of this chapter, you should be able to:

- Review the Kubernetes object model.
- Discuss Kubernetes building blocks, e.g. Pods, ReplicaSets, Deployments, Namespaces.
- Discuss Labels and Selectors.

Kubernetes Object Model

Kubernetes has a very rich object model, representing different persistent entities in the Kubernetes cluster. Those entities describe:

- What containerized applications we are running and on which node
- Application resource consumption
- Different policies attached to applications, like restart/upgrade policies, fault tolerance, etc.

With each object, we declare our intent **`spec`** section. The Kubernetes system manages the **`status`** section for objects, where it records the actual state of the object. At any given point in time, the Kubernetes Control Plane tries to match the object's actual state to the object's desired state.

Examples of Kubernetes objects are Pods, ReplicaSets, Deployments, Namespaces, etc. We will explore them next.

When creating an object, the object's configuration data section from below the **`spec`** field has to be submitted to the Kubernetes API server. The **`spec`** section describes the desired state, along with some basic information, such as the object's name. The API request to create an object must have the **`spec`** section, as well as other details. Although the API server accepts object definition files in a JSON format, most often we provide such files in a YAML format which is converted by **`kubectl`** in a JSON payload and sent to the API server.

Below is an example of a Deployment object's configuration in YAML format:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
```

```
    spec:
      containers:
      - name: nginx
        image: nginx:1.15.11
        ports:
        - containerPort: 80
```

The **`apiVersion`** field is the first required field, and it specifies the API endpoint on the API server which we want to connect to; it must match an existing version for the object type defined. The second required field is **`kind`**, specifying the object type - in our case it is **`Deployment`**, but it can be Pod, Replicaset, Namespace, Service, etc. The third required field **`metadata`**, holds the object's basic information, such as name, labels, namespace, etc. Our example shows two **`spec`** fields (**`spec`** and **`spec.template.spec`**). The fourth required field **`spec`** marks the beginning of the block defining the desired state of the Deployment object. In our example, we want to make sure that 3 Pods are running at any given time. The Pods are created using the Pods Template defined in **`spec.template`**. A nested object, such as the Pod being part of a Deployment, retains its **`metadata`** and **`spec`** and loses the **`apiVersion`** and **`kind`** - both being replaced by **`template`**.
In **`spec.template.spec`**, we define the desired state of the Pod. Our Pod creates a single container running the **`nginx:1.15.11`** image from Docker Hub.

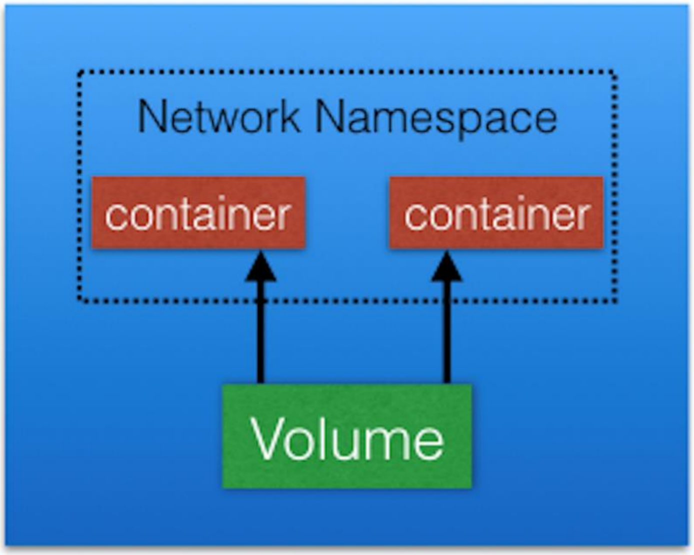
Once the Deployment object is created, the Kubernetes system attaches the **`status`** field to the object; we will explore it later.

Next, we will take a closer look at some of the Kubernetes objects, along with other building blocks.

Pods

A Pod is the smallest and simplest Kubernetes object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, which:

- Are scheduled together on the same host with the Pod
- Share the same network namespace
- Have access to mount the same external storage (volumes).



Pods

Pods are ephemeral in nature, and they do not have the capability to self-heal by themselves. That is the reason they are used with controllers which handle Pods' replication, fault tolerance, self-healing, etc. Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc. We attach a nested Pod's specification to a controller object using the Pod Template, as we have seen in the previous section.

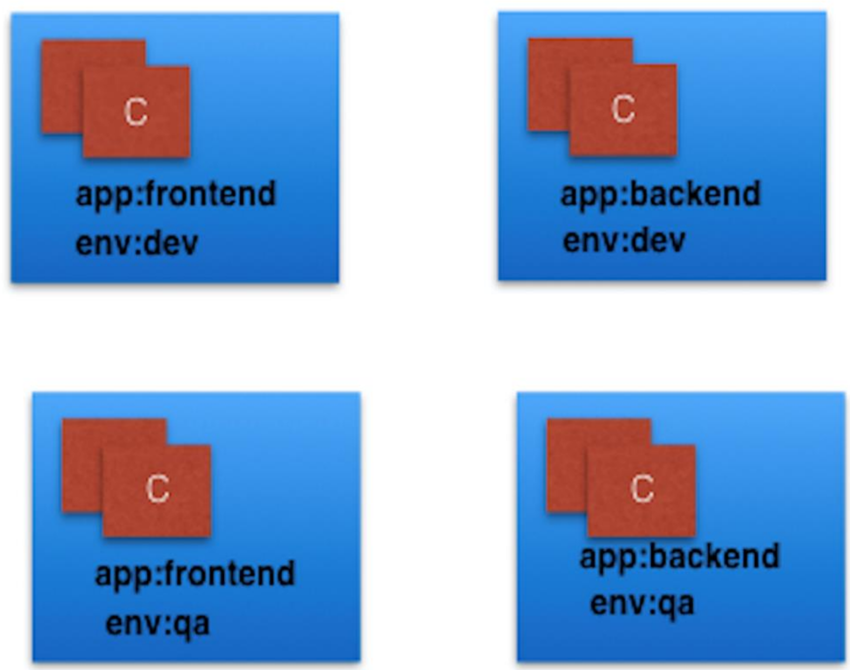
Below is an example of a Pod object's configuration in **YAML** format:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.15.11
    ports:
    - containerPort: 80
```

The **apiVersion** field must specify **v1** for the **Pod** object definition. The second required field is **kind** specifying the **Pod** object type. The third required field **metadata**, holds the object's name and label. The fourth required field **spec** marks the beginning of the block defining the desired state of the Pod object - also named the **PodSpec**. Our Pod creates a single container running the **nginx:1.15.11** image from Docker Hub

Labels

[Labels](#) are key-value pairs attached to Kubernetes objects (e.g. Pods, ReplicaSets). Labels are used to organize and select a subset of objects, based on the requirements in place. Many objects can have the same Label(s). Labels do not provide uniqueness to objects. Controllers use Labels to logically group together decoupled objects, rather than using objects' names or IDs.



Labels

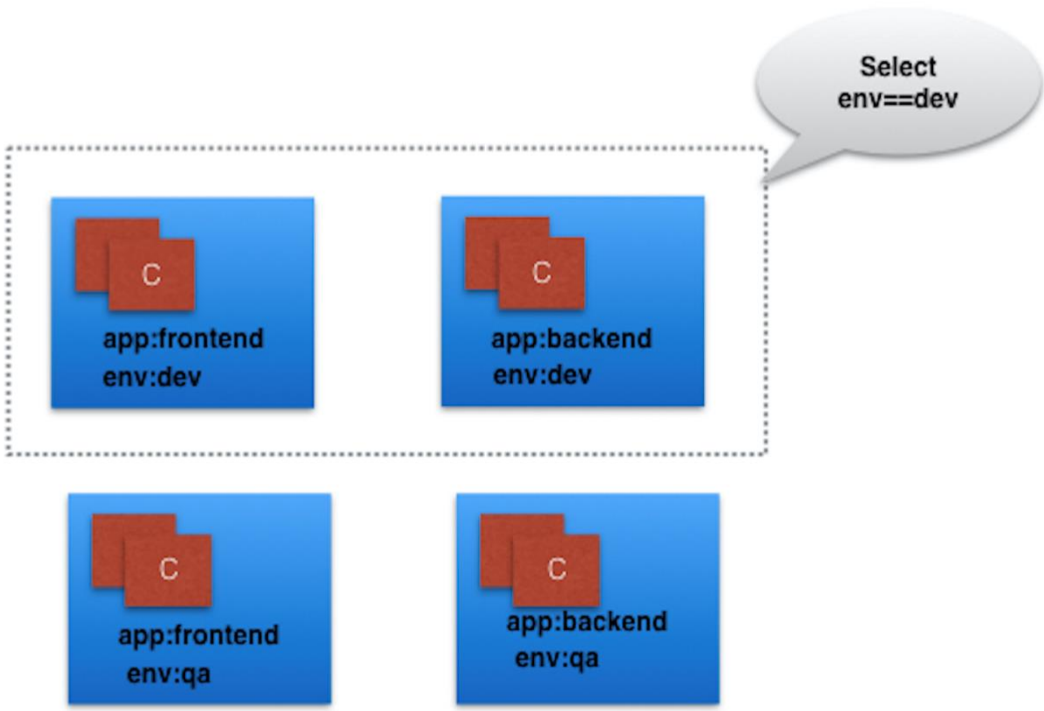
In the image above, we have used two Label keys: **`app`** and **`env`**. Based on our requirements, we have given different values to our four Pods. The Label **`env=dev`** logically selects and groups the top two Pods, while the Label **`app=frontend`** logically selects and groups the left two Pods. We can select one of the four Pods - bottom left, by selecting two Labels: **`app=frontend`** and **`env=qa`**.

Label Selectors

Controllers use [Label Selectors](#) to select a subset of objects. Kubernetes supports two types of Selectors:

- **Equality-Based Selectors**
  Equality-Based Selectors allow filtering of objects based on Label keys and values. Matching is achieved using the **=**, **==** (equals, used interchangeably), or **!=** (not equals) operators. For example, with **env==dev** or **env=dev** we are selecting the objects where the **env** Label key is set to value **dev**.

- **Set-Based Selectors**
  Set-Based Selectors allow filtering of objects based on a set of values. We can use **in**, **notin** operators for Label values, and **exist/does not exist** operators for Label keys. For example, with **env in (dev,qa)** we are selecting objects where the **env** Label is set to either **dev** or **qa**; with **!app** we select objects with no Label key **app**.



Selectors

ReplicationControllers

Although no longer a recommended method, a ReplicationController is a controller that ensures a specified number of replicas of a Pod is running at any given time. If there are more Pods than the desired count, a replication controller would terminate the extra

Pods, and, if there are fewer Pods, then the replication controller would create more Pods to match the desired count. Generally, we don't deploy a Pod independently, as it would not be able to re-start itself if terminated in error. The recommended method is to use some type of replication controllers to create and manage Pods.

The default controller is a Deployment which configures a ReplicaSet to manage Pods' lifecycle.

ReplicationControllers

Although no longer a recommended method, a ReplicationController is a controller that ensures a specified number of replicas of a Pod is running at any given time. If there are more Pods than the desired count, a replication controller would terminate the extra Pods, and, if there are fewer Pods, then the replication controller would create more Pods to match the desired count. Generally, we don't deploy a Pod independently, as it would not be able to re-start itself if terminated in error. The recommended method is to use some type of replication controllers to create and manage Pods.

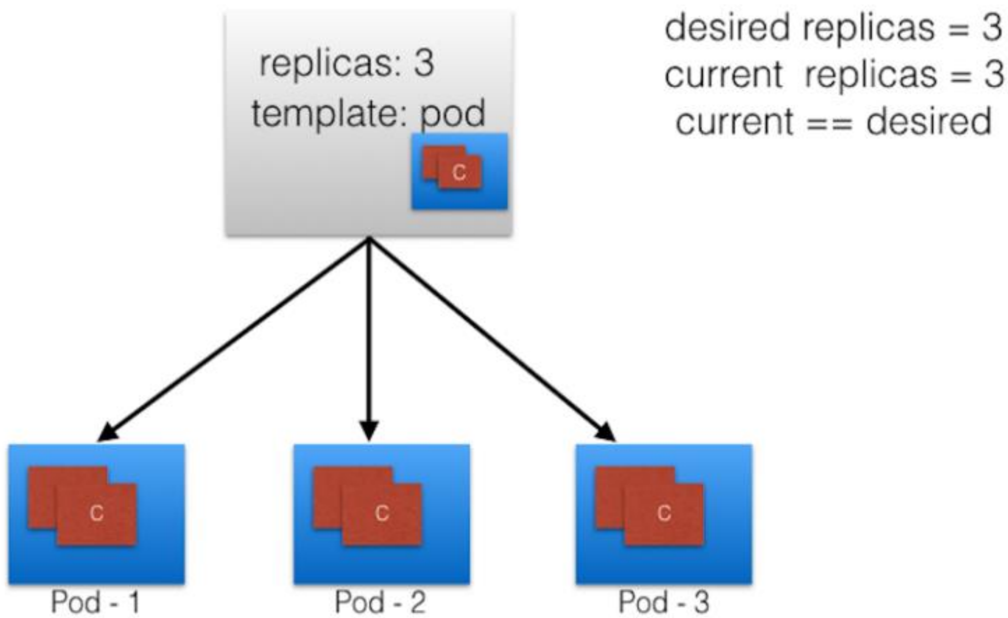
The default controller is a Deployment which configures a ReplicaSet to manage Pods' lifecycle.

ReplicaSets I

A ReplicaSet is the next-generation ReplicationController. ReplicaSets support both equality- and set-based selectors, whereas ReplicationControllers only support equality-based Selectors. Currently, this is the only difference.

With the help of the ReplicaSet, we can scale the number of Pods running a specific container application image. Scaling can be accomplished manually or through the use of an autoscaler.

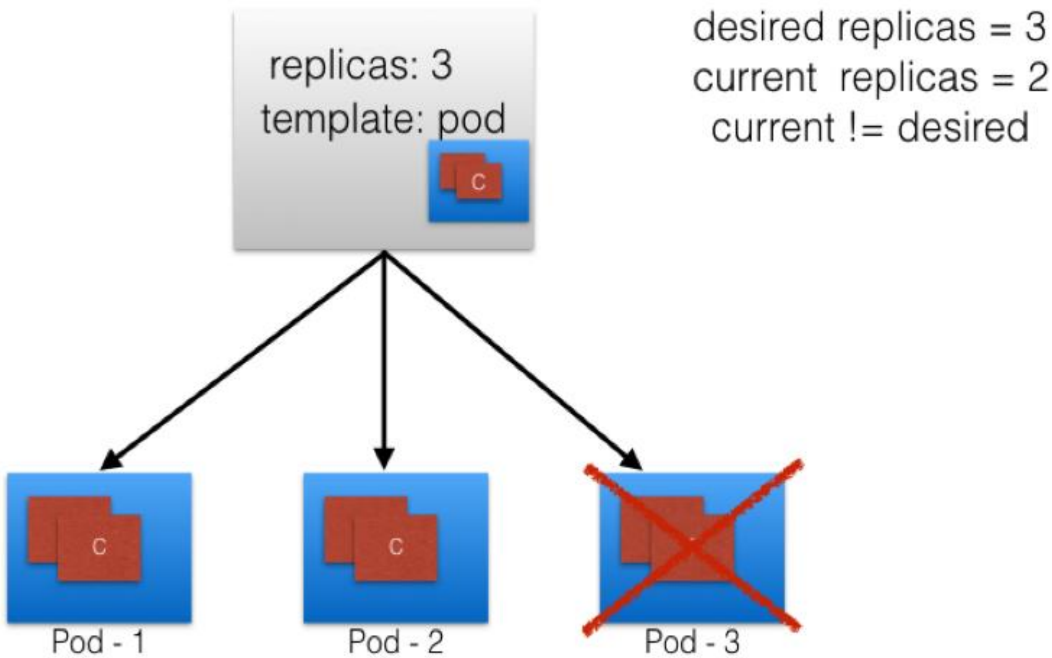
Next, you can see a graphical representation of a ReplicaSet, where we have set the replica count to 3 for a Pod.



ReplicaSet (Current State and Desired State Are the Same)

ReplicaSets II

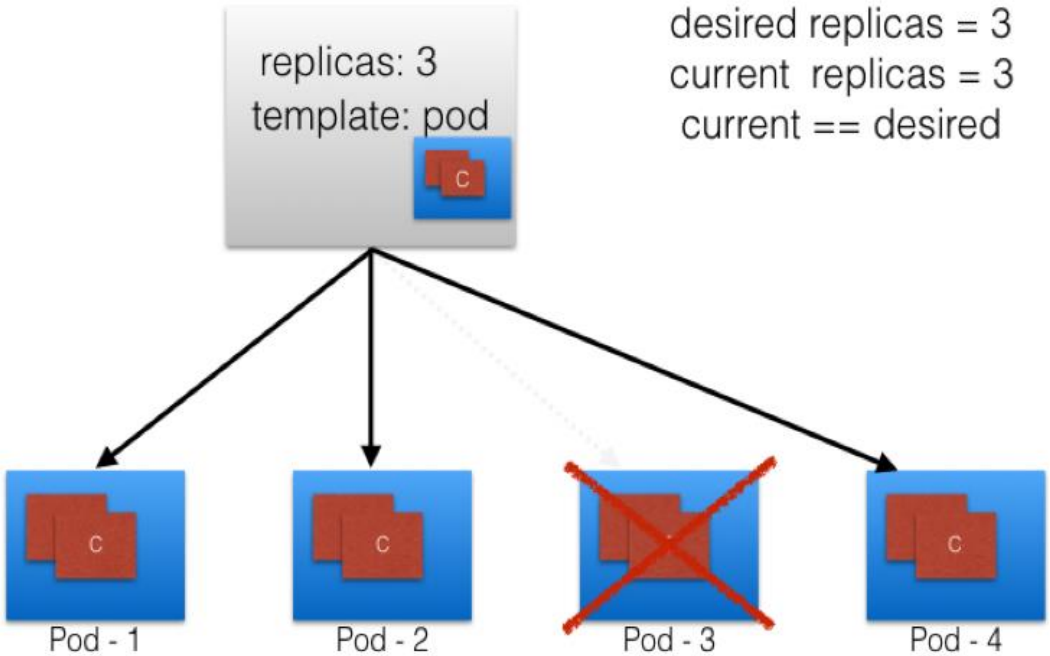
Now, let's continue with the same example and assume that one of the Pods is forced to terminate (due to insufficient resources, timeout, etc.), and the current state is no longer matching the desired state.



ReplicaSet (Current State and Desired State Are Different)

ReplicaSets III

The ReplicaSet will detect that the current state is no longer matching the desired state. The ReplicaSet will create an additional Pod, thus ensuring that the current state matches the desired state.



