O'REILLY®

Managing Kubernetes

OPERATING KUBERNETES CLUSTERS IN THE REAL WORLD

Brendan Burns & Craig Tracey

Managing Kubernetes

Operating Kubernetes Clusters in the Real World

Brendan Burns and Craig Tracey

Managing Kubernetes

by Brendan Burns and Craig Tracey

Printed in the United States of America.

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

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

- Editor: Virginia Wilson
- Production Editor: Justin Billing
- Copyeditor: FILL IN COPYEDITOR
- Proofreader: FILL IN PROOFREADER
- Indexer: FILL IN INDEXER
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Rebecca Demarest

- October 2018: First Edition

Revision History for the First Edition

- YYYY-MM-DD: First Release

See http://oreilly.com/catalog/errata.csp?isbn=9781492033912 for release details.

978-1-492-03391-2

[FILL IN]

Chapter 1. Introduction

Kubernetes is an open source orchestrator for deploying containerized applications. Kubernetes was open sourced by Google, inspired by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs.[1] and developed over the last four years by a vibrant community of open source contributors.

It is used by a large and growing number of developers to deploy reliable distributed systems, as well as run machine learning, big data and other batch workloads. A Kubernetes cluster provides an orchestration API that enables applications to be defined and deployed using a simple declarative syntax. Further, the Kubernetes cluster itself provides numerous online, self-healing control algorithms that repair applications in the presence of failures. Finally the Kubernetes API exposes concepts like Deployments that make it easier to perform zero-downtime updates of your software and Service load balancers that make it easy to spread traffic across a number of replicas of your service. Additionally Kubernetes provides tools for naming and discovery of services so that you can build loosely coupled microservice architectures. Kubernetes is widely used across public and private clouds as well as physical infrastructure.

This book is dedicated to the topic of managing a Kuberentes cluster. Whether you are managing your own cluster on your own hardware, part of a team managing a cluster for a larger organization or a Kubernetes user who wants to go beyond the APIs and learn more about the internals of the system, you will find that deepening your knowledge of how to manage Kubernetes makes you more capable at accomplishing all of the things you need Kubernetes to do for you.

Note

When we speak about a “cluster” we’re talking about a collection of machines that work together to provide the aggregate computing power that Kubernetes makes available to its end-users. A Kubernetes “cluster” is a collection of machines that are all controlled by a single API and can be used by consumers of that API.

When we talk about managing Kubernetes there are a variety of topics that make up the necessary skills for managing a cluster:

- How the cluster operates
- How to adjust, secure and tune the cluster
- How to understand your cluster and respond when things go wrong
- How to extend your cluster with new and custom functionality

How the cluster operates

Ultimately, if you are going to manage a system you need to understand how that system operates. What are the pieces that it is made up of and how do they fit together. Without at least a rough understanding of the components and how they inter-operate you are unlikely to be successful at managing any system. Managing a piece of software, especially one as complex as Kubernetes, without this understanding is like attempting to repair a car with out knowing how the tail-pipe relates to the engine. Its a bad idea.

However, in addition to understanding how all the pieces fit together, it's also essential to understand how the user consumes the Kubernetes cluster. Only by understanding how a tool like Kubernetes is intended to be used, can you truly understand the needs and demands required for successful management of that tool. To revisit our analogy of the car, without understanding the way in which a driver sits in the vehicle and guides it down the road, you are unlikely to be able to successfully manage the vehicle. The same is true of a Kubernetes cluster.

Finally, it is critical that you understand the role that the Kubernetes cluster plays in a users daily existence. What is the cluster accomplishing for the end user? What applications are they deploying on it? What complexity and hardship is the cluster removing? What complexity is the Kubernetes API adding? To complete the usage of our car analogy, in order to understand the importance of a car to it's end-user, it is critical to know that it is the thing that ensures a person shows up to work on time. Likewise with Kubernetes, if you don't understand that the cluster is the place where a user's mission critical application will be running, and the Kubernetes API is the thing that a developer will be relying on to fix a problem when something goes wrong at three am, then you don't really understand what is needed to successfully manage that cluster.

Adjust, secure and tune the cluster

In addition to understanding how the pieces of the cluster fit together, and how the Kubernetes API is used by developers to build and deploy their application, it is also critical to understand the various APIs and configuration options to adjust, secure and tune your cluster. A Kubernetes cluster, or really any significant piece of software, is not something that you simply turn up, start running and walk away.

The cluster and its usage have a lifecycle. Developers join and leave teams. New teams are formed and old ones die. The cluster scales with the growth of the business. New Kubernetes releases come out to fix bugs, add new features and improve stability. Increased demand on the cluster exposes performance problems that had previously been ignored. Responding to all of these changes in the lifespan of your cluster requires understanding the ways in which Kubernetes can be configured via command line flags, deployment options and API configuration.

Additionally, your cluster is not just a target for application deployment, it can also be a vector for attacking the security of your applications. Configuring your cluster to be secure against many different attacks from application compromises to denial of service is a critical component of sucessfully managing a cluster. Much of the time this hardening is in fact simply to prevent mistakes. In many cases, the value of hardening and security is it prevents one team or user from accidentally “attacking” another team’s service. However, additionally active attacks do sometimes happen, and the configuration of the cluster is critical to both detecting attacks when they occur as well as preventing them from occuring in the first place.

Finally, depending on the usage of the cluster, you may need to demonstrate a compliance to various standards for security that are required for developers of applications in a variety of different industries such as health-care, finance or government. Understanding how to build a compliant cluster means that Kubernetes can be put to work in such environments.

Responding when things go wrong

It would be a great world to live in, if things never went wrong. Sadly of course, that is not the world that we live in (at least not any computer system I've ever been a part of managing). What's critical when things go wrong is that you learn of it quickly. That you learn of it through automation and alerts rather than through a user informing you your system is down and that you are capable of responding and restoring the system as quickly as possible.

The first step in detecting when things break and understanding why they are broken is in having the right metrics in place. Fortunately, there are two technologies present in the Kubernetes cluster that makes this job easier. The first is that Kubernetes itself is generally deployed inside of containers. In addition to the value in reliable packaging and deployment, the container itself forms a boundary where basic metrics such as CPU, memory, network and disk usage can be observed. These metrics can then be recorded into a monitoring system for both alerting and introspection.

In addition to these container generated metrics, the Kubernetes code base itself has been instrumented with a significant number of application metrics. These metrics include things like the number of requests sent or received by various components, as well as the latency of those requests. These metrics are expressed using a format popularized by the Prometheus open source project (*https://prometheus.io*), and they can be easily collected and populated into Prometheus which can be used directly, or with other tools like Grafana for visualization and introspection.

Combined together, the baseline metrics from the operating system containers as well as the application metrics from Kubernetes itself provide a rich set of data that can be used both to generate alerts that tell you when the system isn't working properly, as well as the historical data necessary to debug and determine what went wrong and when.

Of course understanding the problem is only the first half of the battle, the next step that is necessary is responding and recovering from the problems with the system. Fortunately Kubernetes has been built in a decoupled, modular manner with minimal state in the system. This means that (generally) at any given time it is safe to restart any component in the system that may be overloaded or misbehaving. This modularity and indempotency means that once you determine the problem developing a solution is often as straight-forward as restarting a few applications.

Of course in some cases, something truly terrible happens, and in such cases, your only recourse is to restore the cluster from a disaster recovery back up somewhere. Of course that presumes that you have enabled such back-ups in the first place. In addition to all of the monitoring to show you what is happening, the alerts to tell you when something breaks and the play-books to tell you how to repair it. Successfully managing a cluster requires that you develop and exercise a disaster response and recovery procedure. Its important to remember that simply developing this plan is insufficient. You need to practice it regularly, or you will not be ready (and the plan itself may be flawed) in the presence of a real problem.

Extending the system with new and custom functionality

One of the most important strengths of the Kubernetes open source project has been the explosive growth of libraries, tools and platforms that build on, extend or otherwise improve the usage of a Kubernetes cluster.

There are tools like Spinnaker or Jenkins for continuous deployment. There are tools like Helm that make it easy to package and deploy complete applications. There are platforms like Deis that provide git-push style developer workflows. There are numerous functions-as-a-service (FaaS) platforms that build on top of Kubernetes to enable users to consume Kubernetes via simple functions. There are tools for automating the creation and rotation of certificates. There are service mesh technologies that make it easy to link and introspect a myriade of microservices.

All of these tools in the ecosystem can be used to enhance, extend and improve the Kubernetes cluster that you are managing. They can provide new functionality that make your users lives easier, and make the software that the deploy more robust and more manageable.

However, all of these tools can also make your cluster more unstable, less secure, more prone to failures. They can expose your users to immature, poorly supported software that feels like an “official” part of the cluster but solely serves to make the user’s life more difficult.

Part of managing a Kubernetes cluster is understanding how and when to add these tools, platforms and projects into the cluster. It requires exploration and understanding not just of what a particular project is attempting to accomplish, but the total set of similar solutions that exist in the ecosystem. Often times a user will come to you with a request for a particular tool based on some video or blog that they happened across. But in truth they are often asking for a capability like CI/CD or certificate rotation, not any specific project. The one they mention is simply the one they happened across in a web search.

It is your job as a cluster manager to act as a curator of such project, an editor and an advisor who can recommend alternate solutions, or determine if a particular project is a good fit for your cluster or if there is some other way of accomplishing the same goal for the end user.

Additionally, the Kubernetes API itself contains rich tools for extending and enhancing the API. A Kubernetes cluster is not limited solely to the APIs that come built into the cluster. Instead new APIs can be dynamically added and removed. In addition to the existing extensions mentioned above, sometimes the job of managing a Kubernetes cluster involves developing new code and new extensions that enhance your cluster in ways that were previously impossible. Part of managing a cluster may very well be developing new tooling, and of course, once developed, sharing that tooling with the growing Kubernetes ecosystem is a great way to give back to the community that brought you the Kubernetes software in the first place.

Summary

Managing a Kubernetes cluster is more than just the act of installing some software on a set of machines. Successful management requires understanding of how Kubernetes is put together and how it is put to use by the developers who are Kubernetes users. It requires that you understand how to maintain, adjust and improve the cluster over time as its usage patterns change. Additionally, you need to know how to monitor the information put off by the cluster in operation and develop the alerts and dashboards to tell you when the cluster is sick and how to make it healthy again. Finally you need to be able to know when and how to extend the Kubernetes cluster with other tools to make it even more useful to your users. We hope that within this book you will find answers and more for all of these topics, and at completion you will find yourself with the skills to be sucessful and *Managing Kubernetes*.

[1] Brendan Burns et al., “Borg, Omega, and Kubernetes: Lessons Learned from Three Container-Management Systems over a Decade,” *ACM Queue* 14 (2016): 70–93, available at *http://bit.ly/2vIrL4S*.

Chapter 2. An overview of Kubernetes

Building, deploying and managing applications on top of the Kubernetes API is a complex topic in its own right. It is beyond the scope of this book to give a complete understanding of the Kubernetes API in all of its detail. For those purposes, there are a number of books, such as Kubernetes Up and Running, and online resources which will give you the knowledge necessary to build an application on Kubernetes. If you are completely new to Kubernetes and interested in building applications on top of Kubernetes, we definitely recommend taking advantage of these resources to augment the information in this chapter.

On the other hand, if you are responsible for managing a Kubernetes cluster, or you have some high-level understanding of the Kubernetes API then this chapter provides an introduction to the basic concepts of Kubernetes and their role in the development of an application. If after reading this chapter you still feel uncomfortable having a conversation with your users about their use of Kubernetes we highly reccomend you avail yourself to these additional resources.

This chapter proceeds as follows: first we introduce the notion of containers and how containers can be used to package and deploy your application. Then we introduce the core concepts behind the Kubernetes API and finally conclude with some higher level concepts that Kubernetes has added to make specific activities easier.

Containers

Containers were popularized by Docker and enabled a revolution in the way in which developers package and deploy their applications. However along the way the very word “container” has taken on many different meanings to many different people. Because Kubernetes is a “container orchestrator” to understand Kubernetes, it’s important to understand what we mean when we say “container”.

In reality, a “container” is made up of two different pieces, and a group of associated features. A “container” is made up of

- A container image
- A set of operating system concepts that isolate a running process or processes.

Starting with the *container image*, it contains the application runtime which consists of binaries, libraries and other data needed to run the container. A developer can package up their application as a container image on their development laptop and have strong assurances that when that image is deployed and run in a different setting, be it another users laptop or a server in a datacenter, the container will behave exactly as it did on the developers laptop. This portability and consistent execution in a variety of environments is one of the primary values of container images.

When a container image is run, it is also executed using namespaces in the operating system. These namespaces contain the process and provide isolation for its process or processes from other things running on the machine. This isolation means for example, that each running container has its own separated file system (like a `chroot`), additionally each container has its own network and PID namespaces, meaning that process number 42 in one container is a different process than process number 42 in another container. There are many other namespaces within the kernel that separate various running containers from each other. Additionally CGroups (control groups) allow the isolation of resource usage like memory or CPU. Finally standard operating system security features like SELinux or AppArmour can also be used with running containers. Combined together, all of this isolation makes it more difficult for different processes running in different containers to interfere with each other.

Note

When we say “isolation” it is incredibly important to know that this isolation is in terms of resources like CPU, memory or files. Containers as implemented in Linux and Windows do *not* currently provide strong security isolation for different processes. Containers when combined with other kernel level isolation can provide reasonable security isolation for some use cases. However, in the general case, only hypervisor level security is strong enough to isolate truly hostile workloads.

In order to make all of this work, a number of different tools were built to help build and deploy

containerized applications.

The first is the container image builder. Typically the `docker` command line tool is used to build a container image. However, the image format has been standardized through the open container image (OCI) standard. This has enabled the development of other image builders, available via cloud API, CI/CD or new alternative tools and libraries.

The `docker` tool uses a `Dockerfile` which specifies a set of instructions for how to construct the container image. Full details of using the `docker` tool is beyond the scope of this book, but there are numerous resources available in books like Docker up and running or in online resources. If you have never built a container image before, put down this book right now go start reading about containers and come back when you have built a few container images.