ReplicaSet (Creating a Pod to Match Current and Desired States)

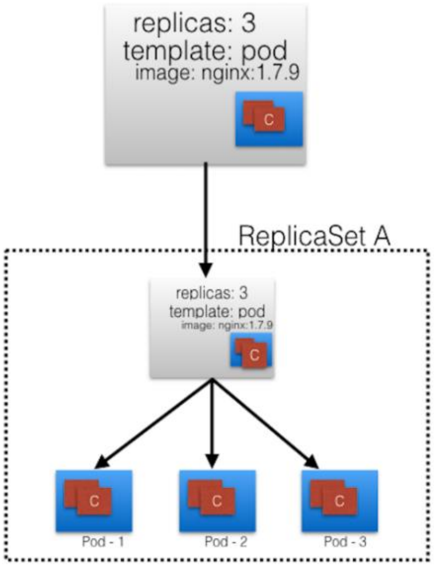
ReplicaSets can be used independently as Pod controllers but they only offer a limited set of features. A set of complementary features are provided by Deployments, the recommended controllers for the orchestration of Pods. Deployments manage the creation, deletion, and updates of Pods. A Deployment automatically creates a ReplicaSet, which then creates a Pod. There is no need to manage ReplicaSets and Pods separately, the Deployment will manage them on our behalf.

We will take a closer look at Deployments next.

Deployments I

Deployment objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the master node's controller manager, and it ensures that the current state always matches the desired state. It allows for seamless application updates and downgrades through **`rollouts`** and **`rollbacks`**, and it directly manages its ReplicaSets for application scaling.

In the following example, a new **Deployment** creates **ReplicaSet A** which then creates **3 Pods**, with each Pod Template configured to run one **nginx:1.7.9** container image. In this case, the **ReplicaSet A** is associated with **nginx:1.7.9** representing a state of the **Deployment**. This particular state is recorded as **Revision 1**.



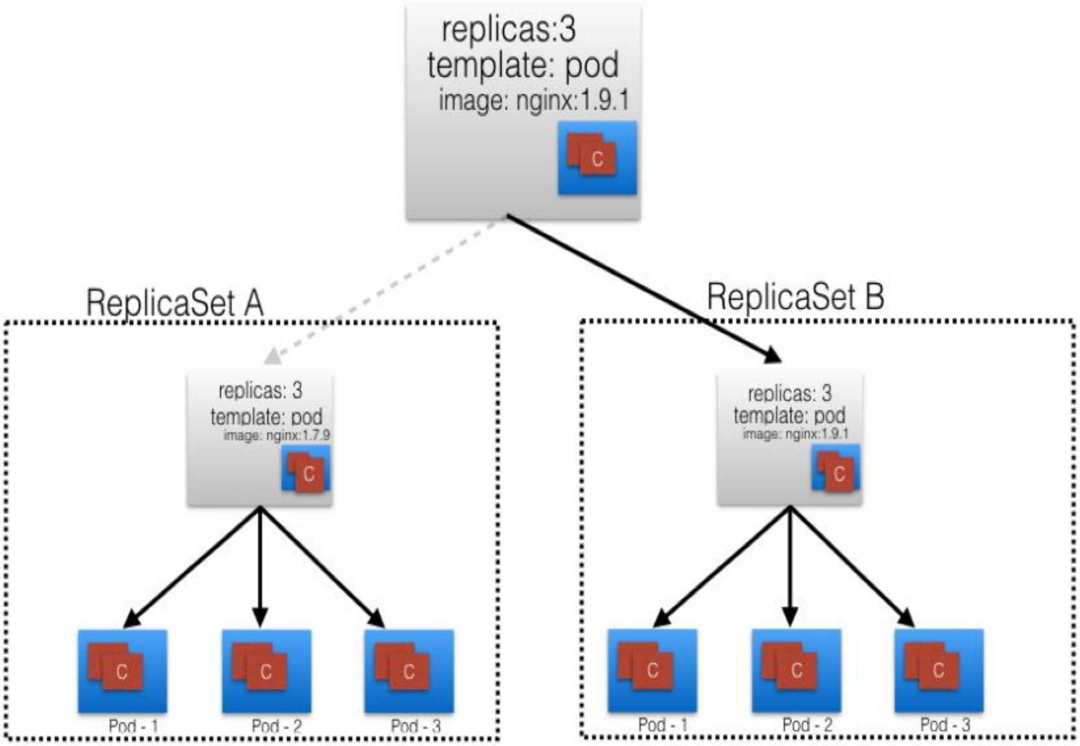
Deployment (ReplicaSet A Created)

Deployments II

Now, in the Deployment, we change the Pods' Template and we update the container image from **nginx:1.7.9** to **nginx:1.9.1**. The **Deployment** triggers a new **ReplicaSet B** for the new container image versioned **1.9.1** and this association represents a new recorded state of the **Deployment**, **Revision 2**. The seamless transition between the two ReplicaSets, from **ReplicaSet A** with 3 Pods versioned **1.7.9** to the new **ReplicaSet B** with 3 new Pods versioned **1.9.1**, or from **Revision 1** to **Revision 2**, is a Deployment **rolling update**.

A **rolling update** is triggered when we update the Pods Template for a deployment. Operations like scaling or labeling the deployment do not trigger a rolling update, thus do not change the Revision number.

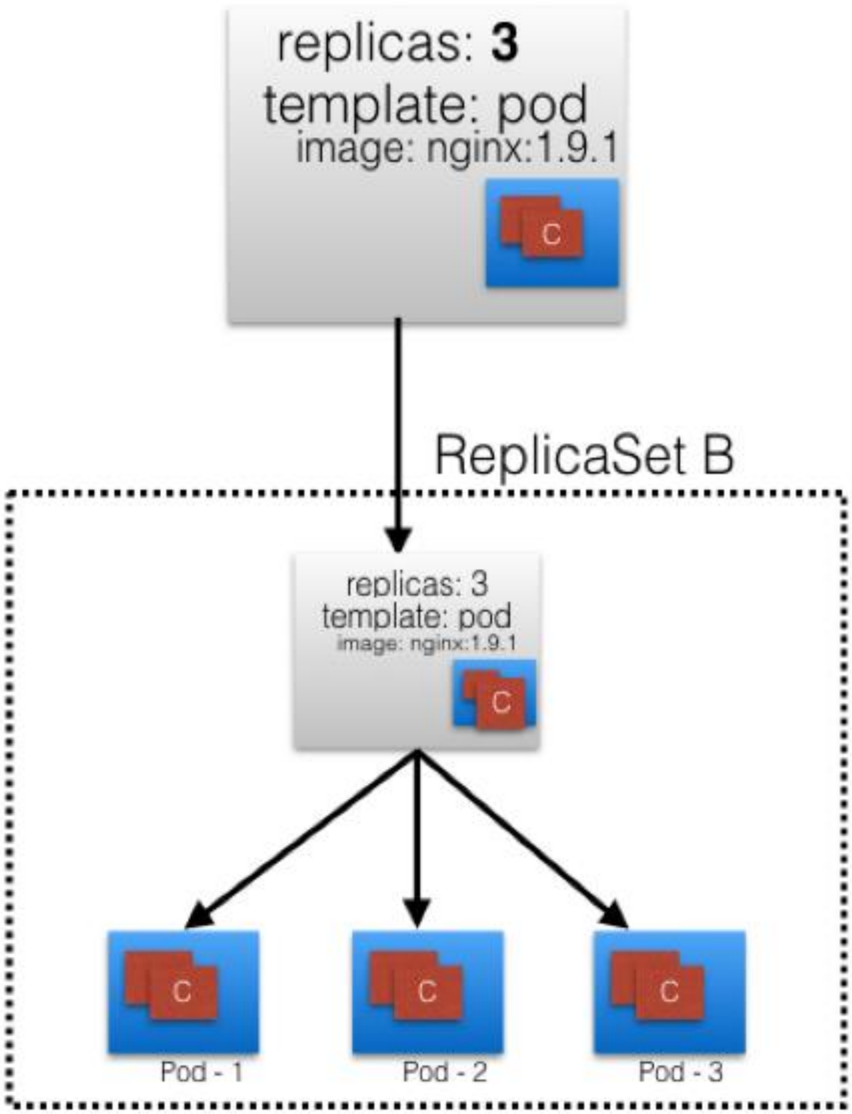
Once the rolling update has completed, the **Deployment** will show both **ReplicaSets A** and **B**, where **A** is scaled to 0 (zero) Pods, and **B** is scaled to 3 Pods. This is how the Deployment records its prior state configuration settings, as **Revisions**.



Deployment (ReplicaSet B Created)

Deployments III

Once **`ReplicaSet B`** and its **`3 Pods`** versioned **`1.9.1`** are ready, the **`Deployment`** starts actively managing them. However, the Deployment keeps its prior configuration states saved as Revisions which play a key factor in the **`rollback`** capability of the Deployment - returning to a prior known configuration state. In our example, if the performance of the new **`nginx:1.9.1`** is not satisfactory, the Deployment can be rolled back to a prior Revision, in this case from **`Revision 2`** back to **`Revision 1`** running **`nginx:1.7.9`**.



Deployment Points to ReplicaSet B

Namespaces

If multiple users and teams use the same Kubernetes cluster we can partition the cluster into virtual sub-clusters using [Namespaces](). The names of the resources/objects created inside a Namespace are unique, but not across Namespaces in the cluster.

To list all the Namespaces, we can run the following command:

```
$ kubectl get namespaces
NAME              STATUS       AGE
default           Active       11h
kube-node-lease   Active       11h
kube-public       Active       11h
kube-system       Active       11h
```

Generally, Kubernetes creates four default Namespaces: **`kube-system`**, **`kube-public`**, **`kube-node-lease`**, and **`default`.** The **`kube-system`** Namespace contains the objects created by the Kubernetes system, mostly the control plane agents. The **`default`** Namespace contains the objects and resources created by administrators and developers. By default, we connect to the **`default`** Namespace. **`kube-public`** is a special Namespace, which is unsecured and readable by anyone, used for special purposes such as exposing public (non-sensitive) information about the cluster. The newest Namespace is **`kube-node-lease`** which holds node lease objects used for node heartbeat data. Good practice, however, is to create more Namespaces to virtualize the cluster for users and developer teams.

With Resource Quotas, we can divide the cluster resources within Namespaces. We will briefly cover **resource quotas** in one of the future chapters.

Video text

LinuxFoundationXLFS158x-V000800_DTH.r

Ch 8-v01 - Deployment Rolling U

Ch 8-v01 - Deployment Rolling U

Chapter 9. Authentication, Authorization, Admission Control

Every API request reaching the API server has to go through three different stages before being accepted by the server and acted upon. In this chapter, we will be looking into the Authentication, Authorization and Admission Control stages of Kubernetes API requests.

Learning Objectives

By the end of this chapter, you should be able to:

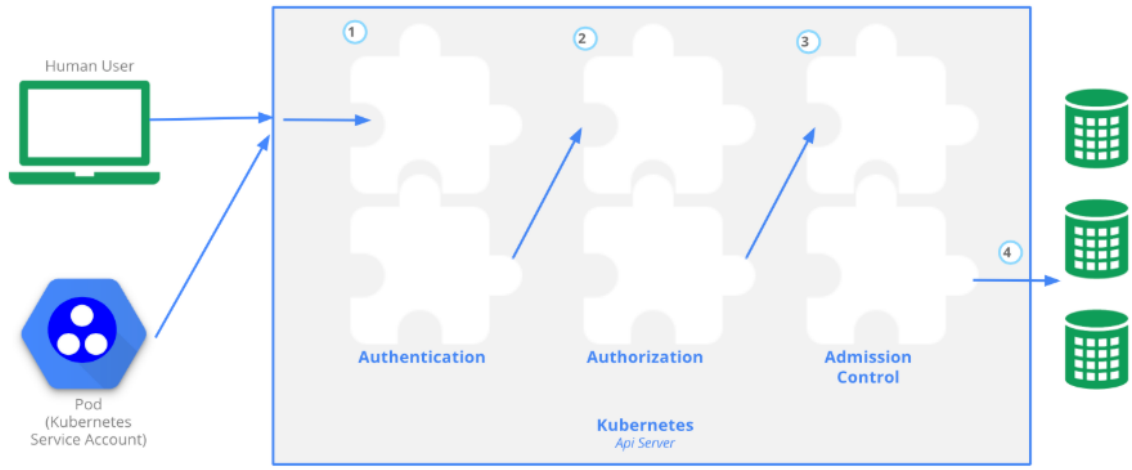
- Discuss the authentication, authorization, and access control stages of the Kubernetes API access.
- Understand the different kinds of Kubernetes users.
- Explore the different modules for authentication and authorization.

Authentication, Authorization, and Admission Control - Overview

To access and manage any Kubernetes resource or object in the cluster, we need to access a specific API endpoint on the API server. Each access request goes through the following three stages:

- **Authentication**
  Logs in a user.
- **Authorization**
  Authorizes the API requests added by the logged-in user.
- **Admission Control**
  Software modules that can modify or reject the requests based on some additional checks, like a pre-set **Quota**.

The following image depicts the above stages:



Accessing the API
(Retrieved from kubernetes.io)

Authentication I

Kubernetes does not have an object called *user*, nor does it store *usernames* or other related details in its object store. However, even without that, Kubernetes can use usernames for access control and request logging, which we will explore in this chapter.

Kubernetes has two kinds of users:

- **Normal Users**
  They are managed outside of the Kubernetes cluster via independent services like User/Client Certificates, a file listing usernames/passwords, Google accounts, etc.
- **Service Accounts**
  With Service Account users, in-cluster processes communicate with the API server to perform different operations. Most of the Service Account users are created automatically via the API server, but they can also be created manually. The Service Account users are tied to a given Namespace and mount the respective credentials to communicate with the API server as Secrets.

If properly configured, Kubernetes can also support **anonymous requests**, along with requests from Normal Users and Service Accounts. **User impersonation** is also supported for a user to be able to act as another user, a useful feature for administrators when troubleshooting authorization policies.

Authentication II

For authentication, Kubernetes uses different authentication modules:

- **Client Certificates**
  To enable client certificate authentication, we need to reference a file containing one or more certificate authorities by passing the **`--client-ca-file=SOMEFILE`** option to the API server. The certificate authorities mentioned in the file would validate the client certificates presented to the API server. A demonstration video covering this topic is also available at the end of this chapter.
- **Static Token File**
  We can pass a file containing pre-defined bearer tokens with the **`--token-auth-file=SOMEFILE`** option to the API server. Currently, these tokens would last indefinitely, and they cannot be changed without restarting the API server.
- **Bootstrap Tokens**
  This feature is currently in beta status and is mostly used for bootstrapping a new Kubernetes cluster.
- **Static Password File**
  It is similar to *Static Token File.* We can pass a file containing basic authentication details with the **`--basic-auth-file=SOMEFILE`** option. These credentials would last indefinitely, and passwords cannot be changed without restarting the API server.
- **Service Account Tokens**
  This is an automatically enabled authenticator that uses signed bearer tokens to verify the requests. These tokens get attached to Pods using the ServiceAccount Admission Controller, which allows in-cluster processes to talk to the API server.
- **OpenID Connect Tokens**
  OpenID Connect helps us connect with OAuth2 providers, such as Azure Active Directory, Salesforce, Google, etc., to offload the authentication to external services.

- **Webhook Token Authentication**
  With Webhook-based authentication, verification of bearer tokens can be offloaded to a remote service.
- **Authenticating Proxy**
  If we want to program additional authentication logic, we can use an authenticating proxy.

We can enable multiple authenticators, and the first module to successfully authenticate the request short-circuits the evaluation. In order to be successful, you should enable at least two methods: the service account tokens authenticator and one of the user authenticators.

Authorization I

After a successful authentication, users can send the API requests to perform different operations. Then, those API requests get authorized by Kubernetes using various authorization modules.

Some of the API request attributes that are reviewed by Kubernetes include user, group, extra, Resource or Namespace, to name a few. Next, these attributes are evaluated against policies. If the evaluation is successful, then the request will be allowed, otherwise it will get denied. Similar to the Authentication step, Authorization has multiple modules/authorizers. More than one module can be configured for one Kubernetes cluster, and each module is checked in sequence. If any authorizer approves or denies a request, then that decision is returned immediately.

Next, we will discuss the authorizers that are supported by Kubernetes.

Authorization II

Authorization modules (Part 1):

- **Node Authorizer**
  Node authorization is a special-purpose authorization mode which specifically authorizes API requests made by kubelets. It authorizes the kubelet's read operations for services, endpoints, nodes, etc., and writes operations for nodes, pods, events, etc. For more details, please review the Kubernetes documentation.

- **Attribute-Based Access Control (ABAC) Authorizer**
  With the ABAC authorizer, Kubernetes grants access to API requests, which combine policies with attributes. In the following example, user *student* can only read Pods in the Namespace **`lfs158`**.

```
{
  "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
   "kind": "Policy",
  "spec": {
   "user": "student",
       "namespace": "lfs158",
   "resource": "pods",
   "readonly": true
  }
}
```

  To enable the ABAC authorizer, we would need to start the API server with the **`--authorization-mode=ABAC`** option. We would also need to specify the authorization policy with **`--authorization-policy-file=PolicyFile.json`**. For more details, please review the Kubernetes documentation.

- **Webhook Authorizer**
  With the Webhook authorizer, Kubernetes can offer authorization decisions to some third-party services, which would return *true* for successful authorization, and *false* for failure. In order to enable the Webhook authorizer, we need to start the API server with the **`--authorization-webhook-config-file=SOME_FILENAME`** option, where **`SOME_FILENAME`** is the configuration of the remote authorization service. For more details, please see the Kubernetes documentation.

Authorization III

Authorization modules (Part 2):

- **Role-Based Access Control (RBAC) Authorizer**
  In general, with RBAC we can regulate the access to resources based on the roles of individual users. In Kubernetes, we can have different roles that can be attached to subjects like users, service accounts, etc. While creating the roles, we restrict resource access by specific operations, such as **`create`**, **`get`**, **`update`**, **`patch`**, etc. These operations are referred to as verbs.

  In RBAC, we can create two kinds of roles:

  Role
  With Role, we can grant access to resources within a specific Namespace.

  ClusterRole
  The ClusterRole can be used to grant the same permissions as Role does, but its scope is cluster-wide.

  In this course, we will focus on the first kind, **Role**. Below you will find an example:

- **kind: Role**
- **apiVersion: rbac.authorization.k8s.io/v1**
- **metadata:**
- **namespace: lfs158**
- **name: pod-reader**
- **rules:**
- **- apiGroups: [""] # "" indicates the core API group**
- **resources: ["pods"]**

```
  verbs: ["get", "watch", "list"]
```

As you can see, it creates a **`pod-reader`** role, which has access only to read the Pods of **`lfs158`** Namespace. Once the role is created, we can bind users with *RoleBinding*.

There are two kinds of *RoleBindings*:

RoleBinding
It allows us to bind users to the same namespace as a Role. We could also refer a ClusterRole in RoleBinding, which would grant permissions to Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.

ClusterRoleBinding
It allows us to grant access to resources at a cluster-level and to all Namespaces.