Once a container image has been built we need a way to distribute that image from a user's laptop up to other users, the cloud or a private data center. This is where the *image registry* comes in. The image registry is an API for uploading and managing images. After an image has been built it is "pushed" to the image registry. Once the image is in the registry, it can be "pulled" or downloaded from that registry to any machine that has access to the registry. Every registry requires some form of authorization to push an image, but some registries are "public" meaning that once an image is pushed anyone in the world can pull and start running the image, while others are "private" and require authorization to pull an image as well. At this point there are registries as a service available from every public cloud, and there are open source registry servers which you can download and run in your own environment. Before you even begin to set up your Kubernetes cluster, its a good idea to figure out where you are going to store the images that you run in that cluster.

Once a developer has packaged their application as a container image and pushed it to a registry, it's time to use that container to deploy their application, and that's where container orchestration comes in.

Container Orchestration

Once you have a container image stored in a registry somewhere, you need to run it to create a working application. This is where a container orchestrator like Kubernetes comes into the picture. Kuberentes' job is to take a group of machines which provide resources like CPU, memory and disk and transform them into a container-oriented API that a developer can use to deploy their containers.

The Kubernetes API enables a developer to declare their desired state of the world. To make a declarative request which is something like "I want this container image to run and it needs 3 cores and 10 gigabytes of memory to run correctly". The Kubernetes system then looks throughout its fleet of machines and finds a good place for that container image to run, and *schedules* the execution of that container on that machine. From the developer's perspective they see their container image running, and more often than not they don't need to concern themselves with the specific location where their container is executing.

Of course running just a single container is neither that interesting, nor that reliable, so the Kubernetes API also provides easy ways to say "I want three copies of this container image running on different machines, each with 3 cores and 10 gigabytes of memory."

But the orchestration system is about more than scheduling containers to machines. In addition to scheduling containers, the Kubernetes orchestrator knows how to heal those containers if they fail. If the process inside your container crashes, Kubernetes will restart it. If you define custom health checks Kubernetes can use them to determine if your application is deadlocked and needs to be restarted (liveness checks) or if it should be part of a load-balanced service (readiness checks).

Speaking of load balancing, Kubernetes also provides API objects for defining a way to load-balance traffic between these various replicas Kubernetes provides a way to say "Please create this load balancer to represent these running containers" These load balancers are also given easy to discover names so that linking different services together within a cluster is easy.

Kubernetes also has objects that perform zero down-time rollouts, manage configurations, persistent volumes, secrets and much more. The following sections detail the specific objects in the Kubernetes API that make all of this possible.

The Kubernetes API

The Kubernetes API is a RESTful API based on HTTP and JSON and provided by an *API Server*. All of the components in Kubernetes communicate through the API. There are more details of this architecture in the following chapter. As an open source project, the Kubernetes API is always evolving, but the core objects have been stable for years and the Kubernetes community provides a strong deprecation policy that ensures that developers and operators don't have to change what they are doing with each revision of the system. Kubernetes provides an OpenAPI specification for the API as well as numerous different client libraries in a variety of different languages (*https://github.com/kubernetes-client*).

Basic objects: Pods, Replica Sets and Services

Although it has a large and growing number of objects in its API, Kubernetes began with a relatively small number of objects, and these are still the core of what Kubernetes does.

Pods

A `Pod` is the atomic unit of scheduling in a Kubernetes cluster. A Pod is made up of a collection of one or more running containers. (A "Pod" is a collection of whales, derived from Docker's whale logo). When we say a Pod is "atomic" what we mean is that all of the containers in a Pod are guaranteed to land on the same machine in the cluster. Pods also share many resources between the containers. For example, they all share the same network namespace, which means that each container in a Pod can see the other containers in the Pod on `localhost`. Pods also share the process and inter-process communication namespaces so that different containers can use tools like shared memory and signalling to coordinate between the different processes in the Pod.

This close grouping means that Pods are ideally suited for symbiotic relationships between their containers, such as a main serving container and a background data loading container. Keeping the container images separate generally makes it more agile for different teams to own or re-use the container images, but grouping them together in a Pod at runtime enables them to operate cooperatively.

When people first encounter Pods in Kubernetes sometimes they spring to the wrong assumptions, for example a user may see a Pod and think: "Ah yes, a frontend and a database server make up a Pod" but this is the generally the wrong level of granularity. To see why, consider that the Pod is also the unit of scaling and replication, which means that if you group your frontend and your database in the same container then you will replicate your database at the same rate you replicate your frontends. This is unlikely the way that you want to do things.

Pods also do things to keep your application running. If the process in a container crashes, Kubernetes automatically restarts it. Pods can also define application level health-checks that can provide a richer, application-specific way of determining if the Pod should be automatically restarted.

ReplicaSets

Of course, if you are deploying a container orchestrator just to run individual containers, you are probably over-complicating your life. In general one of the main reasons for container orchestration is to make it easier to build replicated, reliable systems. While individual containers may fail, or be incapable of serving the load of a system, replicating an application out to a number of different running containers dramatically reduces the probability that your service will completely fail at a particular moment in time, and horizontal scaling enables you to grow your application in response to load. In the Kubernetes API, this sort of stateless replication is handled by a `ReplicaSet` object. A replica set takes a Pod definition and a number of replicas and ensures

that that number of replicas exist within the system. The actual replication is handled by the Kubernetes “controller manager” which creates Pod objects that are scheduled by the Kubernetes “scheduler”. These details of the architecture are described in later chapters.

Note

ReplicaSets are a slightly newer object. At it’s v1 release Kubernetes had an API object called a ReplicationController. Due to the deprecation policy, ReplicationControllers continue to exist in the Kubernetes API but their usage is strongly discouraged in favor of using ReplicaSets

Services

Once you can replicate your application out using a ReplicaSet, the next logical goal is to create a load balancer to spread traffic to these different replicas. To accomplish this, Kubernetes has a `Service` object. A Service represents a TCP or UDP load balanced service. Every Service that is created, whether TCP or UDP gets three things:

- It’s own IP address
- A DNS entry in the Kubernetes cluster DNS
- Load-balancing rules that proxy traffic to the Pods which implement the service

When a Service is created it is assigned a fixed IP address. This IP address is virtual, it does not correspond to any interface present on the network, instead it is programmed into the network fabric as a load-balanced IP address when packets are sent to that IP they are load-balanced out to a set of Pods that implement the Service. The load-balancing that is performed can either be round-robin or deterministic based on (source IP, destination IP) tuples.

Given this fixed IP address, a DNS name is programmed into the Kubernetes cluster’s DNS server. This DNS address provides a semantic name (e.g. “frontend”) which is the same as the name of the Kubernetes `Service` object and which enables other containers in the cluster to discover the IP address of the Service load balancer.

Finally, the Service load balancing is programmed into the network fabric of the Kubernetes cluster so that any container that tries to talk to the Service IP address is correctly load-balanced to the corresponding pods. This programming of the network fabric is dynamic, so as Pods come and go due to failures or scaling of a ReplicaSet, the load balancer is constantly re-programmed to match the current state of the cluster. This means that clients can rely on connections to the Service IP address always resolving to a Pod that implements the Service.

Storage: Persistent Volumes, ConfigMaps and Secrets

A common question that comes up after an initial exploration of Kubernetes is “what about my files?”. With all of these containers coming and going within the cluster and landing on different machines, it’s difficult to understand how you should manage the files and storage you want to be associated with that running container. Fortunately Kubernetes provides several different API

objects to help you manage your files.

The first storage concept introduced in Kubernetes was the `Volume` which is actually a part of the Pod API. Within a Pod you can define a set of Volumes. Each volume can be one of a large number of different types. At present there are more than ten different types of volumes you can create, including `NFS`, `iSCSI`, `git`, cloud storage based volumes and more.

Note

Though the `Volume` interface was initially a point of extensibility via writing code within Kubernetes, the explosion of different volume types eventually showed how unsustainable this model is. Instead, new volume types are developed outside of the Kubernetes code and implement the container storage interface (CSI) an interface for storage that is independent of Kubernetes.

When you add a `Volume` to your Pod, you can chose to mount it to an arbitrary location in each running container. This enables your running container to have access to the storage within the volume. Different containers can mount these volumes at different locations, or ignore the volume entirely.

In addition to basic files, there are several types of Kubernetes objects that can themselves be mounted into your Pod as a Volume. The first of these is the `ConfigMap` object. A `ConfigMap` represents a collection of configuration files. In Kubernetes Configuration is separated from container images, because often you want to have different configurations for the same container image. When you add a ConfigMap based Volume to your Pod, the files in the config map show up in the specified directory in your running container.

A special type of configuration is some sort of secure information like a database password or certificate. Though these are essentially the same as configuration, it is useful to treat them as a separate type, because their handling is generally more restricted. Kubernetes uses the `Secret` type for such data. In the context of volumes, a Secret works identically to a ConfigMap. It can be attached to a Pod via a Volume and mounted into a running container for use.

Over the course of people deploying applications with Volumes, it became clear that the tight binding of Volumes to pods was actually problematic. For example, when creating a replicated container (via a ReplicaSet) the same exact volume must be used by all replicas. In many situations this is acceptable, but in some cases you want a different volume for each replica. Additionally, specifying a precise volume type (e.g. an Azure disk persistent volume) binds your Pod definition to a specific environment (in this case the Microsoft Azure cloud), but it is often desirable to have a Pod definition that requests a generic type of storage (e.g. 10 gigabytes of network storage) without specifying a provider. To accomplish this, Kubernetes introduced the notion of `PersistentVolumes` and `PersistentVolumeClaims`. Instead of binding a Volume directly into a Pod, a `PersistentVolume` is created as a separate object, this object is then claimed to a specific Pod by a `PersistentVolumeClaim` and finally mounted into the Pod via this claim. At first blush this seems overly complicated, but the abstraction of Volume and Pod enables both the portability and automatic volume creation required by the two previous use cases.

Organizing your cluster with namespaces, labels and annotations

The Kubernetes API makes it quite easy to create a large number of objects in the system, and such a collection of objects can easily make administering a cluster a nightmare. Fortunately, Kubernetes also has a number of objects that make it easier to manage, query and reason about the objects in your cluster.

Namespaces

The first object for organizing your cluster is a `Namespace` you can think of a namespace as something like a folder for your Kubernetes API objects. Namespaces provide directories for container most of the other objects in the cluster. Namespaces also can provide a scope for role based access control (RBAC) rules. Like a folder, when you delete a namespace, all of the objects within that namespace are also destroyed, so be careful when you delete a namespace! Every Kubernetes cluster has a single built in namespace named `default` and most installations of Kubernetes also include a namespace named `kube-system` where cluster administration containers are created.

Note

Kubernetes objects are devided into “namespaced” and “non-namespaced” objects depending on whether they can be placed in a namespace. Most common Kubernetes API objects are namespaced objects. But some objects that apply to an entire cluster, for example `Namespace` objects themselves, or cluster level RBAC, are not namespaced.

In addition to organizing Kubernetes objects, namespaces are also placed into the DNS names created for services and the DNS search paths that are provided to containers. The complete DNS name for a Service is something like: `my-service.svc.my-namespace.cluster.internal` which means that two different services in different namespaces will end up with different FQDNs. Additionally, the DNS search paths for each container include the namespace, thus a DNS lookup for `frontend` will be translated to `frontend.svc.foo.cluster.internal` for a container in the `foo` namespace and `frontend.svc.bar.cluster.internal` for a container in the `bar` namespace.

Labels and Label Queries

Every object in the Kubernetes API can have an arbitrary set of *labels* associated with the object. Labels are string key/value pairs which help identify the object. For example a label might be “role”: “frontend”, which indicates that the object is a frontend. These labels can be used to query and filter objects in the API. For example, you can request that the API server provide you with a list of all Pods where the label “role” is “backend”. These requests are called “label queries” or “label selectors”. Many objects within the Kubernetes API use label selectors as a way to identify of set of objects that they apply to. For example a Pod can have a “node selector”

which identifies the set of nodes on which the Pod is elegible to run (nodes with GPUs for example). Likewise a Service has a “pod selector” which identifies the set of Pods that the Service should load balance traffic to. Labels and label selectors are the fundamental manner in which Kubernetes loosely couples its objects together.

Annotations

Not every metadata value that you want to assign to an API object is identifying information. Some of the information is simply an *annotation* about the object itself. Thus every Kubernetes API object can also have arbitrary annotations. These annotations might include something like the icon to display next to the object, or a modifier that changes the way that the object is interpretted by the system.

Often times experimental or vendor-specific features in Kubernetes are initially implemented using annotations, since they are not part of the formal API specification. In the case of such annotations, the annotation itself should carry some notion of the stability of the feature (e.g. `beta.kubernetes.io/activate-some-beta-feature`).

Advanced concepts: StatefulSets, Ingress and Deployments

Of course, simple replicated, load-balanced services are not the only style of application that you might want to deploy in containers, and as Kubernetes has evolved, it has added new API objects to better suit more specialized use cases, including improved rollouts, HTTP-based load balancing and routing, and stateful workloads.

Deployments

Though ReplicaSets are the primitive for running many different copies of the same container image, applications are not static entities. They evolve as developers add new features and fix bugs. This means that the act of rolling out new code to a service is as important a feature as replicating it to reliably handle load.

The `Deployment` object was added to the Kubernetes API to represent this sort of safe rollout from one version to another. A Deployment can hold pointers to multiple ReplicaSets, (e.g. `v1` and `v2`) and it can control the slow and safe migration from one ReplicaSet to another.

To understand how a Deployment works, imagine that you have an application which is deployed to three replicas in a ReplicaSet named `rs-v1`. When you ask a Deployment to rollout a new image (`v2`) the Deployment creates a new ReplicaSet (`rs-v2`) with a single replica. The Deployment waits for this replica to becomes healthy, and when it is the Deployment reduces the number of replicas in `rs-v1` to two. It then increases the number of replicas in `rs-v2` to two also, and waits for the second replica of `v2` to become healthy. This process continues until there are no more replicas of `v1` and there are three healthy replicas of `v2`.

Note

Deployments feature a large number of different knobs that can be tuned to provide a safe rollout for the specific details of an application. Indeed in most modern clusters, users exclusively use Deployment objects and don’t manage ReplicaSets directly.

HTTP load balancing with Ingress

While Service objects provide a great way to do simple TCP level load balancing, they don’t provide an Application level way to do load-balancing and routing. The truth is that most of the applications that users deploy using containers and Kubernetes are HTTP web-based applications. These applications are better served by a load-balancer that understands HTTP. To address these needs, the `Ingress` API was added to Kubernetes. Ingress represents a path and host-based HTTP load balancer and router. When you create an Ingress object, it receives a virtual IP address just like a Service, but instead of the 1-1 relationship between a service IP address and a set of Pods, an Ingress can use the content of an HTTP request to route requests to different

services.

To get a clearer understanding of how Ingress works, imagine that I have two Kubernetes services named “foo” and “bar”. Each has it’s own IP address, but I really want to expose them to the internet as part of the same host. For example, `foo.company.com` and `bar.company.com`. I can do this, by creating an Ingress object and associating its IP address with both the `foo.company.com` and `bar.company.com` DNS names. In the Ingress object, I also map the two different hostnames to the two different Kubernetes services. That way, when a request for `https://foo.company.com` is received, it is routed to the `foo` service in the cluster, and similarly for `https://bar.company.com`. With Ingresss the routing can be based on either host, or path or both, so `https://company.com/bar` can also be routed to the `bar` service.

Note

The Ingress API is one of the most decoupled and flexible APIs in Kubernetes. By default while Kubernetes will store Ingress objects, nothing will happen when they are created. Instead you need to also run an “ingress controller” in the cluster to take appropriate actions when the Ingress object is created. One of the most popular ingress controllers is the `nginx` ingress controller, but there are numerous different implementations that use other HTTP load balancers, or use cloud or physical load-balancer APIs

StatefulSets

Many applications are happily replicated horizontally and treated as identical clones of the same application. Each replica really has no unique identity independent of any other. For representing such applications a Kubernetes ReplicaSet is a perfect object. However, some applications, especially stateful storage workloads, or sharded applications require more differentiation between the different replicas in the application. While it is possible to add this differentiation at the application level on top of a ReplicaSet, doing so is complicated, error-prone and repetitive for end-users.

To resolve this, Kubernetes has recently introduced StatefulSets as a compliment to ReplicaSets, but for more stateful workloads. Like ReplicaSets, StatefulSets create multiple instances of the same container image running in a Kubernetes cluster, but the manner in which containers are created and destroyed is more deterministic, as are the names of each container.

In a ReplicaSet, each replicated Pod receives a name that involves a random hash (e.g. “frontend-14a2”), importantly there is no notion of ordering in a ReplicaSet. In contrast, with StatefulSets each replica receives a monotonically increasing index (e.g. “backed-0”, “backend-1” and so on).

Further, StatefulSets guarantee that replica zero will be created and become healthy before replica one is created and so forth. When combined this means that applications can easily bootstrap themselves using the initial replica (e.g. “backend-0”) as a bootstrap master. All subsequent replicas can rely on the fact that “backend-0” has to exist. Likewise when replicas are removed from a StatefulSet they are removed at the highest index. If a StatefulSet is scaled down from five to four replicas it is guaranteed that the fifth replica is the one that will be removed.

Additionally, StatefulSets receive DNS names so that each replica can be accessed directly, in addition to the complete StatefulSet. This allows clients to easily target specific shards in a sharded service.

Batch workloads: Job and ScheduledJob

In addition to stateful workloads, another specialized class of workloads are batch or one-time workloads. In contrast to all of the other workloads discussed previously, these workloads are not contstantly serving traffic. Instead they come into existence to perform some computation and are destroyed when the computation is complete.

In Kubernetes, a `Job` represents a set of tasks that need to be run. Like ReplicaSets and StatefulSets, Jobs operate by creating Pods to execute work by running container images. However, unlike Replica and StatefulSets, the Pods created by a Job only run until they complete and exit. In addition to the definition of the Pod to create, a Job contains a desired number of repetitions of the Job that should be run, as well as the maximum number of Pods to create in parallel. For example, a job with 100 repetitions and a maximum parallelism of 10 will run 10 pods simultaneously, creating new Pods as old ones complete, until there have been 100 successful executions of the container image.

ScheduledJobs build on top of the Jobs object by adding a schedule to a Job. A scheduled Job contains the definition of the Job object that you want to create, as well as the schedule on which that Job should be created.

Cluster agents and utlities: DaemonSets

One of the most common questions that comes up when people are moving to Kubernetes is: “how do I run my machine agents?” Example of such agents include things like intrusion detection, logging and monitoring and others. Many people attempt to use non-Kubernetes approaches to enable these agents such as adding new `systemd` unit files, or initialization scripts. While these approaches can work, they have several downsides. The first is that the resources used by these agents is not accounted for in Kubernetes’ accounting of resources in use on the cluster. The second is that all of the utility of container images and Kubernetes APIs for health checking, monitoring and more can not be applied to these agents. Fortunately, Kubernetes makes the `DaemonSet` API available to users to install such agents on their clusters. A DaemonSet provides a template for a Pod that should be run on every machine. When a DaemonSet is created, Kubernetes ensures that this Pod is running on each node in the cluster. If at some later point a new node is added, Kubernetes creates a Pod on that node as well. While by default Kubernetes places a Pod on every node in the cluster, a DaemonSet can also provide a node selector label query, and Kubernetes will only place that DaemonSet’s pods onto nodes that match that label query.

Summary

The goals of this book are to teach you how to successfully manage a Kubernetes cluster. But to successfully manage any service, you need to understand what that service makes available to the end user as well as how the user uses the service that you are managing. In this case we are delivering a reliable Kubernetes API to developers. These developers in turn are using this API to successfully build and deploy their applications. Understanding the various parts of the Kubernetes API will enable you to understand your end-users and do a better job managing the system that they rely on for their daily activities. This chapter is really an appreviated summary of topics that are covered in multiple hundred pages books like Kubernetes Up and Running as well as the core Kubernetes website (*https://kubernetes.io*). Readers who are interested in going more deeply into the Kubernetes API are strongly recomended to learn more from these resources.

Chapter 3. Kubernetes Architecture

Though Kubernetes is intended to make it easier to deploy and manage distributed systems. It itself is a distributed system that needs to be managed. To be able to do that, an operator needs to have a strong understanding of the system architecture. The role of each piece in the system and how they all fit together.

Concepts

To understand the architecture of Kubernetes, it is helpful, at first, to have an understanding of the concepts and design principals that governed the development of Kubernetes. Though the system can seem quite complex, it is actually based on a relatively small number of concepts that are repeated throughout this system. This allows Kubernetes to grow, while still remaining approachable to operators, learning about one component in the system often can be directly applied to other pieces of the system.

Declarative Configuration

The notion of declarative configuration is one of the primary drivers behind the development of Kubernetes. When we say declarative configuration, what we mean is that a user of Kubernetes declares a desired state of the world, for example, a user might say to Kubernetes “I want there to be five replicas of my web server running at all times” Kubernetes in turn takes that declarative statement and takes responsibility for ensuring that it is true. Of course, Kubernetes is sadly unable to understand natural language instructions and so that declaration is actually in the form of a structured YAML or JSON document.

Declarative configuration is in contrast to imperative configuration, with imperative configuration the user takes a series of direct actions, for example creating each of the five replicas that they want to have up and running. Imperative actions are often simpler to understand, one can simply say “run this” instead of understanding a more complex declarative syntax. However, the power of a declarative approach is that you are giving the system more than a sequence of instructions, you are giving it a declaration of your desired state. Because Kubernetes understands your desired state, it can take autonomous action independent of user interaction. This means that it can implement automatic self-correcting and self-healing behaviors. For an operator this is critical, since it means that the system can fix itself without waking you up in the middle of the night.

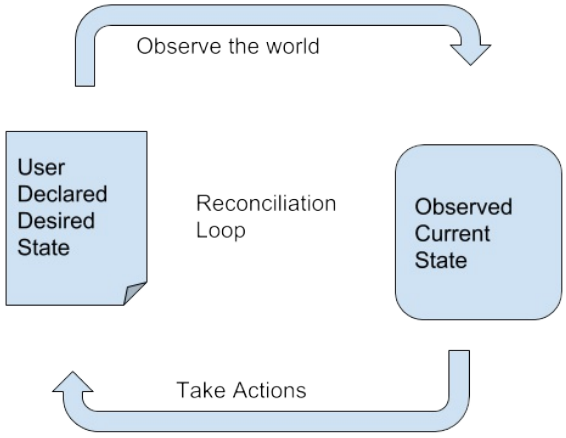
Reconciliation or controllers

To achieve these self-healing or self-correcting behaviors, Kubernetes is structured in terms of a large number of independent reconciliation or control loops. When designing a system like Kubernetes there are generally two different approaches that you can take; a monolithic state-based approach, or a decentralized controller based approach.

In the monolithic system design, the system is aware of the entire state of the world, and using this complete view to move everything forward in a coordinated fashion. This can be very attractive since the operation of the system is centralized and thus easier to understand. The problem with the monolithic approach is that it is not nearly as stable. If anything unexpected or unplanned happens, the entire system comes crashing down. Kubernetes takes an alternative decentralized approach in its design. Instead of single monolithic controller, Kubernetes is composed of a large number of controllers, each performing its own independent reconciliation loop. Each individual loop is only responsible for a small piece of the system, for example updating the list of endpoints for a particular load balancer, and each small controller is wholly unaware of the rest of the world. This focus on a small problem and the corresponding ignorance of the broader state of the world makes the entire system significantly more stable. Each controller is largely independent of all others, and thus unaffected by problems or changes unrelated to itself. The downside, though, of this distributed approach is that the overall behavior of the system can be harder to understand, since there is no single location to look for an explanation of why the system is behaving the way that it does. Instead it is necessary to look at the interoperation of a large number of independent processes.

In addition to understanding how a distributed architecture of many different control loops make Kubernetes more stable and flexible, the actual operation of the control loop is a pattern that is repeated throughout Kubernetes. The basic idea behind a control loop is that it is continually repeating the following steps:

- Obtain the desired state of the world.
- Observe the world
- Finding differences between the observation of the world and the desired state of the world
- Taking actions to make the observation of the world match the desired state.



The easiest example to help you understand the operation of a reconciliation control loop is the thermostat in your home. It has a desired state (the temperature that you entered on the thermostat), it makes observations of the world (the current temperature of your house), it finds the difference between these values and then takes actions (either heating or cooling) to make the real world match the desired state of the world.

The controllers in Kubernetes do the same thing, they observe the desired state of the world, via the declarative statements that are made to the Kubernetes API server. For example, a user might declare “I want four replicas of that web server.”, the Kubernetes replication controller takes this desired state and then observes the world. It might see that there are currently three replicas of the web serving container. The controller finds the difference between the current and desired state, (one missing web server) and then takes action to make the current state match the desired state by creating a fourth web serving container.

Of course, one of the challenges of managing this declarative state is finding the set of web servers that the reconciliation control loop should be paying attention to. This is where labels and label queries enter the Kubernetes design.

Implicit or Dynamic Grouping

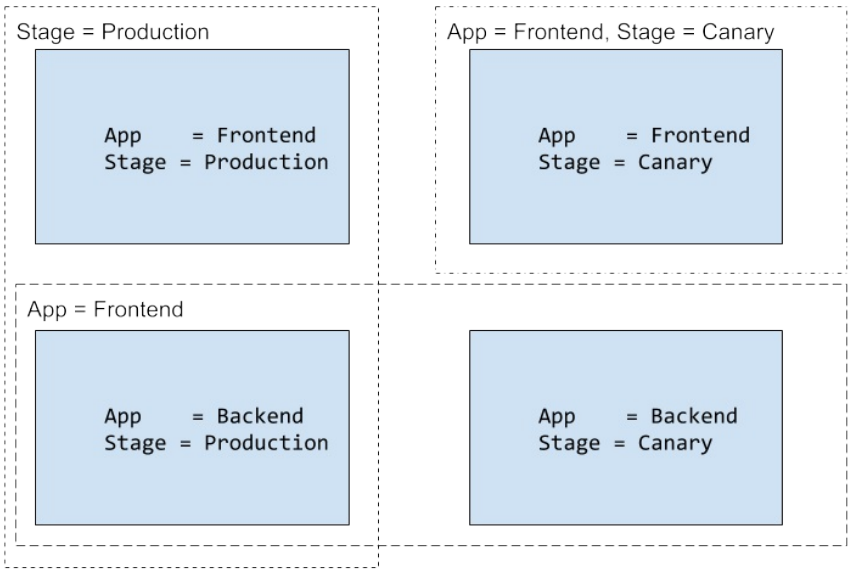
Whether it is grouping together a set of replicas, or identifying the backends for a load balancer, there are numerous times in the implementation of Kubernetes that it is necessary to identify a set of things. When grouping things together into a set there are two possible approaches: explicit/static or implicit/dynamic grouping. With static grouping, every group is defined by a concrete list, for example “the members of my team are Alice, Bob and Carol”, the list explicitly calls out the name of each member of the group, and the list is static, the membership doesn’t change unless the list itself changes. Much like a monolithic approach to design, the advantages of this static grouping is that it is easily understandable. To know who is in a group, one simply has to read the list. The challenge with static grouping is that it is inflexible, it can not respond to a dynamically changing world. Hopefully at this point, you know that Kubernetes choses a more dynamic approach to grouping. In Kubernetes groups are implicitly defined.

The alternative to explicit static groups, is implicit, dynamic groups. With implicit groups, instead of the list of members, the group is defined by a statement like “the members of my team are the people wearing orange” This group is implicitly defined. Nowhere in the definition of the group are the members defined, instead they are implied by evaluating the group definition against a set of people who are present. Because the set of people who are present can always change, the membership of the group is likewise dynamic and changing. While this can introduce complexity, because a second step (in the example case, looking for people wearing orange) it is also significantly more flexible and stable, it can handle a changing environment without requiring constant adjustments to static lists.

In Kubernetes, this implicit grouping is achieved via labels and label queries or label selectors. Every API object in Kubernetes can have an arbitrary number of key/value pairs called “labels” that are associated with the object. You can then use a label query or label selector to identify a set of objects that match that query. A concrete example of this is shown below.

Note

Every Kubernetes object has both labels and annotations. Initially they might seem redundant, but their uses are different. Labels can be queried and should be used for information that serves to identify the object in some way. Annotations can not be queried and should be used for general metadata about the object, that doesn’t represent its identity (for example, the icon to display next to the object when it is rendered graphically).


Stage = Production
App = Frontend
Stage = Production
App = Frontend, Stage = Canary
App = Frontend
Stage = Canary
App = Frontend
App = Backend
Stage = Production
App = Backend
Stage = Canary

Structure

Now that you have some sense for the design concepts that are implemented in the Kubernetes system, we'll now consider the design principles for how Kubernetes is built. There are two (and probably more) fundamental tenets that guide the design of Kubernetes.