In this course, we will focus on the first kind, ***RoleBinding***. Below, you will find an example:

```
kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

  name: pod-read-access

  namespace: lfs158

subjects:

- kind: User

  name: student

  apiGroup: rbac.authorization.k8s.io
```

```
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

As you can see, it gives access to the *student* user to read the Pods of **`lfs158`** Namespace.

To enable the RBAC authorizer, we would need to start the API server with the **`--authorization-mode=RBAC`** option. With the RBAC authorizer, we dynamically configure policies. For more details, please review the Kubernetes documentation.

Admission Control

Admission control is used to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc. We force these policies using different admission controllers, like ResourceQuota, DefaultStorageClass, AlwaysPullImages, etc. They come into effect only after API requests are authenticated and authorized.

To use admission controls, we must start the Kubernetes API server with the **`--enable-admission-plugins`**, which takes a comma-delimited, ordered list of controller names:

```
--enable-admission-plugins=NamespaceLifecycle,ResourceQuota,PodSecurityPolicy,DefaultStorageClass
```

Kubernetes has some admission controllers enabled by default. For more details, please review the Kubernetes documentation.

Authentication and Authorization Exercise Guide

LINIKLFS2017-V001100_DTH.mp4 8-Authentication-Authorization-en.srt 8-Authentication-Authorization-en.txt

This exercise guide assumes the following environment, which by default uses the certificate and key from **`/var/lib/minikube/certs/`**, and **`RBAC`** mode for authorization:

Minikube v1.0.1

Kubernetes v1.14.1

Docker 18.06.3-ce

This exercise guide may be used together with the video demonstration following on the next page and it has been updated for the environment mentioned above, while the video presents an older Minikube distribution with Kubernetes v1.9.

Start Minikube:

```
$ minikube start
```

View the content of the **`kubectl`** client's configuration file, observing the only context **`minikube`** and the only user **`minikube`**, created by default:

```
$ kubectl config view
```

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
current-context: minikube
kind: Config
preferences: {}
```

```
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/client.crt
    client-key: /home/student/.minikube/client.key
```

Create **`lfs158`** namespace:

```
$ kubectl create namespace lfs158

namespace/lfs158 created
```

Create **`rbac`** directory and **`cd`** into it:

```
$ mkdir rbac

$ cd rbac/
```

Create a **`private key`** for the **`student`** user with **`openssl`** tool, then create a **`certificate signing request`** for the **`student`** user with **`openssl`** tool:

```
~/rbac$ openssl genrsa -out student.key 2048

Generating RSA private key, 2048 bit long modulus (2 primes)
.................................................+++++
.........................+++++
e is 65537 (0x010001)

~/rbac$ openssl req -new -key student.key -out student.csr -subj
"/CN=student/O=learner"
```

Create a YAML configuration file for a **`certificate signing request`** object, and save it with a blank value for the **`request`** field:

```
~/rbac$ vim signing-request.yaml

apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
  name: student-csr
spec:
  groups:
```

```
 - system:authenticated
 request: <assign encoded value from next cat command>
 usages:
 - digital signature
 - key encipherment
 - client auth
```

View the **certificate**, encode it in **base64**, and assign it to the **request** field in the **signing-request.yaml** file:

```
~/rbac$ cat student.csr | base64 | tr -d '\n'

LS0tLS1CRUd...1QtLS0tLQo=

~/rbac$ vim signing-request.yaml

apiVersion: certificates.k8s.io/v1beta1
kind: CertificateSigningRequest
metadata:
 name: student-csr
spec:
 groups:
 - system:authenticated
 request: LS0tLS1CRUd...1QtLS0tLQo=
 usages:
 - digital signature
 - key encipherment
 - client auth
```

Create the **certificate signing request** object, then list the certificate signing request objects. It shows a **pending** state:

```
~/rbac$ kubectl create -f signing-request.yaml

certificatesigningrequest.certificates.k8s.io/student-csr created

~/rbac$ kubectl get csr

NAME        AGE  REQUESTOR     CONDITION
student-csr  27s  minikube-user  Pending
```

Approve the **certificate signing request** object, then list the certificate signing request objects again. It shows both **approved** and **issued** states:

```
~/rbac$ kubectl certificate approve student-csr

certificatesigningrequest.certificates.k8s.io/student-csr approved

~/rbac$ kubectl get csr

NAME        AGE   REQUESTOR      CONDITION
student-csr   77s   minikube-user   Approved,Issued
```

Extract the approved **certificate** from the **certificate signing request**, decode it with **base64** and save it as a **certificate file**. Then view the certificate in the newly created certificate file:

```
~/rbac$ kubectl get csr student-csr -o jsonpath='{.status.certificate}' | base64 --decode > student.crt

~/rbac$ cat student.crt

-----BEGIN CERTIFICATE-----
MIIDGzCCA...
...
...NOZRRZBVunTjK7A==
-----END CERTIFICATE-----
```

Configure the **student** user's credentials by assigning the **key** and **certificate**:

```
~/rbac$ kubectl config set-credentials student --client-certificate=student.crt --client-key=student.key

User "student" set.
```

Create a new **context** entry in the **kubectl** client's configuration file for the **student** user, associated with the **lfs158** namespace in the **minikube** cluster:

```
~/rbac$ kubectl config set-context student-context --cluster=minikube --namespace=lfs158 --user=student

Context "student-context" created.
```

View the contents of the **kubectl** client's configuration file again, observing the new **context** entry **student-context**, and the new **user** entry **student**:

```
~/rbac$ kubectl config view

apiVersion: v1
clusters:
- cluster:
    certificate-authority: /home/student/.minikube/ca.crt
    server: https://192.168.99.100:8443
  name: minikube
contexts:
- context:
    cluster: minikube
    user: minikube
  name: minikube
- context:
    cluster: minikube
    namespace: lfs158
    user: student
  name: student-context
current-context: minikube
kind: Config
preferences: {}
users:
- name: minikube
  user:
    client-certificate: /home/student/.minikube/client.crt
    client-key: /home/student/.minikube/client.key
- name: student
  user:
    client-certificate: /home/student/rbac/student.crt
    client-key: /home/student/rbac/student.key
```

While in the default **minikube context**, create a new **deployment** in the **lfs158** namespace:

```
~/rbac$ kubectl -n lfs158 create deployment nginx --image=nginx:alpine

deployment.apps/nginx created
```

From the new **`context student-context`** try to list pods. The attempt fails because the **`student`** user has no permissions configured for the **`student-context`**:

```
~/rbac$ kubectl --context=student-context get pods

Error from server (Forbidden): pods is forbidden: User "student" cannot list resource "pods" in API group "" in the namespace "lfs158"
```

The following steps will assign a limited set of permissions to the **`student`** user in the **`student-context`**.

Create a YAML configuration file for a **`pod-reader role`** object, which allows only **`get, watch, list`** actions in the **`lfs158`** namespace against **`pod`** objects. Then create the **`role`** object and list it from the default **`minikube context`**, but from the **`lfs158`** namespace:

```
~/rbac$ vim role.yaml

apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader
  namespace: lfs158
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

~/rbac$ kubectl create -f role.yaml

role.rbac.authorization.k8s.io/pod-reader created

~/rbac$ kubectl -n lfs158 get roles

NAME         AGE
pod-reader   57s
```

Create a YAML configuration file for a **`rolebinding`** object, which assigns the permissions of the **`pod-reader role`** to the **`student`** user. Then create the **rolebinding** object and list it from the default **minikube context**, but from the **lfs158** namespace:

```
~/rbac$ vim rolebinding.yaml

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-read-access
  namespace: lfs158
subjects:
- kind: User
  name: student
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io

~/rbac$ kubectl create -f rolebinding.yaml

rolebinding.rbac.authorization.k8s.io/pod-read-access created

~/rbac$ kubectl -n lfs158 get rolebindings

NAME            AGE
pod-read-access   23s
```

Now that we have assigned permissions to the **student** user, we can successfully list **pods** from the new **context student-context**.

```
~/rbac$ kubectl --context=student-context get pods

NAME                  READY   STATUS    RESTARTS   AGE
nginx-77595c695-f2xmd   1/1     Running   0          7m41s
```

Chapter 10. Services

Although the microservices driven architecture aims to decouple the components of an application, microservices still need agents to logically tie or group them together and to load balance traffic to the ones that are part of such a logical set.

In this chapter, we will learn about **Services**, used to group Pods to provide common access points from the external world to the containerized applications. We will learn about the **kube-proxy** daemon, which runs on each worker node to provide access to services**.** We will also discuss **service discovery** and **service types**, which decide the access scope of a service.

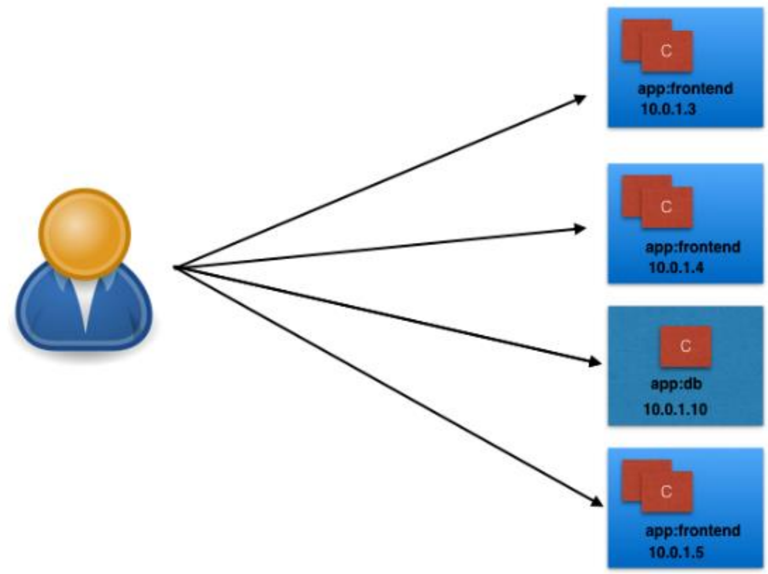
By the end of this chapter, you should be able to:

- Discuss the benefits of logically grouping Pods with Services to access an application.
- Explain the role of the **`kube-proxy`** daemon running on each worker node.
- Explore the Service discovery options available in Kubernetes.
- Discuss different Service types.

Connecting Users to Pods

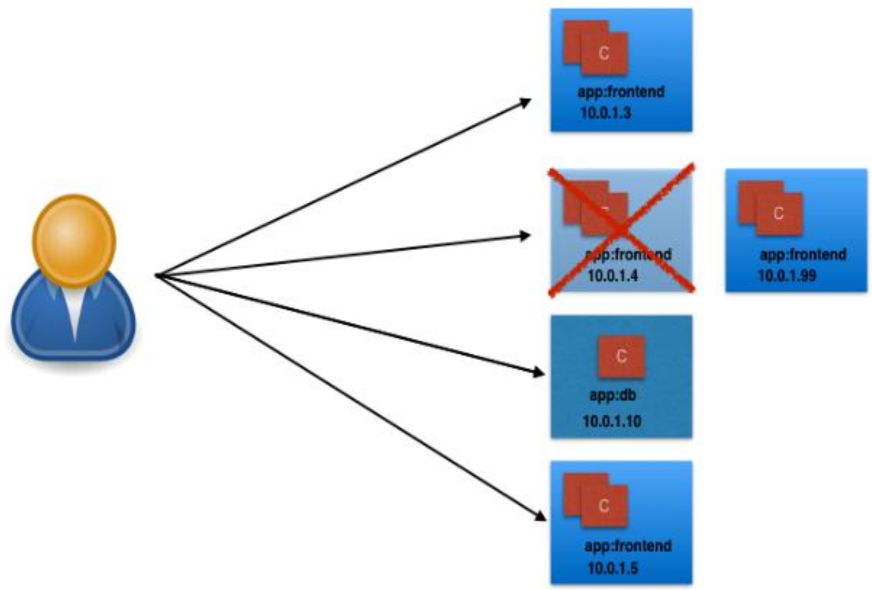
To access the application, a user/client needs to connect to the Pods. As Pods are ephemeral in nature, resources like IP addresses allocated to it cannot be static. Pods could be terminated abruptly or be rescheduled based on existing requirements.

Let's take, for example, a scenario in which a user/client is connected to a Pod using its IP address.



A Scenario Where a User Is Connected to a Pod via Its IP Address

Unexpectedly, the Pod to which the user/client is connected is terminated, and a new Pod is created by the controller. The new Pod will have a new IP address, which will not be known automatically to the user/client of the earlier Pod.

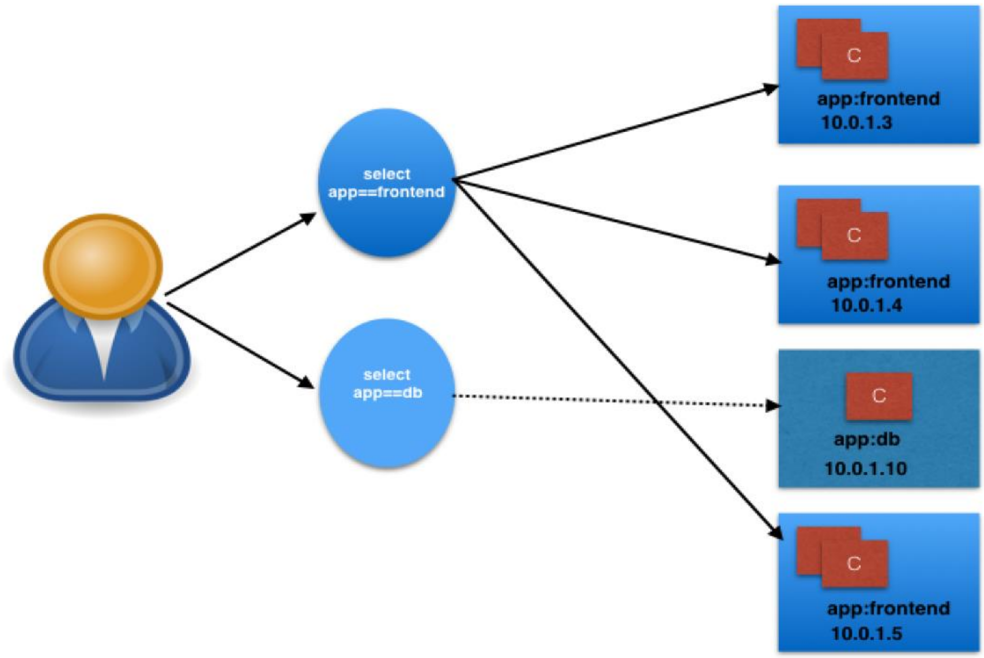


A New Pod Is Created After the Old One Terminated Unexpectedly

To overcome this situation, Kubernetes provides a higher-level abstraction called [Service](), which logically groups Pods and defines a policy to access them. This grouping is achieved via **Labels** and **Selectors**.

Services

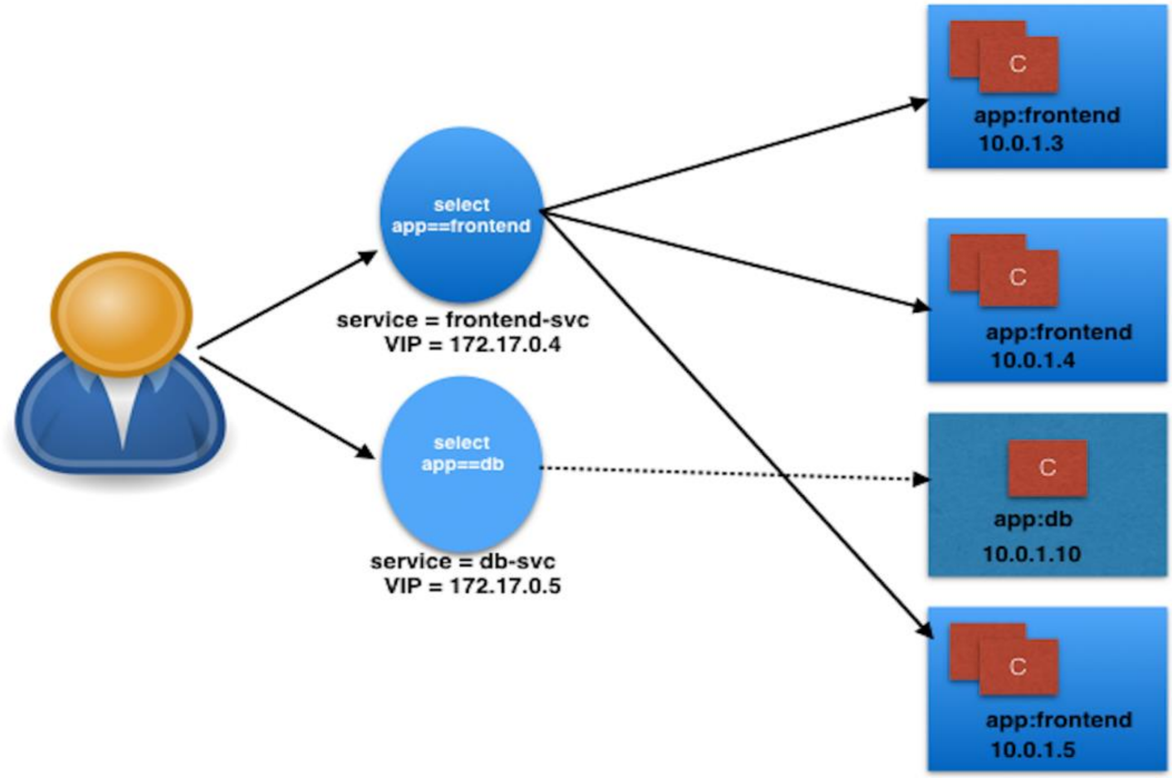
In the following graphical representation, `app` is the Label key, `frontend` and `db` are Label values for different Pods.



Grouping of Pods using Labels and Selectors

Using the selectors **`app==frontend`** and **`app==db`**, we group Pods into two logical sets: one with 3 Pods, and one with a single Pod.

We assign a name to the logical grouping, referred to as a **Service**. In our example, we create two Services, **`frontend-svc`**, and **`db-svc`**, and they have the **`app==frontend`** and the **`app==db`** Selectors, respectively.



Grouping of Pods using the Service object

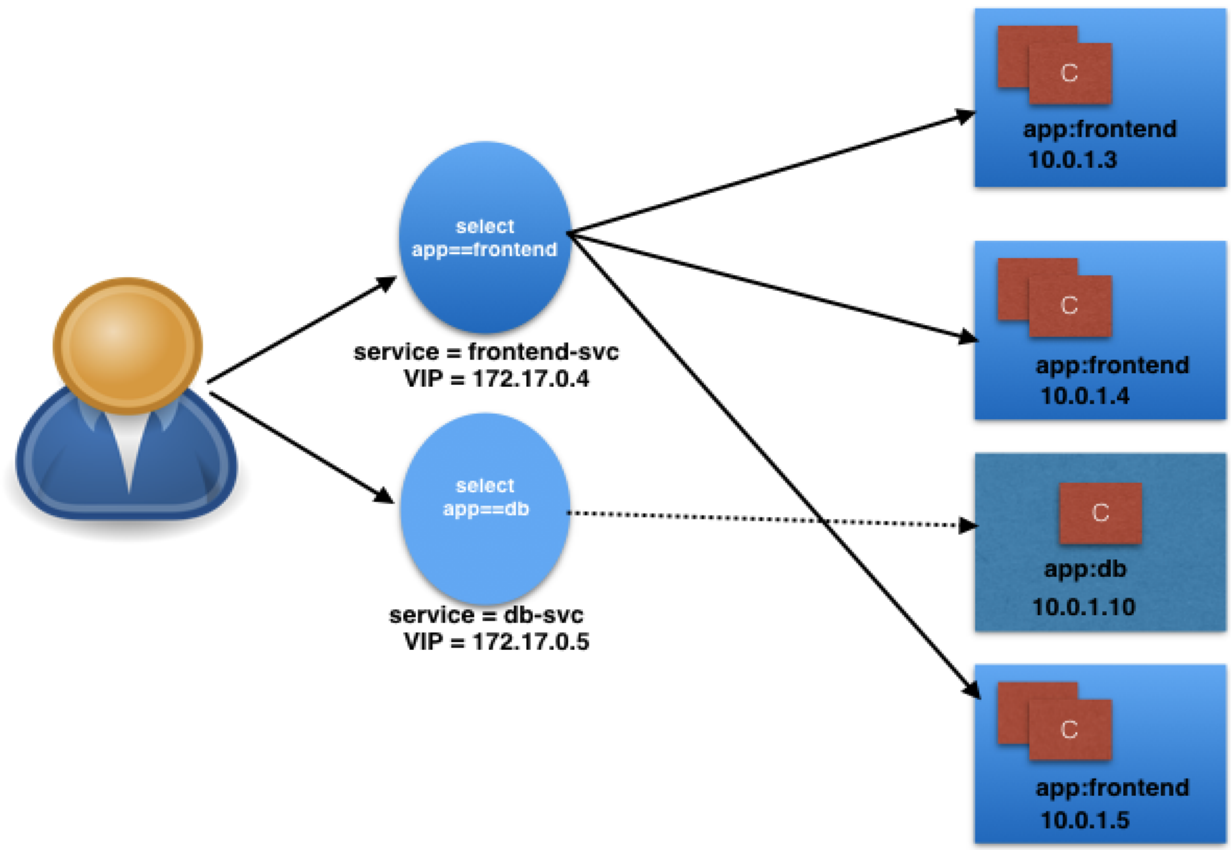
Services can expose single Pods, ReplicaSets, Deployments, DaemonSets, and StatefulSets.

Service Object Example

The following is an example of a Service object definition:

```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
  - protocol: TCP
    port: 80
    targetPort: 5000
```

In this example, we are creating a **`frontend-svc`** Service by selecting all the Pods that have the Label key=**`app`** set to value=**`frontend`**. By default, each Service receives an IP address routable only inside the cluster, known as **`ClusterIP`**. In our example, we have **`172.17.0.4`** and **`172.17.0.5`** as ClusterIPs assigned to our **`frontend-svc`** and **`db-svc`** Services, respectively.



Accessing the Pods using Service Object

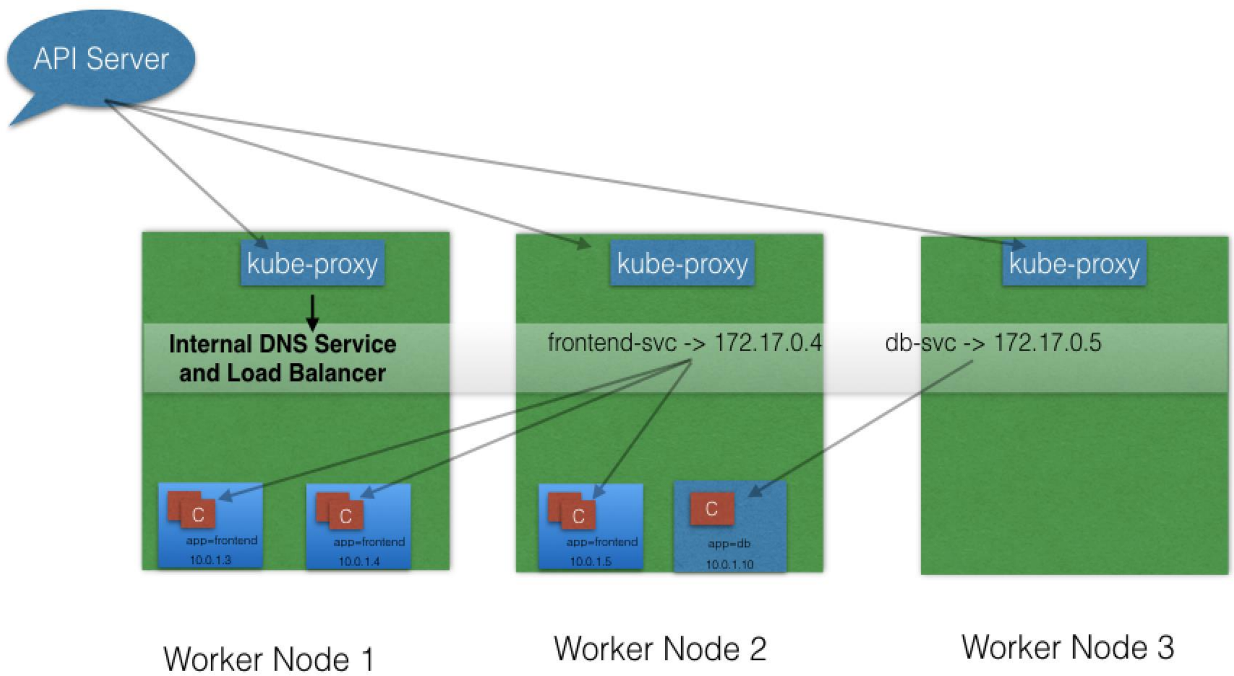
The user/client now connects to a Service via its **`ClusterIP`**, which forwards traffic to one of the Pods attached to it. A Service provides load balancing by default while selecting the Pods for traffic forwarding.

While the Service forwards traffic to Pods, we can select the **`targetPort`** on the Pod which receives the traffic. In our example, the **`frontend-svc`** Service receives requests from the user/client on **`port`** `80` and then forwards these requests to one of the attached Pods on the **`targetPort`** `5000`. If the **`targetPort`** is not defined explicitly, then traffic will be forwarded to Pods on the **`port`** on which the Service receives traffic.

A logical set of a Pod's IP address, along with the **`targetPort`** is referred to as a **`Service endpoint`**. In our example, the **`frontend-svc`** Service has 3 endpoints: **`10.0.1.3:5000`**, **`10.0.1.4:5000`**, and **`10.0.1.5:5000`**. Endpoints are created and managed automatically by the Service, not by the Kubernetes cluster administrator.

kube-proxy

All worker nodes run a daemon called **kube-proxy**, which watches the API server on the master node for the addition and removal of Services and endpoints. In the example below, for each new Service, on each node, **`kube-proxy`** configures **iptables** rules to capture the traffic for its ClusterIP and forwards it to one of the Service's endpoints. Therefore any node can receive the external traffic and then route it internally in the cluster based on the **iptables** rules. When the Service is removed, **`kube-proxy`** removes the corresponding **iptables** rules on all nodes as well.



kube-proxy, Services, and Endpoints

Service Discovery

As Services are the primary mode of communication in Kubernetes, we need a way to discover them at runtime. Kubernetes supports two methods for discovering Services:

- **Environment Variables**
  As soon as the Pod starts on any worker node, the **`kubelet`** daemon running on that node adds a set of environment variables in the Pod for all active Services. For example, if we have an active Service called **`redis-master`**, which exposes port **`6379`**, and its ClusterIP is **`172.17.0.6`**, then, on a newly created Pod, we can see the following environment variables:

  ```
  REDIS_MASTER_SERVICE_HOST=172.17.0.6
  REDIS_MASTER_SERVICE_PORT=6379
  REDIS_MASTER_PORT=tcp://172.17.0.6:6379
  REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379
  REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
  REDIS_MASTER_PORT_6379_TCP_PORT=6379
  REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6
  ```

  With this solution, we need to be careful while ordering our Services, as the Pods will not have the environment variables set for Services which are created after the Pods are created.

- **DNS**
  Kubernetes has an add-on for DNS, which creates a DNS record for each Service and its format is **`my-svc.my-namespace.svc.cluster.local`**. Services within the same Namespace find other Services just by their name. If we add a Service **`redis-master`** in **`my-ns`** Namespace, all Pods in the same Namespace lookup the Service just by its name, **`redis-master`**. Pods from other Namespaces lookup the same Service by adding the respective Namespace as a suffix, such as **`redis-master.my-ns`**.

  This is the most common and highly recommended solution. For example, in the previous section's image, we have seen that an internal DNS is configured, which maps our Services **`frontend-svc`** and **`db-svc`** to **`172.17.0.4`** and **`172.17.0.5`**, respectively.

ServiceType

While defining a Service, we can also choose its access scope. We can decide whether the Service:

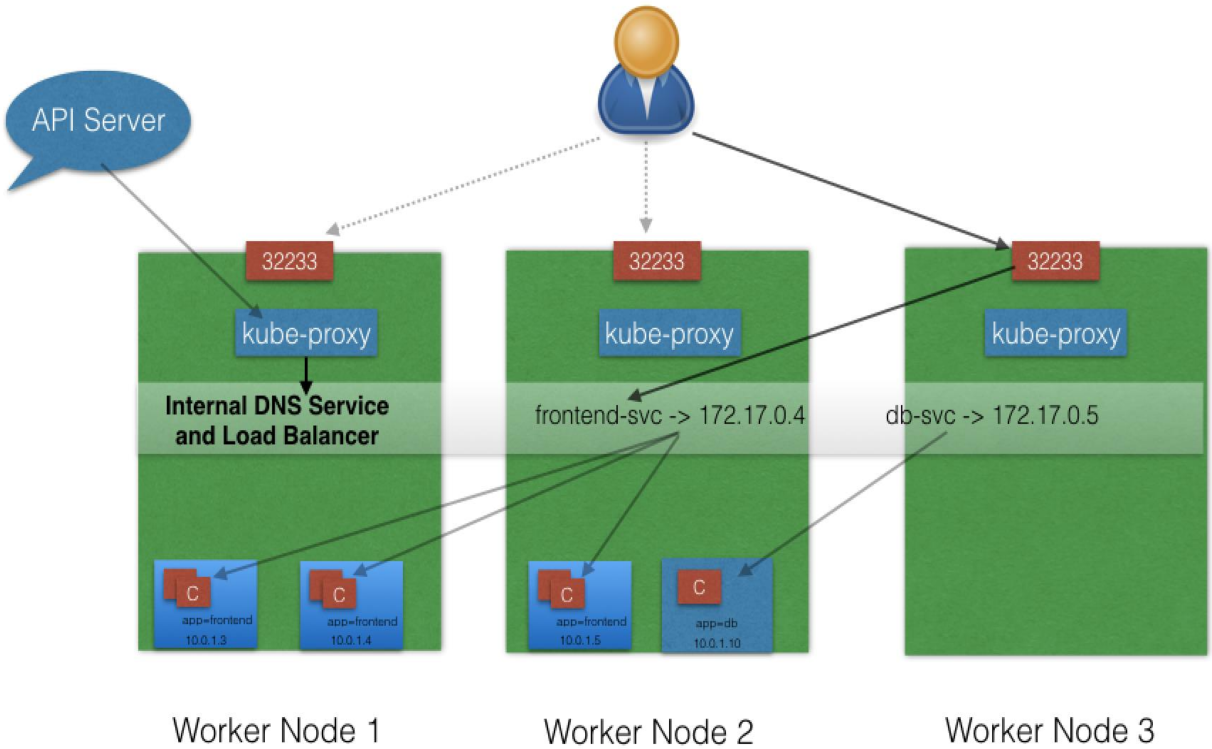
- Is only accessible within the cluster
- Is accessible from within the cluster and the external world
- Maps to an entity which resides either inside or outside the cluster.

Access scope is decided by ServiceType, which can be configured when creating the Service.

ServiceType: ClusterIP and NodePort

ClusterIP is the default *ServiceType*. A Service receives a Virtual IP address, known as its ClusterIP. This Virtual IP address is used for communicating with the Service and is accessible only within the cluster.

With the **NodePort** *ServiceType*, in addition to a ClusterIP, a high-port, dynamically picked from the default range **`30000-32767`**, is mapped to the respective Service, from all the worker nodes. For example, if the mapped NodePort is **`32233`** for the service **`frontend-svc`**, then, if we connect to any worker node on port **`32233`**, the node would redirect all the traffic to the assigned ClusterIP - **`172.17.0.4`**. If we prefer a specific high-port number instead, then we can assign that high-port number to the NodePort from the default range.



NodePort

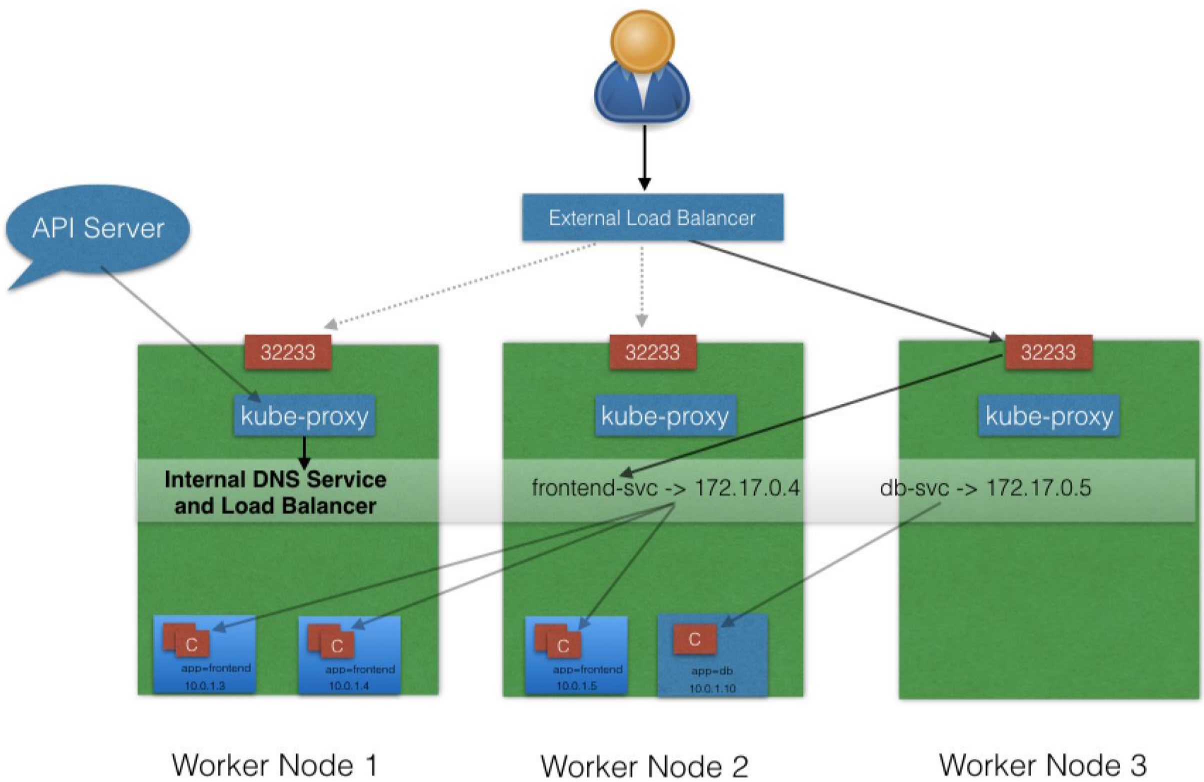
The **`NodePort`** *ServiceType* is useful when we want to make our Services accessible from the external world. The end-user connects to any worker node on the specified high-port, which proxies the request internally to the ClusterIP of the Service, then the request is forwarded to the applications running inside the cluster. To access multiple applications from the external world, administrators can configure a reverse proxy - an ingress, and define rules that target Services within the cluster.

ServiceType: LoadBalancer

With the **LoadBalancer** *ServiceType*:

- NodePort and ClusterIP are automatically created, and the external load balancer will route to them
- The Service is exposed at a static port on each worker node

- The Service is exposed externally using the underlying cloud provider's load balancer feature.

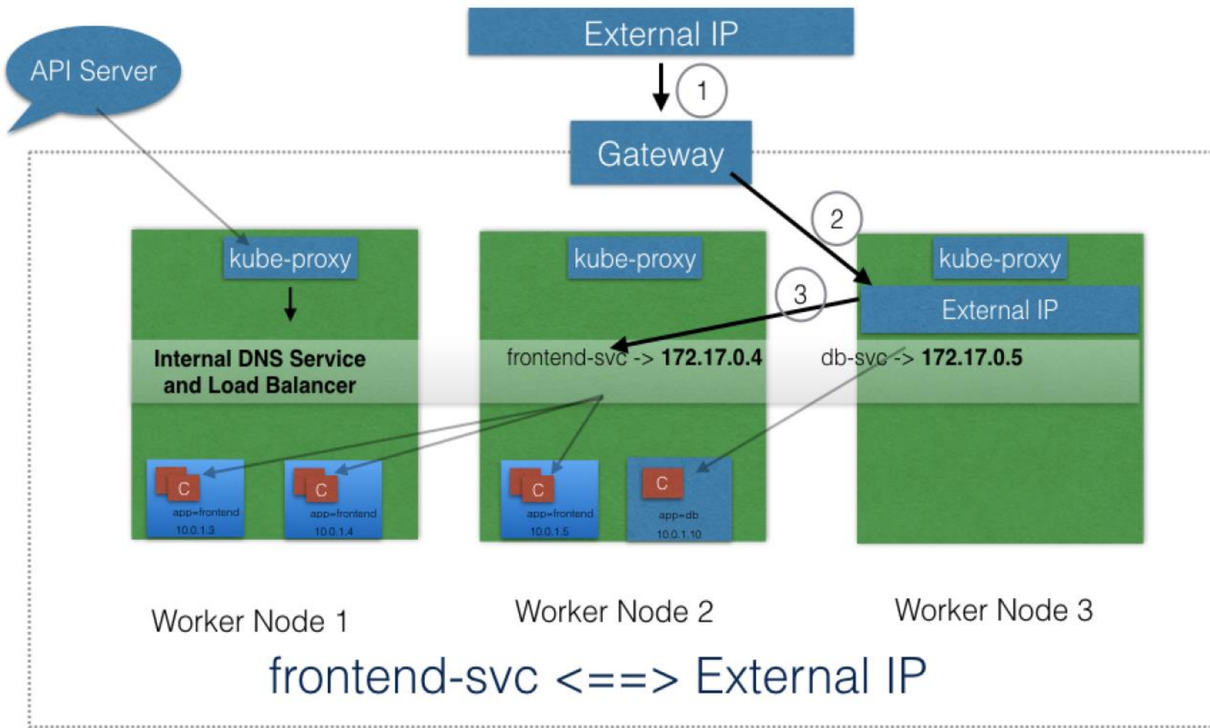


LoadBalancer

The LoadBalancer *ServiceType* will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes, as is the case with the Google Cloud Platform and AWS. If no such feature is configured, the LoadBalancer IP address field is not populated, and the Service will work the same way as a NodePort type Service.