Unix philosophy of many components

Kubernetes ascribes to the general Unix philosophy of modularity and of small pieces that do their jobs well. Kubernetes is not a single monolithic application that implements all of the various functionality in a single binary. Instead it is a collection of different applications, that all work together, largely ignorant of each other, to implement the overall system known as “kubernetes.” Even when there is a binary (for example the controller-manager) which groups together a large number of different functions into a single binary. Those functions are held almost entirely independently from each-other in that binary. They are compiled together largely to make the task of deploying and managing Kubernetes easier, not because of any tight binding between the components.

Again, the advantage of this modular approach is that Kubernetes is flexible, large pieces of functionality can be ripped out and replaced, without the rest of the system noticing or caring. The downside, of course, comes in the form of complexity, where deploying, monitoring and understanding the system requires integrating information and configuration across a number of different tools.

Though at times pieces of Kubernetes are compiled together, they rarely if ever work directly together, prefering instead to integrate via API calls and declarative state read or written from the API server.

API driven interactions

The second structural design within Kubernetes is that all interaction between components is driven through a centralized API surface area. An important corrollary of this design is that the API that the components use is the exact same API that is used by every other cluster user. This has two important consequences for the design of Kubernetes. The first is that no part of the system is more privileged or has more direct access to “internals” than any other. Indeed with the exception of the API-server that implements the API no one has access to the internals at all. Thus, every component can be swapped for an alternative implementation, and new functionality can be added without re-architecting the core components. As we will see in later chapters, even core components like the scheduler can be swapped out and replaced (or merely augmented) with alternative implementations.

The other important design consequence of the API driven interactions is that the system is inherently designed for stable operation in the presence of version skew. The simple truth is that when you roll out a distributed system to a grouping of machines, for a period of time you are going to have both the older version and the new version of the software running simultaneously. If you haven’t planned directly for this version skew, then the unplanned (and often untested) interactions between old and new versions can cause instability and outages. Because in Kubernetes everything is mediated through the API, and the API provides strongly defined API versions and conversion between different version numbers, the problems of version skew can largely be avoided. Though in reality, occasional problems can still crop-up and version skew and upgrade testing is an important part of Kubernetes release qualification.

Components

With a knowledge of both the concepts and structure in the Kubernetes architecture, we can now discuss the individual components that make up Kubernetes. This is a glossary of sorts, or a world map that you can refer to when you need a total perspective of how the various pieces of the Kubernetes system fit together. Some of the components are quite significant, and thus are covered in much more detail in later chapters, but this reference guide will help ground and contextualize those later explorations.

Worker and Head nodes and scheduled components.

Kubernetes is a system which groups together a large fleet of machines into a single unit that can be consumed via an API, but the implementation of Kubernetes actually sub-divides the set of machines into two groups: worker nodes and head nodes. Most of the components that make up the Kubernetes infrastructure run on head or “control plane” nodes. There are a limited number of such nodes in a cluster, general one, three or five nodes. These nodes run the components that implement Kubernetes like etcd and the apiserver. There is an odd number of these nodes, since they need to keep quorum in a RAFT/Paxos shared state implementation. The actual work done by the cluster is done on the worker nodes. These nodes also run a more limited selection of Kubernetes components, finally there is a selection of Kubernetes components that are actually scheduled to the Kubernetes cluster once it is created. From a Kubernetes perspective, these components are indistinguishable from other workloads, but they do implement part of the overall Kubernetes API.

The description of the Kubernetes components below breaks them into these three groupings. The components that run on head nodes, the components that run on all nodes and the components that run scheduled onto the cluster.

Head node components

A head node is the brains of the Kubernetes cluster. It contains a collection of core components that implement the Kubernetes API functionality. Typically only these components run on head nodes, there are no user containers that share these nodes.

etcd

The etcd system is at the heart of any Kubernetes cluster. It implements the key value store where all of the objects in a kubernetes cluster are persisted. The etcd servers implement a distributed consensus algorithm, namely RAFT, which ensures that even if one of the storage servers fails, there is sufficient replication to maintain the data stored in etcd, and recover data when an etcd server becomes healthy again and re-adds itself to the cluster. The etcd servers also provide two other important pieces of functionality that Kubernetes makes heavy use of. The first is optimistic concurrency. Every value stored in etcd has a corresponding resource version. When a (key, value) pair is written to an etcd server, it can be conditionalized on a particular resource version. This means that using etcd you can implement compare-and-swap, which is at the core of any concurrency system. Compare and swap enables a user to read a value, and update the value with assurances that no other component in the system has also updated the value. These assurances enable the system to safely have multiple threads manipulating data in etcd without the need for pessimistic locks which can significantly reduce throughput to the server.

In addition to implementing compare and swap, the etcd servers also implement a watch protocol. The value of watch is that it enables clients to efficiently watch for changes in the (key, value) stores for an entire directory of values. As an example, all objects in a namespace are stored within a directory in etcd. The use of a watch, enables a client to efficiently wait for and react to changes without continuous polling of the etcd server.

Apiserver

While etcd is at the core of a Kubernetes cluster, there is actually only a single server that is allowed to have direct access to the Kubernetes cluster, and that is the API server. The API server is the hub of the Kubernetes cluster, it mediates all interactions between clients and the API objects stored in etcd. Consequently it is the central meeting point for all of the various componments. Because of its importance, the API server deserves a deeper introspection and that is covered in a subsequent chapter.

Scheduler

With etcd and the api-server operating correctly, a Kubernetes cluster is in some-ways functionally complete. You can create all of the different API objects like Deployments and Pods, the trouble is, that despite successfully creating a Pod, you will find that it never actually begins to run. Finding a location for a Pod to run is the job of the Kubernetes Scheduler. The

Scheduler scans the api-server for unscheduled Pods and then determines the best node on which to run them. Like the api-server, the scheduler is a complex and rich topic that is covered more deeply in its own chapter.

Controller-manager

Once etcd, the api-server and the scheduler are operational, then you can successfully create Pods and see them scheduled out onto nodes, but you will find that neither Replica Sets, Deployments or Services work as you expect them to. This is because all of the reconciliation control loops needed to implement this functionality are not currently running. Executing these loops is the job of the controller-manager. The controller-manager is the most grab-bag of all of the Kubernetes components, as it has within it numeous different reconciliation control loops to implement many parts of the overall Kubernetes system. Like the api-server and scheduler, the controller manager is covered in detail in its own chapter later in the book.

Components on all nodes

In addition to the components that run exclusively on the head nodes, there are a few components that are present on all nodes in the Kubernetes cluster. These pieces implement essential functionality that is required to be present on all nodes.

Kubelet

The Kubelet is the node-daemon for all machines that are part of a Kubernetes cluster. The Kubelet is the bridge that joins the available CPU, disk and memory for a node into the large Kubernetes cluster. The Kubelet communicates with the API-server to find containers that should be running on its node. Likewise, the Kubelet communicates the state of these containers back up to the api-server so that other reconciliation control loops can observe the current state of these containers.

In addition to scheduling and reporting the state of containers running in Pods on their machines, Kubelets are also responsible for health-checking and restarting the containers that are supposed to be executing on their machines. It would be quite inefficient to push all of the health-state information back up to the API-server for reconciliation loops to take action to fix the health of a container on a particular machine. Instead the Kubelet shot-circuits this interaction and runs the reconciliation loop itself. Thus if a container being run by the Kubelet dies or fails its health-check, the Kubelet restarts it, while also communicating this health state (and the restart) back up to the api-server itself.

Kube Proxy

The other component that runs on all machines is the kube-proxy. The kube-proxy is responsible for implementing the Kubernetes Service load-balancer networking model. The kube-proxy is always watching the endpoints objects for all Services in the Kubernetes cluster. The kube-proxy then programs the network on its node, so that network requests to the virtual IP address of a service, are in-fact routed to the endpoints which implement this service. Every Service in Kubernetes gets a virtual IP address, the kube-proxy is the daemon responsible for defining and implementing the local load-balancer that routes traffic from Pods on the machine to Pods, anywhere in the cluster, that implement the Service.

Scheduled Components

When all of the components described above are successfully operating, they provide a minimally viable Kubernetes cluster, but there are several additional components that are essential to the Kubernetes cluster, but actually rely on the cluster itself for their implementation. This means that while they are essential to cluster function, they also are scheduled, health-checked, operated and updated using calls to the Kubernetes API-server itself.

KubeDNS

The first of these "scheduled" components is the KubeDNS server. When a Kubernetes service is created it gets a virtual IP address, but that IP address is also programmed into a DNS server for easy service discovery. The KubeDNS containers implement this name-service for Kubernetes service objects. The KubeDNS service is itself expressed as a Kubernetes service, so the same routing provided by the kube-proxy routes DNS traffic to the KubeDNS containers. The one important difference is that the KubeDNS service is given a static virtual IP address. This means that the API-server can program the DNS server into all of the containers that it creates, implementing the naming and service discovery for Kubernetes services.

In addition to the KubeDNS service which has been present in Kubernetes since the first versions, there is also a newer alternative CoreDNS implementation that reached general availability (GA) in the 1.11 release of Kubernetes. More information about CoreDNS can be found at *https://coredns.io*.

The ability for the DNS service to be swapped out shows both the modularity and the value of using Kubernetes itself to run components like the DNS server. Replacing KubeDNS with CoreDNS is as easy as stopping one Pod and starting another.

Heapster

The other scheduled component is a binary called "heapster." Heapster is responsible for collecting metrics like CPU, network and disk usage from all containers running inside the Kubernetes cluster. These metrics can be pushed to a monitoring system like influxdb, for alerting and general monitoring of application health in the cluster. Also, importantly, these metrics are used to implement auto-scaling of Pods within the Kubernetes cluster. Kubernetes has an auto-scaler implementation, that, for example, can automatically scale the size of a Deployment whenever the CPU usage of the containers in the Deployment goes above 80%. Heapster is the component that collects and aggregates these metrics to power the reconciliation loop implemented by the auto-scaler. The auto-scaler observices the current state of the world through API calls to heapster.

Note

As of the writing of this book, Heapster is still the source of metrics for auto-scaling in many

Kubernetes clusters. However, as of the 1.11 release it has been deprecated in favor of the new `metrics-server` and Metrics API. Heapster will be removed from Kubernetes in release 1.13.

Add-ons

In addition to these core components, there are numerous systems that you will find on most installations of Kubernetes, these include the Kubernetes dashboard, as well as community add-ons like functions as a service, automatic certificate agents and many more. There are too many Kubernetes add-ons to describe in a few paragraphs, and thus extending your Kubernetes cluster is covered in its own chapter, later on in the book.

Summary

Kubernetes itself is a somewhat complicated distributed system with a number of different components that implement the complete Kubernetes API. The control plane nodes which run the API server. The `etcd` cluster which forms the backing store for the API. The scheduler which interacts with the API server to schedule containers onto specific worker nodes. The controller-manager which operates most of the control loops that keep the cluster functioning correctly. Once the cluster is functioning correctly, there are numerous components that run on top of the cluster itself, including the cluster DNS services, Kubernetes `Service` load-balancer infrastructure, container monitoring and more. We'll learn about even more components you can run on your cluster in Chapters 12 and 13.

Chapter 4. The Kubernetes API Server

As has been mentioned in the previous overview of the Kubernetes components. The API Server is the gateway to the Kubernetes cluster. It is the central touch-point that is accessed by all users, automation and components in the Kubernetes cluster. The API server implements a RESTful API over HTTP, it performs all API operations and is responsible for storing API objects into a persistent storage backend. This chapter covers the details of this operation.

Basic characteristics for manageability

For all of its complexity, from the standpoint of management, the Kubernetes API server is actually relatively simple to manage. Because all of the API server's persistent state is stored in a database that is external to the API server, the API server itself is stateless and can be replicated to handle request load and for fault tolerance. Typically in a highly available cluster, the API server is replicated three times.

The API server can be quite chatty in terms of the logs that it outputs. It outputs at least a single line for every request that it receives. Because of this, it is critical that some form of log rolling is added to the API server or else it can consume all available disk space. However, because the API server logs are essential to understanding the operation of the API server, it is highly recommended that logs be shipped from the API server to a log aggregation service for subsequent introspection and querying for the purposes of debugging user or component requests to the API.

Pieces of the API server

When considering the operation of the Kubernetes API server, there are three large groupings of functionality that are presented by the API server:

- API management, the process by which APIs are exposed and managed by the server
- Request processing, the largest set of functionality which processes individual API requests from a client.
- Internal control loops, Internals that are responsible for background operations necessary to the successful operation of the API server

The following sections cover each of these broad categories.

API Management

Though the primary use for the API is servicing individual client requests, before API requests can be processed, the client must know how to make an API request. Ultimately the API server is an HTTP server and thus every API request is an HTTP request. But the characteristics of those HTTP requests must be described so that the client and server know how to communicate. For the purposes of exploration, it's great to have an API server actually up and running so that you can poke at it. You can either use an existing Kubernetes cluster that you have access to, or you can use the `minikube` (*https://github.com/minikube/minikube*) tool for a local Kubernetes cluster. To make it easy to use the `curl` tool to explore the API server, run the `kubectl` tool in proxy mode to expose an unauthenticated API server on localhost:8001 using the following command:

```
kubectl proxy
```

API Paths

Every request to the API server follows a RESTful API pattern where the request is defined by the HTTP path of the request. All Kubernetes requests begin with the prefix `/api/` (the core APIs) or `/apis/` (APIs grouped by API group). The two different sets of paths are primarily historical. API Groups did not originally exist in the Kubernetes API, so the original or *core* objects like Pods and Services are maintained under the '/api/' prefix without an API group. Subsequent APIs have generally been added under API groups, so they follow the '/apis/<api-group>/' path. For example, the `Job` object is part of the `batch` API group and is thus found under: `/apis/batch/v1/…`

One additional wrinkle for resource paths is whether or not the resource is 'namespaced'. Namespaces in Kubernetes add a layer of grouping to objects, namespaced resources must exist within a namespace, and this namespace is maintained in the path. Of course there are resources that do not live in a namespace (the most obvious example is the Namespace API object itself) and in this case they do not have a namespaces component in their HTTP path.