ServiceType: ExternalIP

A Service can be mapped to an **ExternalIP** address if it can route to one or more of the worker nodes. Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the Service endpoints. This type of service requires an external cloud provider such as Google Cloud Platform or AWS.



ExternalIP

Please note that ExternalIPs are not managed by Kubernetes. The cluster administrator has to configure the routing which maps the ExternalIP address to one of the nodes.

ServiceType: ExternalName

ExternalName is a special *ServiceType*, that has no Selectors and does not define any endpoints. When accessed within the cluster, it returns a `CNAME` record of an externally configured Service.

The primary use case of this *ServiceType* is to make externally configured Services like `my-database.example.com` available to applications inside the cluster. If the externally defined Service resides within the same Namespace, using just the name **`my-database`** would make it available to other applications and Services within that same Namespace.

Chapter 11. Deploying a Stand-Alone Application

In this chapter, we will learn how to deploy an application using the **Dashboard (Kubernetes WebUI)** and the **Command Line Interface (CLI)**. We will also expose the application with a NodePort type Service, and access it from the external world.

By the end of this chapter, you should be able to:

- Deploy an application from the dashboard.
- Deploy an application from a YAML file using kubectl.
- Expose a service using NodePort.
- Access the application from the external world.

Deploying an Application Using the Dashboard I

In the next few sections, we will learn how to deploy an **`nginx`** webserver using the **`nginx:alpine`** Docker image.

Start Minikube and verify that it is running

Run this command first:

```
$ minikube start
```

Allow several minutes for Minikube to start, then verify Minikube status:

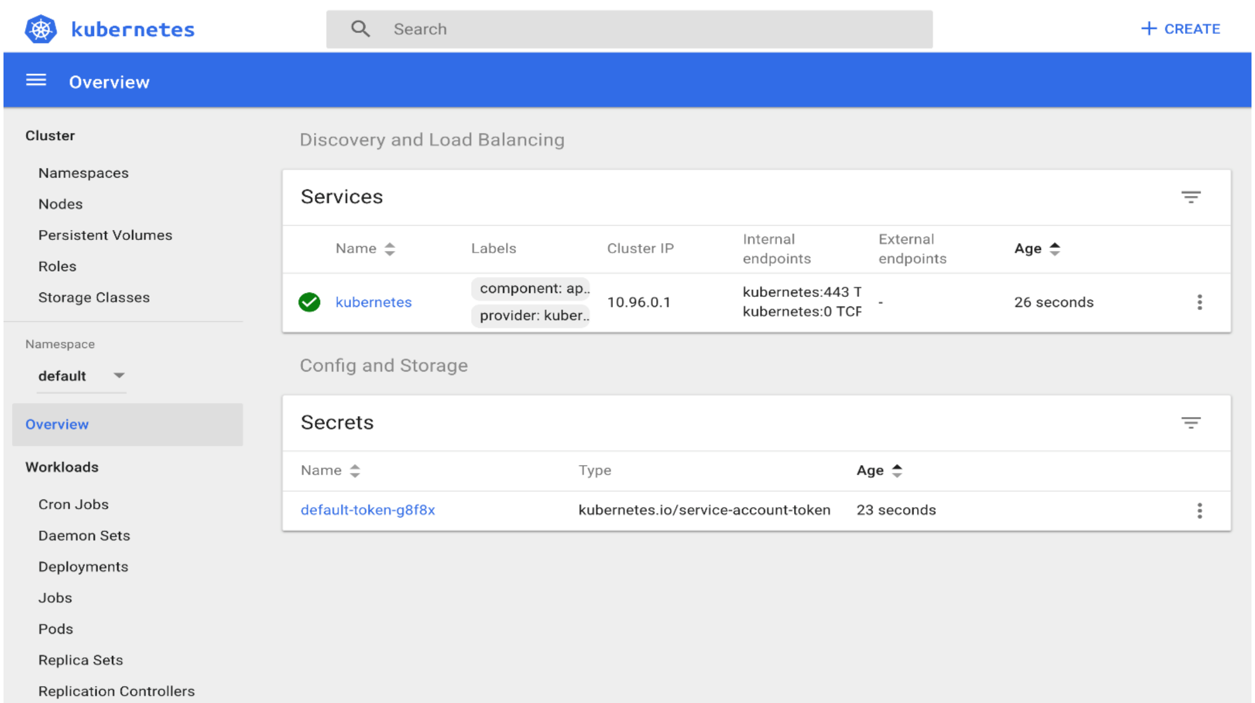
```
$ minikube status
host: Running
kubelet: Running
apiserver: Running
kubectl: Correctly Configured: pointing to minikube-vm at 192.168.99.100
```

Start the Minikube Dashboard

To access the Kubernetes Web IU, we need to run the following command:

```
$ minikube dashboard
```

Running this command will open up a browser with the Kubernetes Web UI, which we can use to manage containerized applications. By default, the dashboard is connected to the `default` Namespace. So, all the operations that we will do in this chapter will be performed inside the `default` Namespace.



Deploying an Application - Accessing the Dashboard

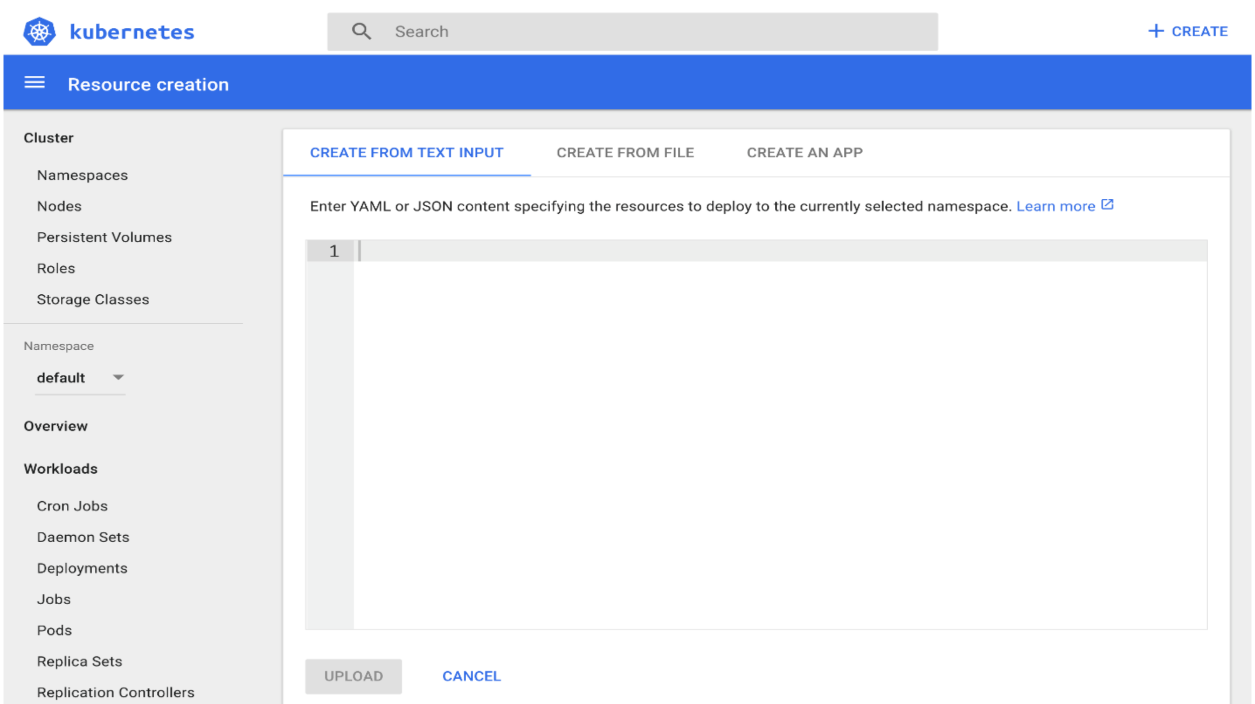
NOTE: In case the browser is not opening another tab and does not display the Dashboard as expected, verify the output in your terminal as it may display a link for the Dashboard (together with some Error messages). Copy and paste that link in a new tab of your browser. Depending on your terminal's features you may be able to just click or right-click the link to open directly in the browser. The link may look similar to:
http://127.0.0.1:37751/api/v1/namespaces/kube-system/services/http:kubernetes-dashboard:/proxy/

Chances are that the only difference is the PORT number, which above is 37751. Your port number may be different.

After a logout/login or a reboot of your workstation the normal behavior should be expected (where the **minikube dashboard** command directly opens a new tab in your browser displaying the Dashboard).

Deploying an Application Using the Dashboard II

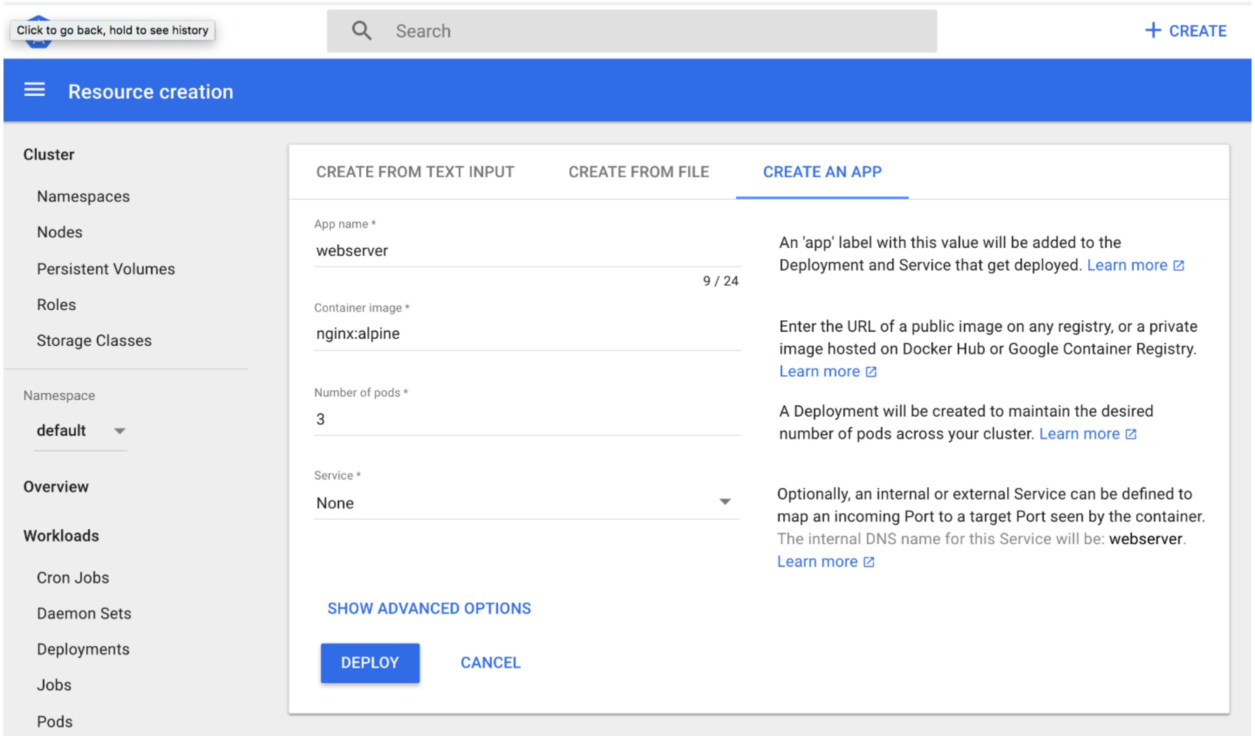
Deploy a webserver using the `nginx:alpine` image
From the dashboard, click on the *+CREATE* tab at the top right corner of the Dashboard. That will open the create interface as seen below:



Deploy a Containerized Application Web Interface

From that, we can create an application using a valid YAML/JSON configuration data of file, or manually from the *CREATE AN APP* section. Click on the *CREATE AN APP* tab and provide the following application details:

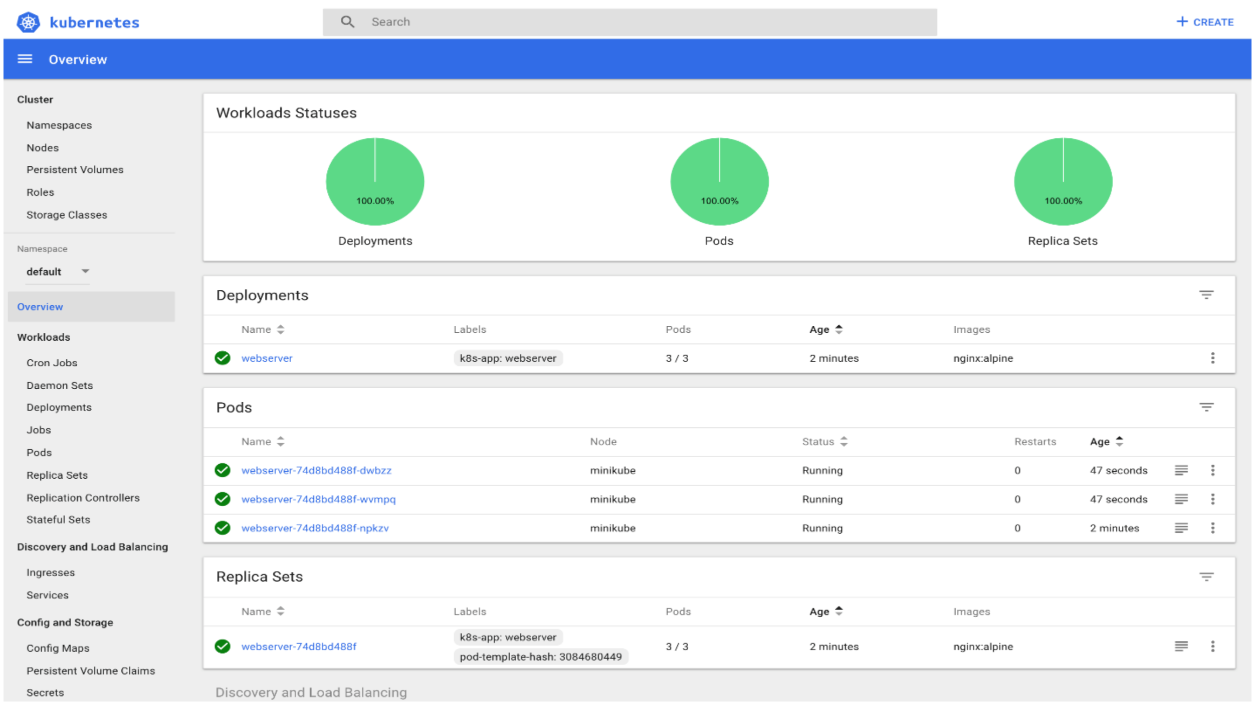
- The application name is **`webserver`**
- The Docker image to use is **`nginx:alpine`**, where **`alpine`** is the image tag
- The replica count, or the number of Pods, is 3
- No Service, as we will be creating it later.



Deploy a Containerized Application Web Interface

If we click on *Show Advanced Options*, we can specify options such as Labels, Namespace, Environment Variables, etc. By default, the **`app`** Label is set to the application name. In our example **`k8s-app:webserver`** Label is set to all objects created by this Deployment: Pods and Services (when exposed).

By clicking on the *Deploy* button, we trigger the deployment. As expected, the Deployment **`webserver`** will create a ReplicaSet (**`webserver-74d8bd488f`**), which will eventually create three Pods (**`webserver-74d8bd488f-xxxxx`**).



Deployment Details

NOTE: Add the full URL in the Container Image field **`docker.io/library/nginx:alpine`** if any issues are encountered with the simple **`nginx:alpine`** image name (or use the **`k8s.gcr.io/nginx:alpine`** URL if it works instead).

Deploying an Application Using the Dashboard III

Once we created the **`webserver`** Deployment, we can use the resource navigation panel from the left side of the Dashboard to display details of Deployments, ReplicaSets, and Pods in the **`default`** Namespace. The resources displayed by the Dashboard match one-to-one resources displayed from the CLI via **`kubectl`**.

List the Deployments

We can list all the Deployments in the **`default`** Namespace using the **`kubectl get deployments`** command:

```
$ kubectl get deployments
NAME         READY   UP-TO-DATE    AVAILABLE    AGE
webserver    3/3     3             3            9m
```

List the ReplicaSets
We can list all the ReplicaSets in the **`default`** Namespace using the **`kubectl get replicasets`** command:

```
$ kubectl get replicasets
NAME                     DESIRED    CURRENT    READY    AGE
webserver-74d8bd488f     3          3          3        9m
```

List the Pods
We can list all the Pods in the **`default`** namespace using the **`kubectl get pods`** command:

```
$ kubectl get pods
NAME            READY  STATUS   RESTARTS  AGE
webserver-74d8bd488f-dwbzz  1/1  Running  0    9m
webserver-74d8bd488f-npkzv  1/1  Running  0    9m
webserver-74d8bd488f-wvmpq  1/1  Running  0    9m
```

Exploring Labels and Selectors I

Earlier, we have seen that labels and selectors play an important role in grouping a subset of objects on which we can perform operations. Next, we will take a closer look at them.

Look at a Pod's Details
We can look at an object's details using **`kubectl describe`** command. In the following example, you can see a Pod's description:

```
$ kubectl describe pod webserver-74d8bd488f-dwbzz
Name:              webserver-74d8bd488f-dwbzz
Namespace:         default
Priority:          0
Node:              minikube/10.0.2.15
Start Time:        Wed, 15 May 2019 13:17:33 -0500
Labels:            k8s-app=webserver
                   pod-template-hash=74d8bd488f
Annotations:       <none>
Status:            Running
IP:                172.17.0.5
Controlled By:     ReplicaSet/webserver-74d8bd488f
Containers:
  webserver:
    Container
```

```
ID:   docker://96302d70903fe3b45d5ff3745a706d67d77411c5378f1f293
a4bd721896d6420
    Image:           nginx:alpine
    Image ID:        docker-
pullable://nginx@sha256:8d5341da24ccbdd195a82f2b57968ef5f95bc27b
3c3691ace0c7d0acf5612edd
    Port:            <none>
    State:           Running
      Started:       Wed, 15 May 2019 13:17:33 -0500
    Ready:           True
    Restart Count:   0
...
```

The **kubectl describe** command displays many more details of a Pod. For now, however, we will focus on the **Labels** field, where we have a Label set to **k8s-app=webserver**.

Exploring Labels and Selectors II

List the Pods, along with their attached Labels

With the **-L** option to the **kubectl get pods** command, we add extra columns in the output to list Pods with their attached Label keys and their values. In the following example, we are listing Pods with the Label keys **k8s-app** and **label2**:

```
$ kubectl get pods -L k8s-app,label2
NAME              READY  STATUS   RESTARTS  AGE  K8S-APP    LABEL2
webserver-74d8bd488f-dwbzz  1/1   Running  0      14m  webserver
webserver-74d8bd488f-npkzv  1/1   Running  0      14m  webserver
webserver-74d8bd488f-
wvmpq   1/1     Running   0          14m   webserver
```

All of the Pods are listed, as each Pod has the Label key **k8s-app** with value set to **webserver**. We can see that in the **K8S-APP** column. As none of the Pods have the **label2** Label key, no values are listed under the **LABEL2** column.

Exploring Labels and Selectors III

Select the Pods with a given Label

To use a selector with the **kubectl get pods** command, we can use the **-l** option. In the following example, we are selecting all the Pods that have the **k8s-app** Label key set to value **webserver**:

```
$ kubectl get pods -l k8s-app=webserver
NAME                          READY     STATUS    RESTARTS   AGE
webserver-74d8bd488f-dwbzz    1/1       Running   0          17m
webserver-74d8bd488f-npkzv    1/1       Running   0          17m
webserver-74d8bd488f-wvmpq    1/1       Running   0          17m
```

In the example above, we listed all the Pods we created, as all of them have the **`k8s-app`** Label key set to value **`webserver`**.

Try using `k8s-app=webserver1` as the Selector

```
$ kubectl get pods -l k8s-app=webserver1
No resources found.
```

As expected, no Pods are listed.

Deploying an Application Using the CLI I

To deploy an application using the CLI, let's first delete the Deployment we created earlier.

Delete the Deployment we created earlier
We can delete any object using the **`kubectl delete`** command. Next, we are deleting the **`webserver`** Deployment we created earlier with the Dashboard:

```
$ kubectl delete deployments webserver
deployment.extensions "webserver" deleted
```

Deleting a Deployment also deletes the ReplicaSet and the Pods it created:

```
$ kubectl get replicasets
No resources found.

$ kubectl get pods
No resources found.
```

Deploying an Application Using the CLI II

Create a YAML configuration file with Deployment details
Let's create the **`webserver.yaml`** file with the following content:

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

Using **kubectl**, we will create the Deployment from the YAML configuration file. Using the **`-f`** option with the **`kubectl create`** command, we can pass a YAML file as an object's specification, or a URL to a configuration file from the web. In the following example, we are creating a **`webserver`** Deployment:

```
$ kubectl create -f webserver.yaml
deployment.apps/webserver created
```

This will also create a ReplicaSet and Pods, as defined in the YAML configuration file.

```
$  kubectl get replicasets
NAME                  DESIRED    CURRENT    READY      AGE
webserver-b477df957   3          3          3          45s
```

```
$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
webserver-b477df957-7lnw6     1/1     Running   0          2m
webserver-b477df957-j69q2     1/1     Running   0          2m
webserver-b477df957-xvdkf     1/1     Running   0          2m
```

Exposing an Application I

In a previous chapter, we explored different *ServiceTypes*. With *ServiceTypes* we can define the access method for a Service. For a **`NodePort`** *ServiceType*, Kubernetes opens up a static port on all the worker nodes. If we connect to that port from any node, we are proxied to the ClusterIP of the Service. Next, let's use the **`NodePort`** *ServiceType* while creating a Service.

Create a `webserver-svc.yaml` file with the following content:

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

Using `kubectl`, create the Service:

```
$ kubectl create -f webserver-svc.yaml
service/web-service created
```

A more direct method of creating a Service is by exposing the previously created Deployment (this method requires an existing Deployment).

Expose a Deployment with the `kubectl expose` command:

```
$ kubectl expose deployment webserver --name=web-service --
type=NodePort
service/web-service exposed
```

Exposing an Application II

List the Services:

```
$ kubectl get services
NAME        TYPE      CLUSTER-IP    EXTERNAL-IP  PORT(S)       AGE
kubernetes  ClusterIP  10.96.0.1     <none>      443/TCP       1d
web-service NodePort   10.110.47.84  <none>      80:31074/TCP  22s
```

Our **`web-service`** is now created and its ClusterIP is **`10.110.47.84`**. In the **`PORT(S)`** section, we see a mapping of **`80:31074`**, which means that we have reserved a static port 31074 on the node. If we connect to the node on that port, our requests will be proxied to the ClusterIP on port **`80`**.

It is not necessary to create the Deployment first, and the Service after. They can be created in any order. A Service will find and connect Pods based on the Selector.

To get more details about the Service, we can use the **`kubectl describe`** command, as in the following example:

```
$ kubectl describe service web-service
Name:                web-service
Namespace:           default
Labels:              run=web-service
Annotations:         <none>
Selector:            app=nginx
Type:                NodePort
IP:                  10.110.47.84
Port:                <unset>  80/TCP
TargetPort:          80/TCP
NodePort:            <unset>  31074/TCP
Endpoints:           172.17.0.4:80,172.17.0.5:80,172.17.0.6:80
Session Affinity:    None
External Traffic Policy:  Cluster
Events:              <none>
```

`web-service` uses **`app=nginx`** as a Selector to logically group our three Pods, which are listed as endpoints. When a request reaches our Service, it will be served by one of the Pods listed in the **`Endpoints`** section.

Accessing an Application

Our application is running on the Minikube VM node. To access the application from our workstation, let's first get the IP address of the Minikube VM:

```
$ minikube ip
192.168.99.100
```

Now, open the browser and access the application on **192.168.99.100** at port **31074**.

Accessing the Application In the Browser

We could also run the following **minikube** command which displays the application in our browser:

```
$ minikube service web-service
Opening kubernetes service default/web-service in default
browser...
```

We can see the *Nginx* welcome page, displayed by the **webserver** application running inside the Pods created. Our requests could be served by either one of the three endpoints logically grouped by the Service since the Service acts as a Load Balancer in front of its endpoints.

Liveness and Readiness Probes

While containerized applications are scheduled to run in pods on nodes across our cluster, at times the applications may become unresponsive or may be delayed during startup. Implementing **Liveness** and **Readiness Probes** allows the **`kubelet`** to control the health of the application running inside a Pod's container and force a container restart of an unresponsive application. When defining both **Readiness** and **Liveness Probes**, it is recommended to allow enough time for the **Readiness Probe** to possibly fail a few times before a pass, and only then check the **Liveness Probe**.
If **Readiness** and **Liveness Probes** overlap there may be a risk that the container never reaches ready state.

In the next few sections, we will discuss them in more detail.

Liveness

If a container in the Pod is running, but the application running inside this container is not responding to our requests, then that container is of no use to us. This kind of situation can occur, for example, due to application deadlock or memory pressure. In such a case, it is recommended to restart the container to make the application available.

Rather than restarting it manually, we can use a **Liveness Probe**. Liveness probe checks on an application's health, and if the health check fails, **`kubelet`** restarts the affected container automatically.

Liveness Probes can be set by defining:

- Liveness command
- Liveness HTTP request
- TCP Liveness Probe.

We will discuss these three approaches in the next few sections.

Liveness Command

In the following example, we are checking the existence of a file **`/tmp/healthy`**:

```
apiVersion: v1
kind: Pod
metadata:
 labels:
  test: liveness
 name: liveness-exec
spec:
 containers:
 - name: liveness
  image: k8s.gcr.io/busybox
  args:
  - /bin/sh
  - -c
  - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600
  livenessProbe:
        exec:
    command:
    - cat
    - /tmp/healthy
   initialDelaySeconds: 5
```

```
    periodSeconds: 5
```

The existence of the **/tmp/healthy** file is configured to be checked every 5 seconds using the **periodSeconds** parameter. The **initialDelaySeconds** parameter requests the kubelet to wait for 5 seconds before the first probe. When running the command line argument to the container, we will first create the **/tmp/healthy** file, and then we will remove it after 30 seconds. The deletion of the file would trigger a health failure, and our Pod would get restarted.

A demonstration video covering this topic is up next.

9-2.Liveness-en.txt   LINIKLFS2017-V001000_DTH.mp4   9-2.Liveness-en.srt

Liveness HTTP Request

In the following example, the kubelet sends the **HTTP GET** request to the **/healthz** endpoint of the application, on port **8080**. If that returns a failure, then the kubelet will restart the affected container; otherwise, it would consider the application to be alive.

```
livenessProbe:
    httpGet:
            path: /healthz
     port: 8080
     httpHeaders:
            - name: X-Custom-Header
      value: Awesome
    initialDelaySeconds: 3
    periodSeconds: 3
```

TCP Liveness Probe

With TCP Liveness Probe, the kubelet attempts to open the TCP Socket to the container which is running the application. If it succeeds, the application is considered healthy, otherwise the kubelet would mark it as unhealthy and restart the affected container.

```
livenessProbe:
   tcpSocket:
         port: 8080
        initialDelaySeconds: 15
   periodSeconds: 20
```

Readiness Probes

Sometimes, applications have to meet certain conditions before they can serve traffic. These conditions include ensuring that the depending service is ready, or acknowledging that a large dataset needs to be loaded, etc. In such cases, we use **Readiness Probes** and wait for a certain condition to occur. Only then, the application can serve traffic.

A Pod with containers that do not report ready status will not receive traffic from Kubernetes Services.

```
readinessProbe:
   exec:
  command:
  - cat
  - /tmp/healthy
 initialDelaySeconds: 5
```

periodSeconds: 5

Readiness Probes are configured similarly to Liveness Probes. Their configuration also remains the same.

Please review the [Kubernetes documentation] for more details.

10-Deploying a Containerized Applica | 10-Deploying a Containerized Applica | LINIKLFS2017-V001300_DTH.mp4

Chapter 12. Kubernetes Volume Management

In today's business model, data is the most precious asset for many startups and enterprises. In a Kubernetes cluster, containers in Pods can be either data producers or data consumers. While some container data is expected to be transient and is not expected to outlive a Pod, other forms of data must outlive the Pod in order to be aggregated and possibly loaded into analytics engines. Kubernetes must provide storage resources in order to provide data to be consumed by containers or to store data produced by containers. Kubernetes uses **Volumes** of several types and a few other forms of storage resources for container data management. In this chapter, we will talk about **PersistentVolume** and **PersistentVolumeClaim** objects, which help us attach persistent storage Volumes to Pods.

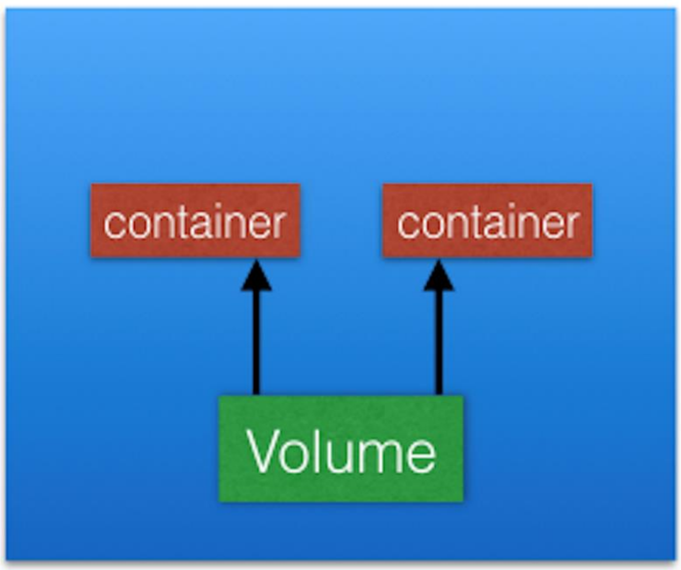
By the end of this chapter, you should be able to:

- Explain the need for persistent data management.
- Discuss Kubernetes Volume and its types.
- Discuss PersistentVolumes and PersistentVolumeClaims.

Volumes

As we know, containers running in Pods are ephemeral in nature. All data stored inside a container is deleted if the container crashes. However, the **`kubelet`** will restart it with a clean slate, which means that it will not have any of the old data.

To overcome this problem, Kubernetes uses Volumes. A Volume is essentially a directory backed by a storage medium. The storage medium, content and access mode are determined by the Volume Type.



Volumes

In Kubernetes, a Volume is attached to a Pod and can be shared among the containers of that Pod. The Volume has the same life span as the Pod, and it outlives the containers of the Pod - this allows data to be preserved across container restarts.

Volume Types

A directory which is mounted inside a Pod is backed by the underlying Volume Type. A Volume Type decides the properties of the directory, like size, content, default access modes, etc. Some examples of Volume Types are:

- **`emptyDir`**

  An **`empty`** Volume is created for the Pod as soon as it is scheduled on the worker node. The

Volume's life is tightly coupled with the Pod. If the Pod is terminated, the content of **`emptyDir`** is deleted forever.

- **`hostPath`**
  With the **`hostPath`** Volume Type, we can share a directory from the host to the Pod. If the Pod is terminated, the content of the Volume is still available on the host.
- **`gcePersistentDisk`**
  With the **`gcePersistentDisk`** Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod.
- **`awsElasticBlockStore`**
  With the **`awsElasticBlockStore`** Volume Type, we can mount an AWS EBS Volume into a Pod.
- **`azureDisk`**
  With **`azureDisk`** we can mount a Microsoft Azure Data Disk into a Pod.
- **`azureFile`**
  With **`azureFile`** we can mount a Microsoft Azure File Volume into a Pod.
- **`cephfs`**
  With **`cephfs`**, an existing CephFS volume can be mounted into a Pod. When a Pod terminates, the volume is unmounted and the contents of the volume are preserved.
- **`nfs`**
  With nfs, we can mount an NFS share into a Pod.
- **`iscsi`**
  With iscsi, we can mount an iSCSI share into a Pod.
- **`secret`**
  With the **`secret`** Volume Type, we can pass sensitive information, such as passwords, to Pods. We will take a look at an example in a later chapter.
- **`configMap`**
  With **`configMap`** objects, we can provide configuration data, or shell commands and arguments into a Pod.
- **`persistentVolumeClaim`**
  We can attach a PersistentVolume to a Pod using a **`persistentVolumeClaim`**. We will cover this in our next section.

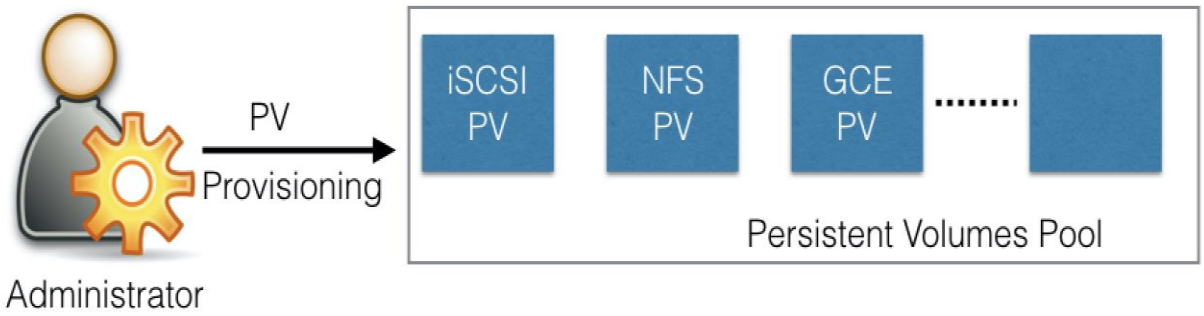
You can learn more details about Volume Types in the Kubernetes documentation.

PersistentVolumes

In a typical IT environment, storage is managed by the storage/system administrators. The end user will just receive instructions to use the storage but is not involved with the underlying storage management.

In the containerized world, we would like to follow similar rules, but it becomes challenging, given the many Volume Types we have seen earlier. Kubernetes resolves this problem with the **PersistentVolume (PV)** subsystem, which provides APIs for users and administrators to manage and consume persistent storage. To manage the Volume, it uses the PersistentVolume API resource type, and to consume it, it uses the PersistentVolumeClaim API resource type.

A Persistent Volume is a network-attached storage in the cluster, which is provisioned by the administrator.



PersistentVolume

PersistentVolumes can be dynamically provisioned based on the StorageClass resource. A StorageClass contains pre-defined provisioners and parameters to create a PersistentVolume. Using PersistentVolumeClaims, a user sends the request for dynamic PV creation, which gets wired to the StorageClass resource.

Some of the Volume Types that support managing storage using PersistentVolumes are:

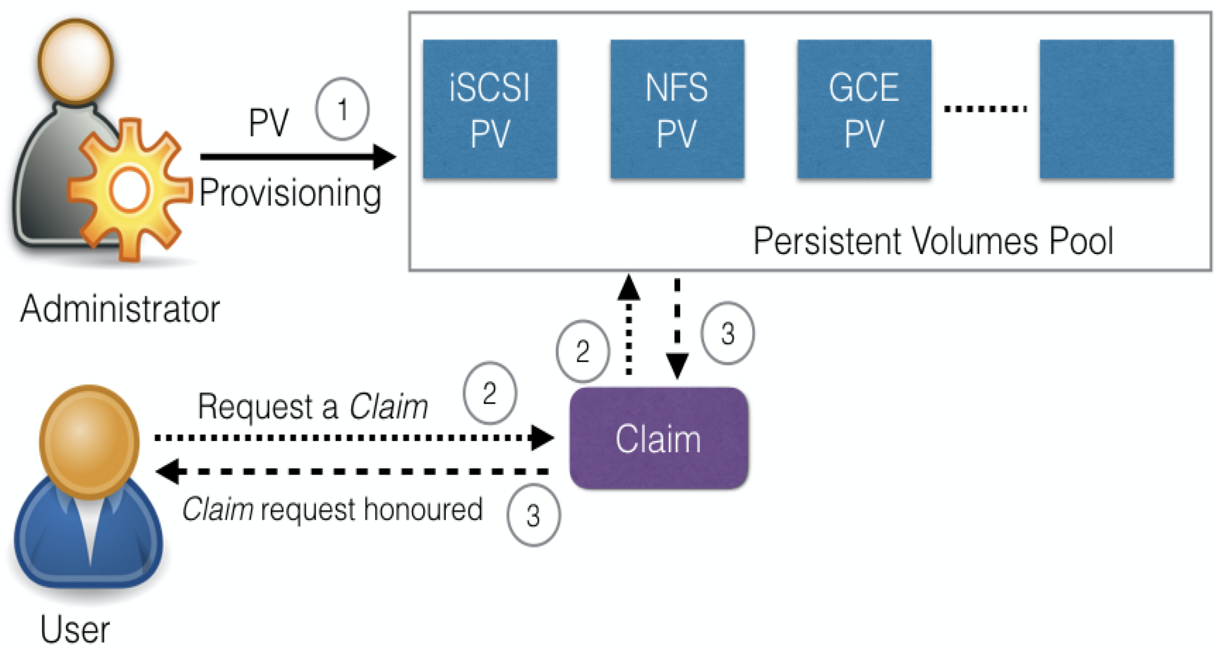
- GCEPersistentDisk
- AWSElasticBlockStore

- AzureFile
- AzureDisk
- CephFS
- NFS
- iSCSI.