Here are the components of the two different paths for namespaced resource types:

- `/api/v1/namespaces/<namespace-name>/<resource-type-name>/<resource-name>`
- `/apis/<api-group>/<api-version>/namespaces/<namespace-name>/<resource-type-name>/<resource-name>`

Here are the components of the two different paths for non-namespaced resource types:

- `/api/v1/<resource-type-name>/<resource-name>`
- `/apis/<api-group>/<api-version>/<resource-type-name>/<resource-name>`

API Discovery

Of course, to be able to make requests to the API it is necessary to understand what API objects are available to the client. This process occurs through API discovery on the part of the client. To see this process in action and to explore the API server in a more hands-on manner, we can perform this API discovery ourselves.

First off, to simplify things, we will use the kubectl command line tools built in proxy to provide authentication to our cluster. Run:

```
kubectl proxy
```

This will create a simple server running on port 8001 on your local machine.

We can use this server to start the process of API discovery. We can begin by examining the `/api` prefix:

```
$ curl localhost:8001/api
{
  "kind": "APIVersions",
  "versions": [
    "v1"
  ],
  "serverAddressByClientCIDRs": [
    {
      "clientCIDR": "0.0.0.0/0",
      "serverAddress": "10.0.0.1:6443"
    }
  ]
}
```

You can see that the server returned us an API object of type `APIVersions`, this object provides us with a `versions` field which lists the available versions.

In this case there is just a single one, but for the `/apis` prefix there are many. We can use this version to continue our investigation:

```
$ curl localhost:8001/api/v1
{
  "kind": "APIResourceList",
  "groupVersion": "v1",
  "resources": [
    {
….
    {
      "name": "namespaces",
      "singularName": "",
      "namespaced": false,
      "kind": "Namespace",
      "verbs": [
        "create",
        "delete",
        "get",
        "list",
        "patch",
        "update",
        "watch"
      ],
      "shortNames": [
        "ns"
      ]
```

```
    },
    …
    {
      "name": "pods",
      "singularName": "",
      "namespaced": true,
      "kind": "Pod",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "proxy",
        "update",
        "watch"
      ],
      "shortNames": [
        "po"
      ],
      "categories": [
        "all"
      ]
    },
    {
      "name": "pods/attach",
      "singularName": "",
      "namespaced": true,
      "kind": "Pod",
      "verbs": []
    },
    {
      "name": "pods/binding",
      "singularName": "",
      "namespaced": true,
      "kind": "Binding",
      "verbs": [
        "create"
      ]
    },
   ….
  ]
}
```

(this output is heavily edited for compactness)

Now we are getting somewhere, we can see that the specific resources that are available on a certain path are printed out by the API server. In this case, the returned object contains the list of resources exposed under the `/api/v1/` path.

In addition to the resource types themselves, there is much interesting information in the OpenAPI/Swagger JSON specification that describes the API (the meta-API object), consider the OpenAPI specification for Pod object:

```
{
      "name": "pods",
      "singularName": "",
      "namespaced": true,
      "kind": "Pod",
      "verbs": [
        "create",
        "delete",
        "deletecollection",
        "get",
        "list",
        "patch",
        "proxy",
        "update",
        "watch"
      ],
      "shortNames": [
```

```
      "po"
    ],
    "categories": [
      "all"
    ]
  },
  {
    "name": "pods/attach",
    "singularName": "",
    "namespaced": true,
    "kind": "Pod",
    "verbs": []
  }
```

Looking at this object, the `name` field provides the name of this resource, it also indicates the sub-path for these resources. Because inferring the pluralization of an English word is challenging, the API resource also contains a `singularName` field which indicates the name that should be used for a singular instance of this resource. We previously discussed namespaces, the `namespaced` field in the object description indicates if the object is namespaced or not. The `Kind` field provides the string that is present in the API object's JSON representation to indicate what *kind* of object it is. The `verbs` field is one of the most important in the API object, because it indicates what kind of actions can be taken on that object. The `pods` object contains all of the possible verbs. Most of the effects of the verbs are obvious from their names. The two which require a little more explanation are `watch` and `proxy`. Watch indicates that you can establish a *watch* for the resource. A watch is long running operation which provides notifications about changes to the object. The watch is covered in detail in later sections. Proxy indicates is a specialized action that establishes a proxy netework connection through the API server to network ports. There are only two resources (Pods and Services) which currently support Proxy.

In addition to the actions that you can take on an object that are described as verbs, there are other actions which are modeled as sub-resources on a resource type. As an example of this, you can see the `attach` command which is modelled as a sub-resource:

```
{
  "name": "pods/attach",
  "singularName": "",
  "namespaced": true,
  "kind": "Pod",
  "verbs": []
}
```

Attach provides you with the ability attach a terminal to a running container within a Pod. The 'exec' functionality that let's you execute a command within a Pod is modelled similarly.

OpenAPI Spec Serving

Of course, simply knowing the resources and paths you can use to access the API server is only part of the information that you need in order to access the Kubernetes API. In addition to knowing the HTTP path, you need to know the JSON payload to send and receive. The API server also provides paths to supply you with information about the schemas for Kubernetes resources. These schemas are represented using the OpenAPI (formerly Swagger) syntax. You can pull down the OpenAPI specification at the following path:

- `/swaggerapi` : Before Kubernetes 1.10, serves Swagger 1.2
- `/openapi/v2` : Kubernetes 1.10 and beyond, serves OpenAPI (Swagger 2.0)

The OpenAPI specification is a complete subject unto itself, and thus beyond the scope of this book. In any event, it is unlikely that you will need to access it in your day-to-day operations of Kubernetes. However, the various client programming language libraries are generated using these OpenAPI specifications (the notable exception to this is the Go client library which is currently hand-coded). Thus if you or a user are having trouble accessing parts of the Kubernetes API via a client library the first stop should be the OpenAPI specification to understand how the API objects are modelled.

API Translation

In Kubernetes an API starts out as an Alpha API (e.g. `v1alpha1`), the alpha designation indicates that the API is unstable and unsuitable for production use cases. Users who adopt alpha APIs should expect both that the API surface area may change between Kubernetes releases, and also that the implementation of the API itself may be unstable and even de-stabilize the entire Kubernetes cluster. Alpha APIs are therefore disabled in production Kubernetes clusters.

Once an API has matured it becomes a Beta API (e.g. `v1beta1`), the beta designation indicates that the API is generally stable, but may have bugs or final API surface refinements. In general beta APIs are assumed to be stable between Kubernetes releases and backwards compatability is a goal, however in special cases, beta APIs may still be incompatible between Kubernetes releases. Likewise beta APIs are intended to be stable, but bugs may still exist. Beta APIs are generally enabled in production Kubernetes clusters, but should be used carefully.

Finally an API becomes generally available ("GA", e.g. `v1`). General availability indicates that the API is stable, generally available APIs come with both a guarantee of backward compatability and a deprecation guarantee. Once an API is marked as scheduled for removal, Kubernetes will retain the API for at least three releases or one year, which ever comes first. Deprecation is also fairly unlikely, APIs are deprecated only after a superior alternative has been developed. Likewise generally available APIs are stable and suitable for all production usage.

A particular release of Kubernetes may support multiple different versions (alpha, beta and GA), in order to accomplish this, the API server has three different representations of the API at all times: The "external" representation which is the representation that comes in via an API request. The "internal" representation which is the in-memory representation of the object that is used within the API server for processing, and the "storage" representation which is recorded into the storage layer to persist the API objects. The API server has code within it that knows how to perform the various translations between all of these representations. An API object may be submitted as a `v1alpha1` version, stored as a `v1` object and subsequently retrieved as a `v1beta1` object, or any other arbitrary supported version. These transformations are achieved with reasonable performance using machine generated "deep-copy" libraries which perform the appropriate translations.

Request management

The main purpose of the API server in Kubernetes is to receive and process API calls in the form of HTTP requests. These requests are either from other components in the Kubernetes system, or they are end-user requests. In either event, they are all processed by the Kubernetes API server in the same manner.

Types of requests

There are several broad categories of requests performed to the Kubernetes API server. The simplest requests are `get` requests for specific resources. These requests retrieve the data associated with a particular resource, for example an HTTP `GET` request to the path `/api/v1/namespaces/default/pods/foo` retrieves the data for a Pod named "foo".

A slightly more complicated, but still fairly straight-forward request is a "collection get", or "list", these are requests to list a number of different requests. For example an HTTP `GET` request to the path `/api/v1/namespaces/default/pods` retrives a collection of all pods in the `default` namespace. List requests can also optionally specifiy a Label query, in which case only resources matching that label query are returned.

To create a resource, a `POST` request is used. The body of the request is the new resource that should be created. In the case of a `POST` request, the path is the resource type (e.g. `/api/v1/namespaces/default/pods`. To update an existing resource, a `PUT` request is made to the specific resource path, e.g. `/api/v1/namespaces/default/pods/foo`.

When the time has come to delete a request, an HTTP `DELETE` request to the path of the resource (e.g. `/api/v1/namespaces/default/pods/foo`) will delete the resource. It's important to note that there is no confirmation in such a request, once the HTTP request is made the resource is deleted.

The content type for all of these requests is generally text-based JSON (`application/json`) but recent releases of Kubernetes also support ProtocolBuffer binary encoding. Generally speaking JSON is better for human-readable and debuggable traffic on the network between client and server, but it is significantly more verbose and expensive to parse. Protocol buffers are harder to instrospect using common tools like curl, but enable greater performance and throughput of API requests.

In addition to these standard requests, many requests use the WebSockets protocol to enable streaming sessions between client and server. Examples of such protocols are the `exec` and `attach` commands. These requests are described below.

Life of a request

To better understand what the APIServer is doing for each of these different requests, we'll take apart and describe the processing of a single request to the API server.

Authentication

The first stage of request processing is authentication. Authentication establishes the identity associated with the request. The APIServer supports several different modes of establishing identity including: client certificates, bearer tokens and HTTP basic authentication. In general, client certificates or bearer tokens should be used for authentication, the use of HTTP basic authentication is generally discouraged.

In addition to these local methods of establishing identity, authentication is pluggable and there are several plugin implementations which use remote identity providers. These include support for the OpenID Connect (OIDC) protocol as well as Azure Active Directory. These authentication plugins are compiled into both the API Server as well as the client libraries, this means that you may need to ensure that both the command line tools and API server are roughly the same version or support the same authentication methods.

The API server also supports remote web-hook based authentication configurations where the authentication decision is delegated to an outside server via forwarding a bearer token. The external server validates the bearer token from the end-user and returns the authentication information to the API server.

Given the importance of this to securing a server, it is covered in depth in a later chapter.

RBAC/Authorization

Once the API Server has determined the identity for a request, it moves on to authorization for that request. Every request to Kubernetes follows a traditional role-based access control (RBAC) model. In order to access a request, the identity associated with the request must have the appropriate role associated with the request. Kubernetes RBAC is a rich and complicated topic and as such, we have devoted an entire chapter to the details of how it operates. For the purposes of this summary of the API server, when processing a request the API server determines if the identity associated with the request can access the combination of the verb and the HTTP path in the request. If the identity of the request has the appropriate role it is allowed to proceed, otherwise an HTTP 403 response is returned.

This is covered in much more detail in a later chapter.

Admission Control

Once a request has been authenticated and authorized, it moves on to admission control.

Authentication and RBAC determine if the request is allowed to occur, this is based on the HTTP properties of the request (headers, method and path). Admission control both determines if the request is well formed, as well as potentially applying modifications to the request before it is processed. Admission control defines a pluggable interface:

```
apply(request): (transformedRequest, error)
```

If any admission controller indicates an error, the request is rejected. If the request is accepted, the transformed request is used instead of the initial request. Admission controllers are called serially, each receiving the output of the previous admission controller.

Because admission control is such a general, pluggable mechanism, it is used for a wide variety of different functionality in the API server. For example, it is used to add default values to objects. It can also be used to enforce policy, e.g. requiring that all objects have a certain label. Additionally it can be used to do things like inject an additional container into every Pod. The service mesh Istio uses this approach to inject its sidecar container transparently.

Admission controllers are quite generic, and can be added dynamically to the API server via webhook based admission control.

Validation

Request validation occurs after admission control, although validation can also be implemented as part of Admission Control, especially for external web-hook based validation. Additionally, validation is only performed on a single object, if validation requires broader knowledge of the cluster state, it must be implemented as an Admission Controller.

Request validation ensures that a specific resource included in a request is valid. For example it ensures that the name of a Service object conforms to the rules around DNS names, since eventually the name of a Service will be programmed into the Kubernetes service discovery DNS server. In general, validation is implemented as custom code that is defined per resource type.

Specialized requests

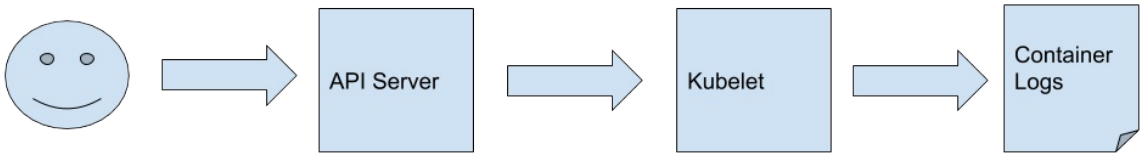
In addition to the standard RESTful requests, the API server has a number of speciallized request patterns that provide expanded functionality to clients.

```
/Proxy, /exec, /attach, /logs
```

The first important class of operations are those that open long-running connections to the API server. These requests provide streaming data rather than an immediate response.

The logs operation is the first streaming request we will describe, because it is the easiest to understand. Indeed by default logs isn't a streaming request at all. A client makes a request to get the logs for a pod by appending `/logs` to the end of the path for a particular Pod (e.g. `/api/v1/namespaces/default/pods/some-pod/logs`) and then specifying the container name as an HTTP query parameter and an HTTP GET request. Given a default request, the API server returns the all of the logs up to the current time as plain text and then closes the HTTP request. However, if

the client requests that the logs are tailed, (by specifying the `follow` query parameter) then the HTTP response is kept open by the API server and new logs are written to the HTTP response as they are received from the kubelet via the API server. This connection is shown below.