For a complete list, as well as more details, you can check out the Kubernetes documentation.

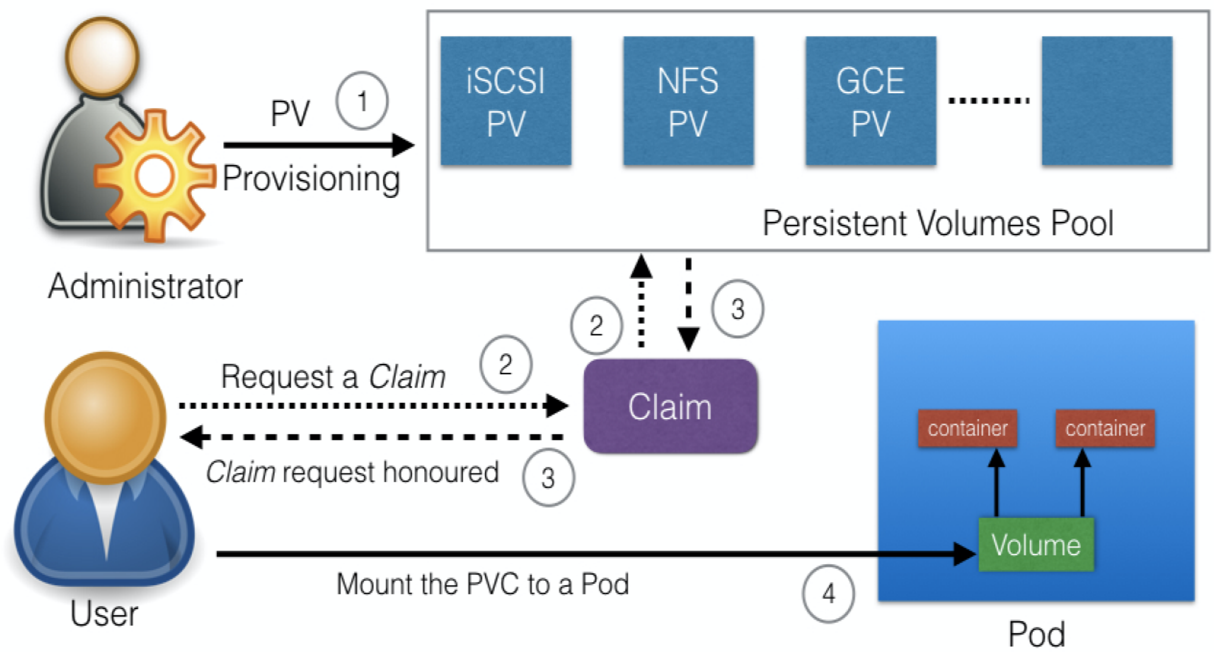
PersistentVolumeClaims

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. Users request for PersistentVolume resources based on type, access mode, and size. There are three access modes: ReadWriteOnce (read-write by a single node), ReadOnlyMany (read-only by many nodes), and ReadWriteMany (read-write by many nodes). Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.



PersistentVolumeClaim

After a successful bound, the PersistentVolumeClaim resource can be used in a Pod.



PersistentVolumeClaim Used In a Pod

Once a user finishes its work, the attached PersistentVolumes can be released. The underlying PersistentVolumes can then be reclaimed (for an admin to verify and/or aggregate data), deleted (both data and volume are deleted), or recycled for future usage (only data is deleted).

To learn more, you can check out the Kubernetes documentation.

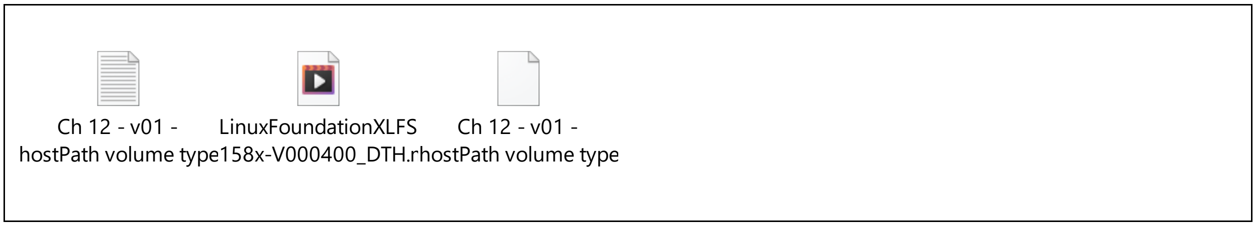
Container Storage Interface (CSI)

Container orchestrators like Kubernetes, Mesos, Docker or Cloud Foundry used to have their own methods of managing external storage using Volumes. For storage vendors, it was challenging to manage different Volume plugins for different orchestrators. Storage vendors and community members from different orchestrators started working together to standardize the Volume interface; a volume plugin built using a standardized CSI designed to work on different container orchestrators. You can find CSI specifications here.

Between Kubernetes releases v1.9 and v1.13 CSI matured from alpha to stable support, which makes installing new CSI-compliant Volume plugins very easy. With CSI, third-party storage providers can develop solutions without the need to add them into the core Kubernetes codebase.

Using a Shared hostPath Volume Type (Demo)

Using a Shared hostPath Volume Type



Chapter 13. ConfigMaps and Secrets

While deploying an application, we may need to pass such runtime parameters like configuration details, permissions, passwords, tokens, etc. Let's assume we need to deploy ten different applications for our customers, and for each customer, we need to display the name of the company in the UI. Then, instead of creating ten different Docker images for each customer, we may just use the template image and pass customers' names as runtime parameters. In such cases, we can use the **ConfigMap API** resource. Similarly, when we want to pass sensitive information, we can use the **Secret API** resource. In this chapter, we will explore ConfigMaps and Secrets.

By the end of this chapter, you should be able to:

- Discuss configuration management for applications in Kubernetes using ConfigMaps.
- Share sensitive data (such as passwords) using Secrets.

ConfigMaps

ConfigMaps allow us to decouple the configuration details from the container image. Using ConfigMaps, we pass configuration data as key-value pairs, which are consumed by Pods or any other system components and controllers, in the form of environment variables, sets of commands and arguments, or volumes. We can create ConfigMaps from literal values, from configuration files, from one or more files or directories.

Create a ConfigMap from Literal Values and Display Its Details

A ConfigMap can be created with the **`kubectl create`** command, and we can display its details using the **`kubectl get`** command.

Create the ConfigMap

```
$ kubectl create configmap my-config --from-literal=key1=value1
--from-literal=key2=value2
configmap/my-config created
```

Display the ConfigMap Details for `my-config`

```
$ kubectl get configmaps my-config -o yaml
apiVersion: v1
data:
  key1: value1
  key2: value2
kind: ConfigMap
metadata:
  creationTimestamp: 2019-05-31T07:21:55Z
  name: my-config
  namespace: default
  resourceVersion: "241345"
  selfLink: /api/v1/namespaces/default/configmaps/my-config
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

With the **`-o yaml`** option, we are requesting the **`kubectl`** command to spit the output in the YAML format. As we can see, the object has the **`ConfigMap kind`**, and it has the key-value pairs inside the data field. The name of **`ConfigMap`** and other details are part of the **`metadata`** field.

Create a ConfigMap from a Configuration File

First, we need to create a configuration file with the following content:

```
apiVersion: v1
kind: ConfigMap
metadata:
```

```
  name: customer1
data:
  TEXT1: Customer1_Company
  TEXT2: Welcomes You
  COMPANY: Customer1 Company Technology Pct. Ltd.
```

where we specify the **`kind`**, **`metadata`**, and **`data`** fields, targeting the **`v1`** endpoint of the API server.

If we name the file with the configuration above as **`customer1-configmap.yaml`**, we can then create the ConfigMap with the following command:

```
$ kubectl create -f customer1-configmap.yaml
configmap/customer1 created
```

Create a ConfigMap from a File

First, we need to create a file **`permission-reset.properties`** with the following configuration data:

```
permission=read-only
allowed="true"
resetCount=3
```

We can then create the ConfigMap with the following command:

```
$ kubectl create configmap permission-config --from-file=<path/to/>permission-reset.properties
configmap/permission-config created
```

Use ConfigMaps Inside Pods

As Environment Variables

Inside a Container, we can retrieve the key-value data of an entire ConfigMap or the values of specific ConfigMap keys as environment variables.

In the following example all the **myapp-full-container** Container's environment variables receive the values of the **`full-config-map`** ConfigMap keys:

```
...
  containers:
  - name: myapp-full-container
    image: myapp
    envFrom:
    - configMapRef:
      name: full-config-map
...
```

In the following example the **myapp-specific-container** Container's environment variables receive their values from specific key-value pairs from separate ConfigMaps:

```
...
  containers:
  - name: myapp-specific-container
    image: myapp
    env:
    - name: SPECIFIC_ENV_VAR1
      valueFrom:
        configMapKeyRef:
          name: config-map-1
          key: SPECIFIC_DATA
    - name: SPECIFIC_ENV_VAR2
      valueFrom:
        configMapKeyRef:
          name: config-map-2
          key: SPECIFIC_INFO
...
```

With the above, we will get the **`SPECIFIC_ENV_VAR1`** environment variable set to the value of **`SPECIFIC_DATA`** key from **`config-map-1`** ConfigMap, and **`SPECIFIC_ENV_VAR2`** environment variable set to the value of **`SPECIFIC_INFO`** key from **`config-map-2`** ConfigMap.

As Volumes

We can mount a **`vol-config-map`** ConfigMap as a Volume inside a Pod. For each key in the ConfigMap, a file gets created in the mount path (where the file is named with the key name) and the content of that file becomes the respective key's value:

```
...
  containers:
  - name: myapp-vol-container
    image: myapp
```

```
    volumeMounts:
    - name: config-volume
      mountPath: /etc/config
  volumes:
  - name: config-volume
    configMap:
      name: vol-config-map
...
```

For more details, please study the Kubernetes documentation.

Secrets

Let's assume that we have a *Wordpress* blog application, in which our **`wordpress`** frontend connects to the **`MySQL`** database backend using a password. While creating the Deployment for **`wordpress`**, we can include the **`MySQL`** password in the Deployment's YAML file, but the password would not be protected. The password would be available to anyone who has access to the configuration file.

In this scenario, the Secret object can help by allowing us to encode the sensitive information before sharing it. With Secrets, we can share sensitive information like passwords, tokens, or keys in the form of key-value pairs, similar to ConfigMaps; thus, we can control how the information in a Secret is used, reducing the risk for accidental exposures. In Deployments or other resources, the Secret object is *referenced*, without exposing its content.

It is important to keep in mind that the Secret data is stored as plain text inside **etcd**, therefore administrators must limit access to the API server and **etcd**. A newer feature allows for Secret data to be encrypted at rest while it is stored in **etcd**; a feature which needs to be enabled at the API server level.

Create a Secret from Literal and Display Its Details

To create a Secret, we can use the **`kubectl create secret`** command:

```
 $ kubectl create secret generic my-password --from-
literal=password=mysqlpassword
```

The above command would create a secret called **`my-password`**, which has the value of the **`password`** key set to **`mysqlpassword`**.

After successfully creating a secret we can analyze it with the **get** and **describe** commands. They do not reveal the content of the Secret. The type is listed as **Opaque**.

```
$ kubectl get secret my-password
NAME          TYPE     DATA   AGE
my-password   Opaque   1      8m
$ kubectl describe secret my-password
Name:          my-password
Namespace:     default
Labels:        <none>
Annotations:   <none>
Type  Opaque
Data
====
password:  13 bytes
```

Create a Secret Manually

We can create a Secret manually from a YAML configuration file. The example file below is named **mypass.yaml**. There are two types of maps for sensitive information inside a Secret: **data** and **stringData**.

With **data** maps, each value of a sensitive information field must be encoded using **base64**. If we want to have a configuration file for our Secret, we must first create the **base64** encoding for our password:

```
$ echo mysqlpassword | base64
bXlzcWxwYXNzd29yZAo=
```

and then use it in the configuration file:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
data:
  password: bXlzcWxwYXNzd29yZAo=
```

Please note that **base64** encoding does not mean encryption, and anyone can easily decode our encoded data:

```
$ echo "bXlzcWxwYXNzd29yZAo=" | base64 --decode
mysqlpassword
```

Therefore, make sure you do not commit a Secret's configuration file in the source code.

With `stringData` maps, there is no need to encode the value of each sensitive information field. The value of the sensitive field will be encoded when the `my-password` Secret is created:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-password
type: Opaque
stringData:
  password: mysqlpassword
```

Using the **mypass.yaml** configuration file we can now create a secret with `kubectl create` command:

```
$ kubectl create -f mypass.yaml
secret/my-password created
```

Create a Secret from a File and Display Its Details

To create a Secret from a File, we can use the **kubectl create secret** command.

First, we encode the sensitive data and then we write the encoded data to a text file:

```
 $ echo mysqlpassword | base64
bXlzcWxwYXNzd29yZAo=
 $ echo -n 'bXlzcWxwYXNzd29yZAo=' > password.txt
```

Now we can create the Secret from the `password.txt` file:

```
 $ kubectl create secret generic my-file-password --from-file=password.txt
 secret/my-file-password created
```

After successfully creating a secret we can analyze it with the **get** and **describe** commands. They do not reveal the content of the Secret. The type is listed as **Opaque**.

```
$ kubectl get secret my-file-password
NAME                TYPE     DATA   AGE
my-file-password    Opaque   1      8m

$ kubectl describe secret my-file-password
Name:           my-file-password
Namespace:      default
Labels:         <none>
Annotations:    <none>

Type  Opaque

Data
====
password.txt:  13 bytes
```

Use Secrets Inside Pods

Secrets are consumed by Containers in Pods as mounted data volumes, or as environment variables, and are referenced in their entirety or specific key-values.

Using Secrets as Environment Variables

Below we reference only the **password** key of the **my-password** Secret and assign its value to the **WORDPRESS_DB_PASSWORD** environment variable:

```
....
spec:
  containers:
  - image: wordpress:4.7.3-apache
    name: wordpress
    env:
    - name: WORDPRESS_DB_PASSWORD
      valueFrom:
        secretKeyRef:
          name: my-password
          key: password
....
```

Using Secrets as Files from a Pod

We can also mount a Secret as a Volume inside a Pod. The following example creates a file for each **`my-password`** Secret key (where the files are named after the names of the keys), the files containing the values of the Secret:

```
....
spec:
 containers:
 - image: wordpress:4.7.3-apache
   name: wordpress
   volumeMounts:
   - name: secret-volume
     mountPath: "/etc/secret-data"
     readOnly: true
 volumes:
 - name: secret-volume
   secret:
     secretName: my-password
....
```

For more details, you can study the Kubernetes documentation.

Using ConfigMaps (Demo)

Ch 13 - v01 - Using ConfigMap-en.txt

Ch 13 - v01 - Using ConfigMap-en.srt

LinuxFoundationXLFS158x-V000600_DTH.r

Chapter 14. Ingress

In an earlier chapter, we saw how we can access our deployed containerized application from the external world via *Services*. Among the *ServiceTypes* the NodePort and LoadBalancer are the most often used. For the LoadBalancer *ServiceType*, we need to have support from the underlying infrastructure. Even after having the support, we may not want to use it for every Service, as LoadBalancer resources are limited and they can increase costs significantly. Managing the NodePort *ServiceType* can also be tricky at times, as we need to keep updating our proxy settings and keep track of the assigned ports. In this chapter, we will explore the **Ingress** API resource, which represents another layer of abstraction, deployed in front of the *Service* API resources, offering a unified method of managing access to our applications from the external world.

By the end of this chapter, you should be able to:

- Explain what Ingress and Ingress Controllers are.
- Learn when to use Ingress.
- Access an application from the external world using Ingress.

Ingress I

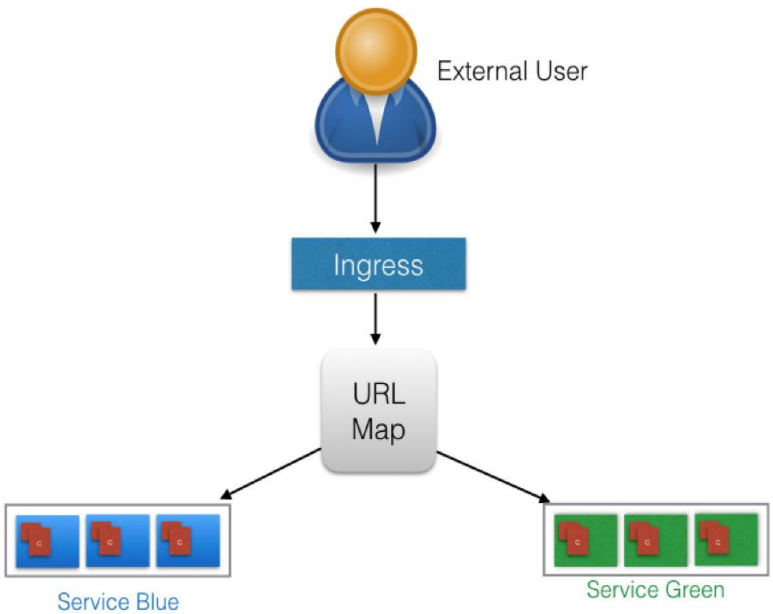
With Services, routing rules are associated with a given Service. They exist for as long as the Service exists, and there are many rules because there are many Services in the cluster. If we can somehow decouple the routing rules from the application and centralize the rules management, we can then update our application without worrying about its external access. This can be done using the **Ingress** resource.

According to kubernetes.io,

"An Ingress is a collection of rules that allow inbound connections to reach the cluster Services."

To allow the inbound connection to reach the cluster Services, Ingress configures a Layer 7 HTTP/HTTPS load balancer for Services and provides the following:

- TLS (Transport Layer Security)
- Name-based virtual hosting
- Fanout routing
- Loadbalancing
- Custom rules.



Ingress

Ingress II

With Ingress, users do not connect directly to a Service. Users reach the Ingress endpoint, and, from there, the request is forwarded to the desired Service. You can see an example of a sample Ingress definition below:

apiVersion: networking.k8s.io/v1beta1

kind: Ingress

metadata:

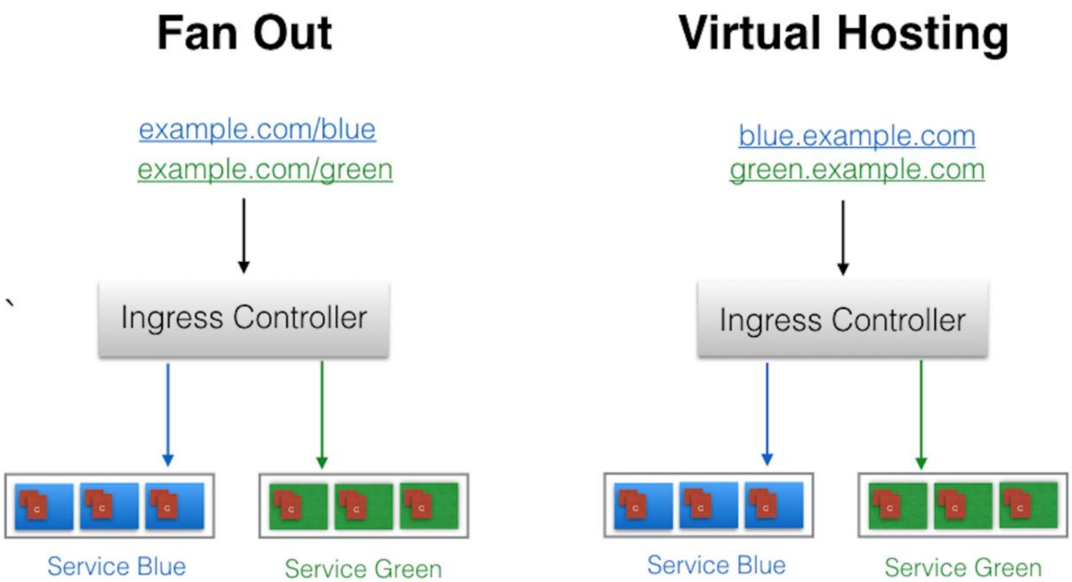
```
  name: virtual-host-ingress
  namespace: default
spec:
  rules:
  - host: blue.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-blue-svc
          servicePort: 80
  - host: green.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-green-svc
          servicePort: 80
```

In the example above, user requests to both **blue.example.com** and **green.example.com** would go to the same Ingress endpoint, and, from there, they would be forwarded to **webserver-blue-svc**, and **webserver-green-svc**, respectively. This is an example of a **Name-Based Virtual Hosting** Ingress rule.

We can also have **Fanout** Ingress rules, when requests to **example.com/blue** and **example.com/green** would be forwarded to **webserver-blue-svc** and **webserver-green-svc**, respectively:

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
```

```yaml
metadata:

  name: fan-out-ingress

  namespace: default

spec:

  rules:

  - host: example.com

    http:

      paths:
      - path: /blue

        backend:

          serviceName: webserver-blue-svc

          servicePort: 80

      - path: /green

        backend:

          serviceName: webserver-green-svc

          servicePort: 80
```



Ingress URL Mapping

The Ingress resource does not do any request forwarding by itself, it merely accepts the definitions of traffic routing rules. The ingress is fulfilled by an Ingress Controller, which we will discuss next.

Ingress Controller

An [Ingress Controller](https://kubernetes.io/docs/concepts/services-networking/ingress/#ingress-controllers) is an application watching the Master Node's API server for changes in the Ingress resources and updates the Layer 7 Load Balancer accordingly. Kubernetes supports different Ingress Controllers, and, if needed, we can also build our own. GCE L7 Load Balancer Controller and Nginx Ingress Controller are commonly used Ingress Controllers. Other controllers are Istio, Kong, Traefik, etc.

Start the Ingress Controller with Minikube

Minikube ships with the Nginx Ingress Controller setup as an addon, disabled by default. It can be easily enabled by running the following command:

```
$ minikube addons enable ingress
```

Deploy an Ingress Resource

Once the Ingress Controller is deployed, we can create an Ingress resource using the **`kubectl create`** command. For example, if we create a **`virtual-host-ingress.yaml`** file with the **Name-Based Virtual Hosting** Ingress rule definition that we saw in the *Ingress II* section, then we use the following command to create an Ingress resource:

```
$ kubectl create -f virtual-host-ingress.yaml
```

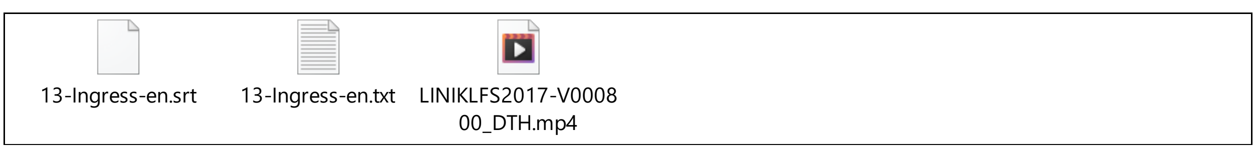
Access Services Using Ingress

With the Ingress resource we just created, we should now be able to access the **`webserver-blue-svc`** or **`webserver-green-svc`** services using the **`blue.example.com`** and **`green.example.com`** URLs. As our current setup is on Minikube, we will need to update the host configuration file (**`/etc/hosts`** on Mac and Linux) on our workstation to the Minikube IP for those URLs. After the update, the file should look similar to:

```
$ cat /etc/hosts
127.0.0.1        localhost
::1              localhost
192.168.99.100   blue.example.com green.example.com
```

Now we can open **`blue.example.com`** and **`green.example.com`** on the browser and access each application.

Using Ingress Rules to Access an Application (Demo)



Chapter 15. Advanced Topics

So far, in this course, we have spent most of our time understanding the basic Kubernetes concepts and simple workflows to build a solid foundation. To support enterprise-class production workloads, Kubernetes also supports auto-scaling, rolling updates, rollbacks, quota management, authorization through RBAC, package management, network and security policies, etc. In this chapter, we will briefly cover a limited number of such advanced topics, but diving into details would be out of scope for this course.

By the end of this chapter, you should be able to:

- Discuss advanced Kubernetes concepts: DaemonSets, StatefulSets, Helm, etc.

Annotations

With Annotations, we can attach arbitrary non-identifying metadata to any objects, in a key-value format:

```
"annotations": {
  "key1" : "value1",
  "key2" : "value2"
}
```

Unlike Labels, annotations are not used to identify and select objects. Annotations can be used to:

- Store build/release IDs, PR numbers, git branch, etc.
- Phone/pager numbers of people responsible, or directory entries specifying where such information can be found
- Pointers to logging, monitoring, analytics, audit repositories, debugging tools, etc.
- Etc.

For example, while creating a Deployment, we can add a description as seen below:

```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
  name: webserver
  annotations:
    description: Deployment based PoC dates 2nd May'2019
....
```

Annotations are displayed while describing an object:

```
$ kubectl describe deployment webserver
Name:                webserver
Namespace:           default
CreationTimestamp:   Fri, 03 May 2019 05:10:38 +0530
Labels:              app=webserver
Annotations:         deployment.kubernetes.io/revision=1
                     description=Deployment based PoC dates 2nd
May'2019
...
```

Jobs and CronJobs

A [Job](#) creates one or more Pods to perform a given task. The Job object takes the responsibility of Pod failures. It makes sure that the given task is completed successfully. Once the task is complete, all the Pods have terminated automatically. Job configuration options include:

- **parallelism** - to set the number of pods allowed to run in parallel;
- **completions** - to set the number of expected completions;
- **activeDeadlineSeconds** - to set the duration of the Job;
- **backoffLimit** - to set the number of retries before Job is marked as failed;
- **ttlSecondsAfterFinished** - to delay the clean up of the finished Jobs.

Starting with the Kubernetes 1.4 release, we can also perform Jobs at scheduled times/dates with [CronJobs](#), where a new Job object is created about once per each execution cycle. The CronJob configuration options include:

- **startingDeadlineSeconds** - to set the deadline to start a Job if scheduled time was missed;
- **concurrencyPolicy** - to *allow* or *forbid* concurrent Jobs or to *replace* old Jobs with new ones.

Quota Management

When there are many users sharing a given Kubernetes cluster, there is always a concern for fair usage. A user should not take undue advantage. To address this concern, administrators can use the ResourceQuota API resource, which provides constraints that limit aggregate resource consumption per Namespace.

We can set the following types of quotas per Namespace:

- **Compute Resource Quota**
  We can limit the total sum of compute resources (CPU, memory, etc.) that can be requested in a given Namespace.
- **Storage Resource Quota**
  We can limit the total sum of storage resources (PersistentVolumeClaims, requests.storage, etc.) that can be requested.
- **Object Count Quota**
  We can restrict the number of objects of a given type (pods, ConfigMaps, PersistentVolumeClaims, ReplicationControllers, Services, Secrets, etc.).

Autoscaling

While it is fairly easy to manually scale a few Kubernetes objects, this may not be a practical solution for a production-ready cluster where hundreds or thousands of objects are deployed. We need a dynamic scaling solution which adds or removes objects from the cluster based on resource utilization, availability, and requirements.

Autoscaling can be implemented in a Kubernetes cluster via controllers which periodically adjust the number of running objects based on single, multiple, or custom metrics. There are various types of autoscalers available in Kubernetes which can be implemented individually or combined for a more robust autoscaling solution:

- Horizontal Pod Autoscaler (HPA)
  HPA is an algorithm based controller API resource which automatically adjusts the number of replicas in a ReplicaSet, Deployment or Replication Controller based on CPU utilization.
- Vertical Pod Autoscaler (VPA)
  VPA automatically sets Container resource requirements (CPU and memory) in a Pod and dynamically adjusts them in runtime, based on historical utilization data, current resource availability and real-time events.
- Cluster Autoscaler
  Cluster Autoscaler automatically re-sizes the Kubernetes cluster when there are insufficient

resources available for new Pods expecting to be scheduled or when there are underutilized nodes in the cluster.

DaemonSets

In cases when we need to collect monitoring data from all nodes, or to run a storage daemon on all nodes, then we need a specific type of Pod running on all nodes at all times. A DaemonSet is the object that allows us to do just that. It is a critical controller API resource for multi-node Kubernetes clusters. The **`kube-proxy`** agent running as a Pod on every single node in the cluster is managed by a **`DaemonSet`**.

Whenever a node is added to the cluster, a Pod from a given DaemonSet is automatically created on it. Although it ensures an automated process, the DaemonSet's Pods are placed on nodes by the cluster's default Scheduler. When the node dies or it is removed from the cluster, the respective Pods are garbage collected. If a DaemonSet is deleted, all Pods it created are deleted as well.

A newer feature of the DaemonSet resource allows for its Pods to be scheduled only on specific nodes by configuring **`nodeSelectors`** and node **`affinity`** rules. Similar to Deployment resources, DaemonSets support rolling updates and rollbacks.

StatefulSets

The StatefulSet controller is used for stateful applications which require a unique identity, such as name, network identifications, strict ordering, etc. For example, **`MySQL cluster`**, **`etcd cluster`**.

The **`StatefulSet`** controller provides identity and guaranteed ordering of deployment and scaling to Pods. Similar to Deployments, StatefulSets use ReplicaSets as intermediary Pod controllers and support rolling updates and rollbacks.

Kubernetes Federation

With Kubernetes Cluster Federation we can manage multiple Kubernetes clusters from a single control plane. We can sync resources across the federated clusters and have cross-cluster discovery. This allows us to perform Deployments across regions, access them using a global DNS record, and achieve High Availability.

Although still an Alpha feature, the Federation is very useful when we want to build a hybrid solution, in which we can have one cluster running inside our private datacenter and another one in the public cloud, allowing us to avoid provider lock-in. We can also assign weights for each cluster in the Federation, to distribute the load based on custom rules.

Custom Resources

In Kubernetes, a **resource** is an API endpoint which stores a collection of API objects. For example, a Pod resource contains all the Pod objects.

Although in most cases existing Kubernetes resources are sufficient to fulfill our requirements, we can also create new resources using **custom resources**. With custom resources, we don't have to modify the Kubernetes source.

Custom resources are dynamic in nature, and they can appear and disappear in an already running cluster at any time.

To make a resource declarative, we must create and install a **custom controller**, which can interpret the resource structure and perform the required actions. Custom controllers can be deployed and managed in an already running cluster.

There are two ways to add custom resources:

- Custom Resource Definitions (CRDs)
  This is the easiest way to add custom resources and it does not require any programming knowledge. However, building the custom controller would require some programming.
- API Aggregation
  For more fine-grained control, we can write API Aggregators. They are subordinate API servers which sit behind the primary API server. The primary API server acts as a proxy for all incoming API requests - it handles the ones based on its capabilities and proxies over the other requests meant for the subordinate API servers.

Helm

To deploy an application, we use different Kubernetes manifests, such as Deployments, Services, Volume Claims, Ingress, etc. Sometimes, it can be tiresome to deploy them one by one. We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata. Such a bundle is referred to as *Chart*. These

Charts can then be served via repositories, such as those that we have for **`rpm`** and **`deb`** packages.

Helm is a package manager (analogous to **`yum`** and **`apt`** for Linux) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster.

Helm has two components:

- A client called *helm*, which runs on your user's workstation
- A server called *tiller*, which runs inside your Kubernetes cluster.

The client *helm* connects to the server *tiller* to manage Charts. Charts submitted for Kubernetes are available here.

Security Contexts and Pod Security Policies

At times we need to define specific privileges and access control settings for Pods and Containers. Security Contexts allow us to set Discretionary Access Control for object access permissions, privileged running, capabilities, security labels, etc. However, their effect is limited to the individual Pods and Containers where such context configuration settings are incorporated in the **`spec`** section.

In order to apply security settings to multiple Pods and Containers cluster-wide, we can define Pod Security Policies. They allow more fine-grained security settings to control the usage of the host namespace, host networking and ports, file system groups, usage of volume types, enforce Container user and group ID, root privilege escalation, etc.

Network Policies

Kubernetes was designed to allow all Pods to communicate freely, without restrictions, with all other Pods in cluster Namespaces. In time it became clear that it was not an ideal design, and mechanisms needed to be put in place in order to restrict communication between certain Pods and applications in the cluster Namespace. Network Policies are sets of rules which define how Pods are allowed to talk to other Pods and resources inside and outside the cluster. Pods not covered by any **Network Policy** will continue to receive unrestricted traffic from any endpoint.

Network Policies are very similar to typical Firewalls. They are designed to protect mostly assets located inside the Firewall but can restrict outgoing traffic as well based on sets of rules and policies.

The **Network Policy** API resource specifies **`podSelectors`**, *Ingress* and/or *Egress* **`policyTypes`**, and rules based on source and destination **`ipBlocks`** and **`ports`**. Very simplistic default allow or default deny policies can be defined as well. As a good practice, it is recommended to define a default deny policy to block all traffic to and from the Namespace, and then define sets of rules for specific traffic to be allowed in and out of the Namespace.

Let's keep in mind that not all the networking solutions available for Kubernetes support Network Policies. Review the Pod-to-Pod Communication section from the Kubernetes Architecture chapter if needed. By default, **Network Policies** are namespaced API resources, but certain network plugins provide additional features so that Network Policies can be applied cluster-wide.

Monitoring and Logging

In Kubernetes, we have to collect resource usage data by Pods, Services, nodes, etc., to understand the overall resource consumption and to make decisions for scaling a given application. Two popular Kubernetes monitoring solutions are the Kubernetes Metrics Server and Prometheus.

- **Metrics Server**
  Metrics Server is a cluster-wide aggregator of resource usage data - a relatively new feature in Kubernetes.
- **Prometheus**
  Prometheus, now part of CNCF (Cloud Native Computing Foundation), can also be used to scrape the resource usage from different Kubernetes components and objects. Using its client libraries, we can also instrument the code of our application.

Another important aspect for troubleshooting and debugging is Logging, in which we collect the logs from different components of a given system. In Kubernetes, we can collect logs from different cluster components, objects, nodes, etc. Unfortunately, however, Kubernetes does not provide cluster-wide logging by default, therefore third party tools are required to centralize and aggregate cluster logs. The most common way to collect the logs is using Elasticsearch, which uses fluentd with custom configuration as an agent on the nodes. **fluentd** is an open source data collector, which is also part of CNCF.

Chapter 16. Kubernetes Community

Just as with any other open source project, the **community** plays a vital role in the development of Kubernetes. The community decides the roadmap of the projects and works towards it. The community becomes engaged in different online and offline forums, like Meetups, Slack, Weekly meetings, etc. In this chapter, we will explore the Kubernetes community and see how you can become a part of it, too.

By the end of this chapter, you should be able to:

- Understand the importance of Kubernetes community.
- Learn about the different channels to interact with the Kubernetes community.
- List major CNCF events.

Kubernetes Community

With more than 53K GitHub stars, Kubernetes is one of the most popular open source projects. The community members not only help with the source code, but they also help with sharing the knowledge. The community engages in both online and offline activities.

Currently, there is a project called K8s Port, which recognizes and rewards community members for their contributions to Kubernetes. This contribution can be in the form of code, attending and speaking at meetups, answering questions on Stack Overflow, etc.

Next, we will review some of the mediums used by the Kubernetes community.

Weekly Meetings and Meetup Groups

Weekly Meetings
A weekly community meeting happens using video conference tools. You can request a calendar invite from here.

Meetup Groups
There are many meetup groups around the world, where local community members meet at regular intervals to discuss Kubernetes and its ecosystem.

There are some online meetup groups as well, where community members can meet virtually.

Slack Channels and Mailing Lists

Slack Channels
Community members are very active on the Kubernetes Slack. There are different channels based on topics, and anyone can join and participate in the discussions. You can discuss with the Kubernetes team on the **`#kubernetes-users`** channel.

Mailing Lists
There are Kubernetes users and developers mailing lists, which can be joined by anybody interested.

SIGs and Stack Overflow

Special Interest Groups
Special Interest Groups (SIGs) focus on specific parts of the Kubernetes project, like scheduling, authorization, networking, documentation, etc. Each group may have a different workflow, based on its specific requirements. A list with all the current SIGs can be found here.

Depending on the need, a new SIG can be created.

Stack Overflow
Besides Slack and mailing lists, community members can get support from Stack Overflow, as well. Stack Overflow is an online environment where you can post questions that you cannot find an answer for. The Kubernetes team also monitors the posts tagged Kubernetes.

CNCF Events

CNCF organizes numerous international conferences on Kubernetes, as well as other CNCF projects. For more information about these events, please click here.

Three of the major conferences it organizes are:

- KubeCon + CloudNativeCon Europe
- KubeCon + CloudNativeCon North America

- KubeCon + CloudNativeCon China.

What's Next on Your Kubernetes Journey?

Now that you have a better understanding of Kubernetes, you can continue your journey by:

- Participating in activities and discussions organized by the Kubernetes community
- Attending events organized by the Cloud Native Computing Foundation and The Linux Foundation
- Expanding your Kubernetes knowledge and skills by enrolling in the self-paced LFS258 - Kubernetes Fundamentals, LFD259 - Kubernetes for Developers, or the instructor-led LFS458 - Kubernetes Administration and LFD459 - Kubernetes for App Developers, paid courses offered by The Linux Foundation
- Preparing for the Certified Kubernetes Administrator or the Certified Kubernetes Application Developer exams, offered by the Cloud Native Computing Foundation
- And many other options.