Logs is the easiest streaming request to understand because it simply leaves the request open and streams in more data. The rest of the operations take advantage of the WebSocket protocol for bidirectional streaming data, additionally they actually multiplex data within those streams to enable an arbitrary number of bi-directional streams over HTTP. If this all sounds a little complicated, it is, but it is also a valuable part of the API server's surface area.

```
The API Server actually supports two different streaming protocols. It supports the SPDY protocol, as
well as HTTP2/WebSockets, SPDY is being replaced by HTTP2/WebSockets
and thus we focus our attention on the WebSockets protocol.
```

The WebSockets protocol is beyond the scope of this book, but it is documented in a number of other places. For the purposes of understanding the API server, you can simply think of WebSockets as a protocol that transforms HTTP into a bi-directional byte-streaming protocol.

However, on top of those streams, the Kubernetes API server actually introduces an additional multi-plexed streaming protocol. The reason for this is that for many of the use cases, it is quite useful for the API server to be able to service multiple independent byte streams. Consider for example executing a command within a container. In this case there are actually three streams that need to be maintained (stdin, stderr and stdout).

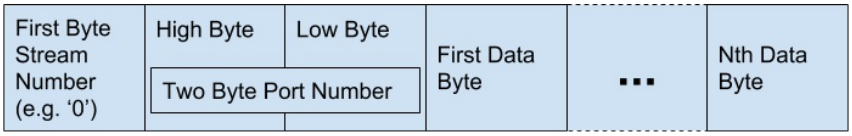
The basic protocol for this streaming is as follows, every stream is assigned a number from zero to 255. This stream number is used for both input and output and conceptually models a single bi-directional byte stream.

For every frame that is sent via the WebSockets protocol, the first byte is the stream number (e.g. ‘0’) and the remainder of the frame is the data that is travelling on that stream.

Using this protocol and web-sockets the API server can simultaneously multi-plex 256 byte streams in a single WebSockets session.

This basic protocol is used for ‘exec’ and ‘attach’ sessions, with the following channels: * 0 - The Stdin stream for writing to the process. Data is not read from this stream. * 1 - The Stdout output stream for reading stdout from the process. Data should not be written to this stream. * 2 - The Stderr output stream for reading stderr from the process. Data should not be written to this stream.

The `/proxy` endpoint is used to port-forward network traffic between the client and containers and services running inside the cluster without those endpoints being externally exposed. For streaming these TCP sessions, the protocol is slightly more complicated. In addition to multi-plexing the various streams, the first two bytes of the stream (after the stream number, so actually the second and third bytes in the Web Sockets frame) are the port number that is being forwarded, so a single WebSockets frame for proxy looks like:



Watch operations

In addition to streaming data, the API server supports a *Watch* API. The idea behind a *Watch* is that it "watches" a path for changes. Thus, instead of polling at some interval for possible updates, which introduces either extra load (due to fast polling) or extra latency (because of slow polling), using a *Watch* enables a user to get low-latency updates with a single connection. When a user establishes a Watch connection to the API server by adding the query parameter `?watch=true` to some API server request. The API server switches into watch mode, and it leaves the connection between client and server open. Likewise, the data returned by the API server is no longer just the API object, it is a `Watch` object which contains both the type of the change (created, updated, deleted), as well as the API object itself. In this way a client can "watch" and observe all changes to that object, or set of objects.

Optimistically Concurrent updates

An additional advanced operation supported by the API server is the ability to perform optimistically concurrent updates of the Kubernetes API. The idea behind optimistic concurrency is the ability to perform most operations without using locks (pessimistic concurrency) and instead detect when a concurrent write has occurred and reject the later of the two concurrent writes. A write that is rejected is not retried, it is up to the client to detect the conflict and retry themselves.

To understand why this optimisitic concurrency and conflict detection is required, it's important to understand the structure of a read/update/write race condition. The operation of many API server clients involves three operations:

Read some data from the API server Update that data in memory Write it back to the API server.

Now imagine what happens when two of these read/update/write patterns happen simultaneously.

Server A reads object O Server B reads object O Server A updates object O in memory on the client Server B updates object O in memory on the client Server A writes object O Server B writes object O

At the end of this, the changes that server A has made are lost because they were overwritten by the update from Server B.

There are two options for solving this race. The first is a pessimistic lock which would prevent other reads from occurring while server A is operating on the object. The trouble with this is that it serializes all of the operations which leads to performance and throughput problems.

The other option, implemented by the Kubernetes API server is optimistic concurrency, which assumes that everything will "just work out" and only detects a problem when a conflicting write is attempted. To achieve this, every instance of an object returns both it's data and a resource version. This resource version indicates the current iteration of the object. When a write occurs, if the resource version of the object is set, then the write is only successful if the current version matches the version of the object. If it does not, an HTTP 409 (Conflict) is returned, and the client is required to retry. To see how this fixes the read/update/write race described above, let's take a look at the operations again:

1. Server A reads object O at version v1
2. Server B reads object O at version v1
3. Server A updates object O at version v1 in memory in the client
4. Server B updates object O at version v1 in memory in the client
5. Server A writes object O at version v1, this is successful
6. Server B writes object O at version v1, but the object is at v2, a 409 conflict is returned.

Alternate Encodings

In addition to supporting JSON encoding of objects for requests, the API server supports two other formats for requests. The encoding of the requests are indicated by the Content-type HTTP header on the request. If this header is missing the content is assumed to be 'application/json' which indicates JSON encoding. The first alternate encoding is YAML, which is indicated by the 'application/yaml' Content-type. YAML is a text-based format that is generally considered to be

more human readable than JSON. In all honesty, there is little reason to use YAML for encoding for communicating with the server, but it can be convenient in a few circumstances (e.g. manully sending files to the server via ‘curl’).

The other alternate encoding for requests and responses is the protocol buffer encoding format. Protocol Buffers are a fairly efficient binary object protocol. Using protocol buffers can result in more efficient and higher throughput requests to the API servers, indeed many of the Kubernetes internal tools use protocol buffers as their transport. The main issue with protocol buffers is that because of their binary nature, they are significantly harder to visualize/debug in their wire format. Additionally not all client libraries currently support protocol buffer requests or responses. The protocol buffer format is indicated by the `application/vnd.kubernetes.protobuf` content-type header.

Common Response Codes

Because the API server is implemented as a RESTful server, all of the responses from the server are aligned with HTTP response codes. Beyond the typical 200 for OK responses and 500s for internal server errors, here are some of the common response codes and their meanings.

202

: Accepted, an asyncronous request to create or delete an object has been received, the result responds with a Status object until the asynchronous request has completed at which point the actual object will be returned.

400

: Bad Request, the server could not parse or understand the request.

401

: Unauthorized, a request was received without a known authentication scheme

403

: Forbidden, the request was received and understood, but access is forbidden.

409

: Conflict, the request was received, but it was a request to update an older version of the object.

422

: Unprocessable entity indicates that the request was parsed correctly but failed some sort of validation.

API Server Internals

In addition to the basics of operating the HTTP restful service, the API server has a few internal services that implement parts of the Kubernetes API. Generally these sort of controll loops are run in a separate binary known as the “controller manager” described in a later chapter, but there are a few control loops that have to be run inside the API server. In each case we describe the functionality as well as the reason for it’s presence in the API server.

CRD Control Loop

Custom Resource Definitions (CRDs) are dynamic API objects that can be added to a running API server. Because the act of creating a CRD inherently creates new HTTP paths which the apiserver must know how to serve, the controller that is responsible for adding these paths is co-located inside the API server. With the addition of delegated API servers (described in a later chapter) this controller has actually been mostly abstracted out of the API server, but it currently still runs in process by default, though it can also be run out of process.

The CRD control loop operates as follows:

```
for crd in AllCustomResourceDefinitions:
    if !RegisteredPath(crd):
       registerPath

for path in AllRegisteredPaths:
    if !CustomResourceExists(path):
       markPathInvalid(path)
       delete custom resource data
       delete path
```

The creation of the custom resource path is fairly straight-forward, but the deletion of a custom resource is a little more complicated. This is because the deletion of a custom resource implies the deletion of all data associated with resources of that type. This is so that if a CRD is deleted, and then at some later data re-added, the old data does not some how get resurected.

Thus, before the HTTP serving path can be removed, the path is first marked as invalid so that new resources can not be created, then all data associated with the CRD is deleted, and finally the path is removed.

Debugging the API Server

Of course, understanding the implementation of the API server is great, but more often than not what you really need to be able to do is to debug what is actually going on with the API server (as well as clients that are calling into the API server). The primary way that this is achieved is via the logs that the API server writes. There are two log streams that the API server exports, the “standard” or “basic” logs, as well as more targetted “audit” logs that try to capture why and how requests were made and the API server state changed. In addition, more verbose logging can be turned on for debugging specific problems.

Basic Logs

By default, the API server logs every request that is sent to the API server. This log includes the client's IP address, the path of the request and the code that the server returned. If an unexpected error results in a server panic, the server also catches this panic, returns a 500 and logs that error to the logs.

```
I0803 19:59:19.929302       1 trace.go:76] Trace[1449222206]: "Create /api/v1/namespaces/default/events"
(started: 2018-08-03 19:59:19.001777279 +0000 UTC m=+25.386403121) (total time: 927.484579ms):
Trace[1449222206]: [927.401927ms] [927.279642ms] Object stored in database
I0803 19:59:20.402215       1 controller.go:537] quota admission added evaluator for: { namespaces}
```

In this log you can see that it starts with the timestamp `I0803 19:59:…` when the log line was emitted, followed by the line number that emitted it `trace.go:76`, and finally the log message itself.

Audit Logs

The audit log is intended to enable a server adminstrator to forensically recover the state of the server and the series of client interactions that resulted in the current state of the data in the Kubernetes API. For example it enables a user to answer questions like: “Why was that replica set scaled up to 100?” “Who deleted that Pod?” and so on.

Audit logs have a pluggable backend for where they are written. Generally audit logs are written to file, but it is also possible for audit logs to be written to a webhook. In either case, the data logged is a structured JSON object of type Event in the `audit.k8s.io` api group.

Auditing itself can be configured via a Policy object in the same API group. This policy allows you to specify the rules by which audit events are emitted into the audit log.

Activating additional logs

Kubernetes uses the `github.com/golang/glog` levelled logging package for it's logging. Using the `--v` flag on the API server you can adjust the level of logging verbosity. In general the Kubernetes project has set log verbosity level 2 '--v=2' as a sane default for logging relevant, but not too spammy messages. If you are looking into specific problems, you can raise the logging level to see more (possibly spammy) messages. Because of the performance impact of excessive logging, its generally recommended not to run with a verbose log level in production. If you are looking for more targetted logging, the `--vmodule` flag enables increasing the log level for individual source files. This can be useful for very targetted verbose logging restricted to a small set of files.

Debugging `kubectl` requests

In addition to debugging the API server via logs, it is also possible to debug interactions with the API server via the `kubectl` command line tool. Like the API server, the `kubectl` command line tool logs via the `github.com/golang/glog` package and supports the `--v` verbosity flag. Setting the verbosity to level 10 (`--v=10`) turns on maximally verbose logging. In this mode, `kubectl` will log all of the requests that it makes to the server, as well as attempting to print `curl` commands that you can use to replicate these requests. Note that these curl commands are sometimes incomplete.

Additionally, if you want to poke at the API server directly, the approach that we used earlier to explore API discovery works well. Running `kubectl proxy` creates a proxy server on localhost that automatically supplies your authentication and authorization credentials based on a local `$HOME/.kube/config` file. Once you run the proxy, it’s fairly straight-forward to poke at various API requests using the curl command.

Summary

As an operator, the core service that you are providing to your users is the Kubernetes API. To effectively provide this service, understanding the core components that make up Kubernetes and how your users will put these APIs together to build applications is critical to implementing a useful and reliable Kubernetes cluster. The topics covered in this chapter should give you basic knowledge of the Kubernetes API and how it is used.

Chapter 5. Scheduler

One of the primary jobs of the Kubernetes API is to schedule containers to worker nodes in the cluster of machines. This task is accomplished by a dedicated binary in the Kubernetes cluster, the Kubernetes scheduler. This chapter describes how the scheduler operates, how it can be extended, and how it can even be replaced or augmented by additional schedulers. Kubernetes can handle a wide variety of workloads from stateless web serving, to stateful applications, big data batch jobs or machine learning on GPUs. The key to ensuring that all of these very different applications can operate in harmony on the same cluster lies in the application of “job scheduling” which ensures that each container is placed onto the worker node best suited to that container.

An overview of scheduling

When a Pod is first created, it generally doesn't have it's `nodeName` field. The `nodeName` indicates the node on which the Pod should execute. The Kubernetes scheduler is constantly scanning the API server (via a Watch request) for Pods which don't have a `nodeName` these are pods that are elegible for scheduling. The Scheduler then selects an appropriate node for the Pod and updates the Pod definition with the `nodeName` that the scheduler selected. Once the `nodeName` is set, the Kubelet running on that node is notified about the Pod's existence (again via a Watch request) and it begins to actually execute that Pod on that node.

Note

If you want to skip the scheduler, you can always set the `nodeName` yourself on a Pod, this "direct schedules" a Pod onto a specific node. This is, in fact, how the DaemonSet controller schedules a single Pod onto each node in the cluster. In general, however, direct scheduling should be avoided, as it tends to make your application more brittle, and your cluster less efficient. In the general use case, you should trust the scheduler to make the right decision, just as you trust the operating system to find a core to execute your program when you launch it on a single machine.

The process of scheduling

When a pod that hasn’t been assigned to a node is discovered by the scheduler it needs to determine which node to schedule the Pod onto. The correct node for a Pod is determined by a number of different factors, some of which are supplied by the user and some of which are calculated by the scheduler. In general the scheduler is trying to optimize a variety of different criteria to find the node that is “best” for the particular Pod.

Predicates

When making the decision about how to schedule a Pod the scheduler uses two generic concepts to make it's decision. The first concept is a Predicate. Simply stated a Predicate indicates whether a Pod "fits" onto a particular node. Predicates are hard constraints, which if violated would lead to a Pod not operating correctly (or at all) on that node. An example of a such a constraint is the amount of memory requested by the Pod. If that memory is unavailable on the node, then the Pod can not get all of the memory that it needs and the contstraint is violated. It is false. Another example of a predicate is a node selector label query specified by the user. In this case, the user has requested that a Pod only run on certain machines as indicated by Node labels. The predicate is false if a node does not have the required label.

Priorities

While predicates indicate situations which are either true or false, the pod either fits or it doesn't, there is an additional generic interface used by the scheduler to determine the preference for one node over another. These preferences are expressed as Priorities or Priority Functions. The role of a priority function is to score the relative value of scheduling a Pod onto a particular node. In contrast to predicates, the priority function does not indicate whether or not the Pod being scheduled onto the node is viable, it is assumed that the Pod can successfully execute on the node, but instead the Predicate function attempts to judge the relative value of scheduling the Pod onto that particular node.

As an example, a priority function would weight nodes where the image has already been pulled, and thus the container would start faster, over nodes where the image is not present and would have to be pulled, delaying Pod startup.

An important priority function is the Spreading priority function. This function is responsible for prioritizing nodes where Pods that are members of the same Kubernetes Service are not present. This Spreading is used to ensure reliability since it reduces the chances that a machine failure will disable all of the containers in a particular service.

Ultimately all of the various predicate values are mixed together to achieve a final priority score for the node and this score is used to determine where the Pod is scheduled.

High level algorithm

For every Pod that needs scheduling, the scheduling algorithm is run, at a high level the algorithm looks like this:

```
schedule(pod): string
    nodes := getAllHealthyNodes()
    viableNodes := []
    for node in nodes:
        for predicate in predicates:
            if predicate(node, pod):
                viableNodes.append(node)

    scoredNodes := PriorityQueue<score, Node[]>
    priorities := GetPriorityFunctions()
    for node in viableNodes:
        score = CalculateCombinedPriority(node, pod, priorities)
        scoredNodes[score].push(node)

    bestScore := scoredNodes.top().score
    selectedNodes := []
    while scoredNodes.top().score == bestScore:
      selectedNodes.append(scoredNodes.pop())

    node := selectAtRandom(selectedNodes)
    return node.Name
```

The actual code itself can be found at *https://github.com/kubernetes/kubernetes/blob/master/pkg/scheduler/core/generic_scheduler.go#*

The basic operation of the scheduler is as follows. First the scheduler gets the list of all currently known and healthy nodes. Then for each predicate, the scheduler evaluates the predicate against the node and the pod being scheduled. If the node is viable (the pod could run on it) the node is added to the list of possible nodes for scheduling. Next all of the priority functions are run against the combination of pod and node. The results are pushed into a priority queue ordered by score, with the best scoring nodes at the top of the queue. Then, all nodes that have the same score are popped off of the priority queue and placed into a final list. All of these nodes are considered to be entirely identical and thus one of them is chosen in a round-robin fashion and is returned as the node where the Pod should be scheduled. Round robin is used instead of random choice to ensure an even distribution of pods amongst identical nodes.

Conflicts

Because there is lag time between when a Pod is scheduled (time `T_1`) and when the container actually executes (time `T_N`), the scheduling decision may become invalid due to other actions during the time interval between scheduling and execution.

In some cases this may mean that a slightly less ideal Node is chosen, when a better one could have been chosen. This could be caused by a Pod terminating after time `T_1` but before time `T_N` or other changes to the cluster. In general these sorts of soft-constraint conflicts aren't that important and normalize in the aggregate. These conflicts are thus ignored by Kubernetes in general. And in general scheduling decisions are only optimal for a single moment in time, they can always become worse as time passes and the cluster changes.

Note

There is some work going on in the Kubernetes community to improve this situation somewhat. There is a "kubernetes-descheduler" project (*https://github.com/kubernetes-incubator/descheduler*) which if run in a Kubernetes cluster scans the cluster for Pods that are determined to be significantly sub-optimal. If such Pods are found the descheduler, evicts the Pod from its current node and consequently the Pod is re-scheduled by the Kubernetes scheduler as if it had just been created.

A more significant kind of conflict occurs when some change to the cluster violates a hard-constraint of the schedule. Imagine for example that the scheduler decides to place a Pod `P` on node `N`. Imagine that `P` requires two cores to operate, and node `N` has exactly two cores of spare capacity. At time `T_1` the scheduler has determined that node `N` has sufficient capacity to run Pod `P`. However after the scheduler makes it's decision in code, and before the decision is written back to the Pod, a new `DaemonSet` is created. This `DaemonSet` creates a different Pod that runs on every node, including node `N` which consumes one core of capacity. Now Node `N` only has a single core free, and yet it has been asked to run Pod `P` which requires two cores. This is not possible, given the new state of node `N`, but the scheduling decision has already been made.

When the Node notices that it has been asked to run a pod that no longer passes the predicates for the Pod and Node, the Pod is marked as *failed*. If the Pod has been created by a ReplicaSet, this failed Pod doesn't count as an active member of the ReplicaSet and thus a new Pod will be created and scheduled onto a different node where it fits. This failure behavior is important to understand because it means that Kubernetes can not be relied upon to reliably run stand-alone Pods. You should always run Pods (even singletons) via a `ReplicaSet` or `Deployment`.

Controlling scheduling with labels, affinity, taints and tolerations

Of course, there are times you want more fine grained control of the scheduling decisions that Kubernetes performs. You could do this by adding your own Predicates and Priorities, but that's a fairly heavy weight task. Fortunately, Kubernetes provides you with a number of tools to customize scheduling without having to implement anything in your own code.

Node Selectors

Remember that every object in Kubernetes has an associated set of *labels*. Labels provide identifying metadata for Kubernetes objects and *label selectors* are often used to dynamically identify sets of API objects for various operations. For example, labels and label selectors are used to identify the set of Pods which serve traffic behind a Kubernetes load balancer.

Label selectors can also be used to identify a subset of the nodes in a Kubernetes cluster that should be used for scheduling a particular Pod. By default, all nodes in the cluster are potential candidates for scheduling, but by filling in the `spec.nodeSelector` field in a `Pod` or `PodTemplate`, the initial set of nodes can be reduced to a subset.

As a concrete example, consider the task of scheduling a workload to a machine that has high-performance storage like NVMe-backed SSD. Such storage (at least in 2018) is very expensive and thus may not be present in every machine. Thus, every machine that has this storage will be given an extra label like:

```
kind: Node
metadata:
  - labels:
      nvme-ssd: true
 ...
```

To create a Pod that will always be scheduled onto a machine with an NVMe SSD, you then set the Pod's `nodeSelector` to match the label on the node:

```
kind: Pod
spec:
  nodeSelector:
    nvme-ssd: true
...
```

Kubernetes has a default Predicate that requires every Node to match the `nodeSelector` label query if it is present. Thus every Pod with the `nvme-ssd` label will always be scheduled onto a Node with the appropriate hardware.

As was mentioned earlier in the section on conflicts, Node selectors are only evaluated at the time of scheduling. If nodes are actively added and removed, then by the time the container executes, it's `nodeSelector` may no longer match the node where it is running.

Node Affinity

Node selectors provide a simple way to guarantee that a Pod lands on a particular node, but it lacks flexibility. In particular they can not represent more complex logical expressions (e.g. “Label foo is either A or B”) and they can not represent *anti-affinity* (“Label foo is A but Label bar is not C”). Finally node selectors are predicates, they specify a requirement, not a preference.

Starting with Kubernetes `1.2`, the notion of *affinity* was added to node selection via the `affinity` structure in the Pod `spec`. Affinity is a more complicated structure to understand, but it is significantly more flexible if you want to express more complicated scheduling policies.

Consider the example above, where a Pod should schedule onto a Node with either label “foo” has a value of either “A” or “B”. This is expressed as the following affinity policy:

```
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          # foo == A or B
          - key: foo
            operator: In
            values:
            - A
            - B
...
```

To show anti-affinity, consider the policy label “foo” has value “A” and label “bar” does not equal “C”. This is expressed in a similar, though slightly more complicated specification:

```
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          # foo == A
          - key: foo
            operator: In
            values:
            - A
          # bar != C
          - key: bar
            operator: NotIn
            values:
            - C
...
```

Note

These two examples include the operators `In` and `NotIn`. Kubernetes also allows `Exists` which only requires that a label key be present regardless of value as well as `NotExists` which requires a label be absent. There are also `Gt` and `Lt` operators which implement greater-than and less-than respectively. If you use the `Gt` or `Lt` operators, then the values array is expected to consist of a

single integer, and likewise your node labels are expected to be integral.

So far, we've seen node affinity provide a more sophisticated way to select nodes, but we have still only expressed a Predicate, this is due to the `requiredDuringSchedulingIgnoredDuringExecution` which is a long-winded but accurate description of the node affinity behavior, the label expression must match when scheduling is performed, but may not be still matching when the Pod is executing.

If you want to express a Priority for nodes instead of a requirement or in addition to a requirement, you can use the `preferredDuringSchedulingIgnoredDuringExecution`. For example, let's suppose given our earlier example, where we required that foo be either "A" or "B", let's also express a preference for scheduling onto nodes labeled "A". The `weight` term in the `preference` struct allows us to tune how significant a preference it is relative to other priorities.

```
kind: Pod
...
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          # foo == A or B
          - key: foo
            operator: In
            values:
            - A
            - B
      preferredDuringSchedulingIgnoredDuringExecution:
        preference:
        - weight: 1
          matchExpressions:
          # foo == A
          - key: foo
            operator: In
            values:
            - A
...
```

Node Affinity is currently a Beta feature. As of Kubernetes 1.4 and beyond, Pod affinity was also introduced with similar syntax (substituting "pod" for "node"). Pod affinity allows you to express a requirement or preference for scheduling alongside, or away from other Pods with particular labels.

Taints and tolerations

Node and Pod affinity allow you to specify preferences for a Pod to schedule (or not) onto a specific set of nodes or near a specific set of Pods. However, that requires user action when creating their containers to achieve the right scheduling behavior. Sometimes as the adminstrator of a cluster you want to affect scheduling without requiring your users to change their behavior.

For example, consider a heterogenous Kubernetes cluster. You may have a mixture of hardware types, some with old 1Ghz processors and some new 3Ghz processors. In general you don't want your users to have their work scheduled onto the older processors unless the user explicitly opts into the older cores. You can achieve this with node anti-affinity, since that would require that every user explicitly add anti-affinity to their Pods for the older machines.

It is this usecase that motivates the development of *node taints*. A Node *taint* is exactly that. When a Taint is applied to a node, the node is considered "tainted" and will be excluded by default from scheduling. Any tainted node will fail a Predicate check at the time of scheduling.

However, consider a user who wants to access the 1Ghz machines. Their work isn't time critical and the 1Ghz machines cost less since they is far less demand. To achieve this, the user opts into the 1Ghz machines by adding a *toleration* for the particular *taint*. This toleration enables the scheduling Predicate to pass, and thus allows for the Node to schedule onto the tainted machine. It is important to note that while a toleration for a taint enables a Pod to run on a tainted machine, it *does not require* that the Pod runs on the tainted machine. Indeed all of the priorities run just as before and thus all of the machines in the cluster are available to execute on. Forcing a Pod onto a particular machine is a use-case for node selectors or affinity as described above.

Summary

One of the core features of Kubernetes is taking a user's request defining a container to execute and scheduling that container onto an appropriate machine. For a cluster adminstrator the operation of the scheduler, and teaching users how to use it well, can be critical to building a cluster that is reliable, and that you can drive to high utlization and efficiency.

Chapter 6. Installing Kubernetes

To fully conceptualize and understand how Kubernetes works, it will be imperative to experiment with an actual Kubernetes cluster. And, fortunately, there is no shortage of tools for one to get going with Kubernetes, and typically this can be achieved within a matter of minutes. Whether it be a local installation on your laptop with a tool like minikube [1] to a managed deployment from any one of the major public cloud providers, a Kubernetes cluster can be had by just about anyone.

While many of these projects and services have greatly helped to commoditize the deployment of a cluster, there are many circumstances that will not allow for this degree of flexibility. Perhaps there are internal compliance or regulatory constraints that prevent the use of public cloud. Or, maybe your organization has already invested heavily in their own data centers. Whatever the circumstances may be, you will be hard pressed to find an environment that would not be suitable for a Kubernetes deployment.

Beyond the logistics of where and how you consume Kubernetes, in order to fully appreciate how the distributed components of Kubernetes operate, it is also important to understand the architectures that make production-ready, containerized application delivery a reality. In this chapter we will explore the services involved and how they are installed.

kubeadm

Among the wide array of Kubernetes installation solutions is the community-supported `kubeadm` utility. This application provides all of the functionality one will need to install Kubernetes. In fact, in the simplest of cases, a user can have a Kubernetes installation operational in a matter of minutes, with just a single command. This simplicity makes it a very compelling tool for developers and those with production-grade needs alike. Because the code for `kubeadm` lives in-tree and is released in conjunction with a Kubernetes release, it borrows common primitives and is thoroughly tested for a large number of use cases.

Note

Because of the simplicity and great utility provided by `kubeadm`, many of the other installation tools actually leverage `kubeadm` under the covers. And, the number of projects following this trend increases regularly. So, regardless of whether you ultimately choose `kubeadm` as your preferred installation tool, understanding how it works will likely help you further understand the tool you have chosen.

A production-grade deployment of Kubernetes will ensure that data may be secured both during transport and at rest, that the Kubernetes components are well matched with their dependencies, that integrations with the environment are well-defined, and that the configuration of all of the cluster components work well together. Ideally, too, these clusters will be easily upgraded, and that the resulting configuration is continually reflective of these best practices. `kubeadm` can help us achieve all of the above.

Requirements

`kubeadm`, just like all of the Kubernetes binaries, is statically linked. As such, there are no dependencies on any shared libraries, and `kubeadm` may be installed on just about any `x86_64`, `ARM`, or `PowerPC` Linux distribution.

Fortunately, there is also not much that we need from a host application perspective either. Most fundamentally, we will require a container runtime and the Kubernetes `kubelet`, but there are also a few necessary standard Linux utilities as well.

When it comes to installing a container runtime, you should ensure that the runtime adheres to the Container Runtime Interface (CRI). This open standard defines the interface that the `kubelet` will use to speak to the runtime available on the host. At the time of this writing, some of the most popular CRI-compliant runtimes are Docker, rkt, and cri-o. For each of these, the operator should consult the installation instructions provided by each of the respective projects.

Note

When choosing a container runtime be sure to reference the Kubernetes release notes. Each release will clearly indicate which container runtimes have been tested against for that release. This can provide a clear indication to an administrator as to which runtimes and versions are known to be both compatible and performant.

kubelet

As you may recall from Chapter 7, the `kubelet` is the on-host process responsible for interfacing with the container runtime. In the most common cases this work will typically amount to reporting Node status to the API server and managing the full life-cycle of Pods that have been scheduled to the host on which it resides.

Installation of the `kubelet` is typically as simple as downloading and installing the appropriate package for the target distribution. In all cases you should be sure to install the `kubelet` with a version that matches the Kubernetes version you intend to run. The `kubelet` will be the single Kubernetes process that will be managed by the host service manager. In almost all cases this will, most probably, be `systemd`.

If you are installing the `kubelet` with the system packages built and provided by the community (currently `deb` and `rpm`), these `kubelet`s will be managed by systemd. As with any process managed in this way, a unit file will define which user the service runs as, what the command line options are, how the service dependency chain is defined, and what the restart policy should be:

```
[Unit]
Description=kubelet: The Kubernetes Node Agent
Documentation=http://kubernetes.io/docs/

[Service]
ExecStart=/usr/bin/kubelet
Restart=always
StartLimitInterval=0
RestartSec=10

[Install]
WantedBy=multi-user.target
```

Note

Even if you are not installing the `kubelet` with the community-provided packages, examining the provided unit files will often be helpful for understanding the common best-practices for running the `kubelet` daemon. These unit files change often, so be sure to reference the versions that match your deployment target.

The behavior of the `kubelet` can be manipulated by adding additional unit files to the `/etc/systemd/system/kubelet.service.d/` path. These unit files will be read lexically (so name them appropriately), and will allow you to override the how the package configures the `kubelet`. This may be required if your environment calls for specific needs (ie. container registry proxies).

For example, when deploying Kubernetes to a supported cloud provider, you will want to set the `--cloud-provider` parameter on the `kubelet`:

```
$ cat /etc/systemd/system/kubelet.service.d/09-extra-args.conf
[Service]
Environment="KUBELET_EXTRA_ARGS= --cloud-provider=aws"
```

With this additional file in place, we simply perform a daemon reload, and then restart the

service:

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

By and large, the default configurations provided by the community are typically more than adequate, and usually do not require modification. With this technique, however, we can utilize the community defaults while simultaneously maintaining our ability to override when applicable.

The `kubelet` and container runtime will be necessary on all hosts in the cluster.

Installing the Control Plane

Within Kubernetes the componentry that directs the actions of the worker nodes is termed the control plane. As we covered in Chapter 3, these components consist of the api server, the controller manager, and the scheduler. Each of these daemons will direct some portion of how the cluster will ultimately operate.

In addition to the Kubernetes components, we will require some place to store our cluster state. That data store is etcd.

Fortunately, `kubeadm` is capable of installing all of these daemons on a host (or hosts) that we, as administrators, have delegated as a control plane node. `kubeadm` does so by creating a static manifest for each of the daemons that we require.

Note

With static manifests we can write Pod specifications directly to disk, and the `kubelet` upon start, will immediately attempt to launch the containers specified therein. In fact, the `kubelet` will also monitor these files for changes, and attempt to reconcile any specified changes. Note well, however, that since these Pods are not managed by the control plane, they cannot be manipulated with the `kubectl` command line interface.

In addition to the daemons, we will want to secure the components with TLS, create a user that can interact with the API, and provide the capability for workers nodes to join the cluster. `kubeadm` does all of this.

In the simplest of scenarios, we can install the control plane components on a node that has already been prepared with a running `kubelet` and functional container runtime like so:

```
$ kubeadm init
```

Simple, right?

After detailed descriptions of the steps `kubeadm` has taken on behalf of the user, the end of the output might look something like so:

```
...
Your Kubernetes master has initialized successfully!

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

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

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

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

  kubeadm join --token 878b76.ddab3219269370b2 10.1.2.15:6443 \
```

```
--discovery-token-ca-cert-hash \
sha256:312ce807a9e98d544f5a53b36ae3bb95cdcbe50cf8d1294d22ab5521ddb54d68
```

kubeadm Configuration

While `kubeadm init` is the simplest case for configuring a controller node, `kubeadm` is capable of managing all sorts of configurations. This can be achieved by way of the various, but somewhat limited, number `kubeadm` command line flags, as well as the (more capable) `kubeadm` API.

This API looks something like so:

```
apiVersion: kubeadm.k8s.io/v1alpha1
kind: MasterConfiguration
api:
  advertiseAddress: <address|string>
  bindPort: <int>
etcd:
  endpoints:
  - <endpoint1|string>
  - <endpoint2|string>
  caFile: <path|string>
  certFile: <path|string>
  keyFile: <path|string>
networking:
  dnsDomain: <string>
  serviceSubnet: <cidr>
  podSubnet: <cidr>
kubernetesVersion: <string>
cloudProvider: <string>
authorizationModes:
- <authorizationMode1|string>
- <authorizationMode2|string>
token: <string>
tokenTTL: <time duration>
selfHosted: <bool>
apiServerExtraArgs:
  <argument>: <value|string>
  <argument>: <value|string>
controllerManagerExtraArgs:
  <argument>: <value|string>
  <argument>: <value|string>
schedulerExtraArgs:
  <argument>: <value|string>
  <argument>: <value|string>
apiServerCertSANs:
- <name1|string>
- <name2|string>
certificatesDir: <string>
```

and may be provided to the `kubeadm` command line with the `--config` flag. Regardless of whether you, as an administrator, decide to explicitly use this configuration format or not, one will always be generated interally upon executing `kubeadm init`. Moreover, this config will be saved as a `ConfigMap` to the just-provisioned cluster. This functionality serves two purposes. First, to provide a reference for those needing to understand how a cluster had been configured. And, secondarily, it may be leveraged when upgrading a cluster. in that case, a user will modify the values of this `ConfigMap` and then execute a `kubeadm upgrade`.

Note

The `kubeadm` config is also accessible by standard `kubectl ConfigMap` interrogation and is, by convention, named the `cluster-info ConfigMap` in the `kube-public` namespace.

Preflight Checks

Once we have run this command, `kubeadm` will first execute a number of “pre-flight” checks. These sanity checks will ensure that our system is appropriate for an install. “Is the `kubelet` running?”, “Has swap been disabled?”, and “Are baseline system utilities installed?” are the types of questions that will be asked here. And, naturally, `kubeadm` will exit with an error if these baseline conditions are not met.

Note

While not recommended, it is possible to sidestep the preflight checks with the `--skip-preflight-checks` command line option. This should only be exercised by advanced administrators.

Certificates

After all preflight checks have been satisfied, `kubeadm`, by default, will generate it's own certificate authority (CA) and key. This CA will then be used to, subsequently, sign various certificates against it. Some of these certificates will be used by the API server when securing inbound requests, authenticating users, when making outbound requests (ie. to an aggregate API server), and for mutual TLS between the API server and all downstream kubelets. Others will be used to secure ServiceAccounts.

All of these PKI assets will be placed in the `/etc/kubernetes/pki` directory on the control plane node:

```
$ ls -al /etc/kubernetes/pki/
total 56
drwxr-xr-x 2 root root 4096 Mar 15 02:42 .
drwxr-xr-x 4 root root 4096 Mar 15 02:42 ..
-rw-r--r-- 1 root root 1229 Mar 15 02:42 apiserver.crt
-rw------- 1 root root 1675 Mar 15 02:42 apiserver.key
-rw-r--r-- 1 root root 1099 Mar 15 02:42 apiserver-kubelet-client.crt
-rw------- 1 root root 1679 Mar 15 02:42 apiserver-kubelet-client.key
-rw-r--r-- 1 root root 1025 Mar 15 02:42 ca.crt
-rw------- 1 root root 1675 Mar 15 02:42 ca.key
-rw-r--r-- 1 root root 1025 Mar 15 02:42 front-proxy-ca.crt
-rw------- 1 root root 1675 Mar 15 02:42 front-proxy-ca.key
-rw-r--r-- 1 root root 1050 Mar 15 02:42 front-proxy-client.crt
-rw------- 1 root root 1675 Mar 15 02:42 front-proxy-client.key
-rw------- 1 root root 1675 Mar 15 02:42 sa.key
-rw------- 1 root root  451 Mar 15 02:42 sa.pub
```

Note

As this default CA is self-signed, any third-party consumers will need to also provide the CA certificate chain when attempting to use a client certificate. Fortunately, this is not typically problematic for Kubernetes users as the `kubeconfig` is capable of embedding this data, and moreover, because `kubeadm` will do so by default with the `kubeconfig` files it generates.

Self-signed certificates, while extremely convenient, are sometimes not the preferred approach. This is often especially true in corporate environments or for those with more exacting compliance requirements. In this case, a user may pre-populate these assets in the `/etc/kubernetes/pki` directory prior to executing `kubeadm init`. In this case, `kubeadm` will attempt to use the keys and certificates that may already be in place, and to generate those that may not already be present.

etcd

In addition to the Kubernetes components that are configured by way of `kubeadm`, by default, if not otherwise specified, `kubeadm` will attempt to start a local `etcd` server instance. This daemon will be started in the same manner as the Kubernetes components (static manifests), and will persist its data to the control plane node's filesystem via local host volume mounts.

Note

At the time of this writing, `kubeadm init`, by itself, does not natively secure the `kubeadm`-managed etcd server with TLS. This basic command is only intended to configure a single control plane node, and typically for development purposes only.

Users that will be using `kubeadm` for production installs should provide a list of TLS-secured etcd endpoints with the `--config` option described earlier in this chapter.

While having an easily-deployable etcd instance is favorable for a simple Kubernetes installation process, this would not be appropriate for a production-grade deployment. Under these circumstances an administrator would normally want to deploy a multi-node and highly-available etcd cluster that sits adjacent to the Kubernetes deployment. As the etcd data store will contain all state for the cluster, it is important to treat it with care. While Kubernetes components are easily replaceable, etcd is not. And, as a result, etcd has a component lifecycle (install, upgrade, maintenance, etc.) that is quite different. For this reason, a production Kubernetes cluster should segregate these responsibilities.

Secrets

All data that is written to etcd is unencrypted by default. If someone were to gain privileged access to the disk backing etcd, the data would be readily available. Fortunately, much of the data that Kubernetes will persist to disk is not sensitive in nature.

The exception, however, is Secret data, and as the name suggests, Secret data should remain, well, secret. To ensure that this data is encrypted on its way to etcd, administrators should leverage the `--experimental-encryption-provider-config kube-apiserver` parameter. With this parameter, administrators may define symmetric keys that will be used to encrypt all Secret data. This is accomplished with an EncryptionConfig:

```
$ cat encryption.conf
kind: EncryptionConfig
apiVersion: v1
resources:
  - resources:
    - secrets
    providers:
    - identity: {}
    - aescbc:
        keys:
        - name: encryptionkey
          secret: BHk4lSZnaMjPYtEHR/jRmLp+ymazbHirgxBHoJZqU/Y=
```

In the case of the recommended `aescbc` encryption type, the secret field should be a randomly-generated 32-byte key. Now by adding `--experimental-encryption-provider-config=/path/to/encryption.conf` to the `kube-apiserver` command line parameters, all Secrets will be encrypted before being written to etcd. This may help alleviate the leakage of sensitive data.

You may have noticed that the EncryptionConfig also includes a `resources` field. For our use case Secrets are the only resources we would like to encrypt, but any resource type may be included here. Use this according to your organization's needs, but remember that encrypting this data does marginally impact performance of the API server's writes. As a general rule of thumb, only encrypt the data that you deem sensitive.

This configuration supports multiple encryption types, some of which may be more or less appropriate for your specific needs. Likewise, this configuration also supports key rotation, a measure that is necessary to ensure a strong security stance. Be sure to consult the Kubernetes documentation for additional details on this experimental feature.

Note

At the time of this writing, this is still an experimental flag. As this is subject to change, native support for this feature in `kubeadm` is limited. You may still make use of this feature by adding `encryption.conf` to the `/etc/kubernetes/pki` directory of all control plane nodes, and by adding this configuration parameter to the `apiServerExtraArgs` field within your `kubeadm` MasterConfig prior to `kubeadm init`.

Note

Your requirements for data at rest will have dependencies on your architecture. If you have chosen to colocate your etcd instances on your control plane nodes, utilizing this feature might not serve your needs, as encryption keys would also be colocated with the data. In the event that privileged access to the disk is gained, they keys may be used to unencrypt the etcd data, thus subverting our efforts to secure these resources. This is yet another compelling reason for segregating etcd from your Kubernetes control plane nodes.

kubeconfig

In addition to creating the PKI assets and configuring the static manifests that will serve the Kubernetes components, `kubeadm` will also generate a number of `kubeconfig` files. Each of these files will be used for some means of authentication. Most of these will be used to authenticate each of the Kubernetes services against the API, but `kubeadm` will also create a primary administrator `kubeconfig` file at `/etc/kubernetes/admin.conf`.

Note

Because `kubeadm` creates this `kubeconfig` with cluster administrator credentials so easily, many users tend to use these generated credentials for much more than their intended use. These credentials should be used *only* for bootstrapping a cluster. Any production deployment should always configure additional identity mechanisms, and these will be discussed in the next chapter.

Taints

For production use cases it is recommended that user workloads be isolated from the control plane components. As such, `kubeadm` will taint all control plane nodes with the `node-role.kubernetes.io/master` taint. This instructs the scheduler to ignore all nodes with this taint when determining where a Pod may be placed.

If your use case is that of a single-node master, you may remove this restriction by removing the taint from the node:

```
kubectl taint nodes <node name> node-role.kubernetes.io/master-
```

Installing Worker Nodes

Worker nodes will follow a very similar installation mechanism. Again, we require the container runtime and the `kubelet` on every node. But, for workers, the only other Kubernetes component that we will need will be the `kube-proxy` daemon. And, just as with the control plane nodes, `kubeadm` will start this process by way of another static manifest.

Most significantly, this process will perform a TLS bootstrapping sequence. Through a shared token exchange process, `kubeadm` will temporarily authenticate the node against the API server, and then attempt to perform a certificate signing request (CSR) against the control plane CA. Once the node's credentials have been signed, these will serve as the authentication mechanism at runtime.

This sounds complex, but, again, `kubeadm`, makes this process extraordinarily simple:

```
$ kubeadm join --token <token> --discovery-token-ca-cert-hash <hash> <api endpoint>
```

OK, so maybe not as simple as the control plane case, but pretty simple nonetheless. And, in the case where you may be using `kubeadm` manually, the output from the `kubeadm init` command even provides the precise command that needs to be run on a worker node.

Obviously, if we are asking a worker node to join itself to the Kubernetes cluster, we will need to tell it where it should register itself. That is where the `<api endpoint>` parameter comes in. This will be comprised of the IP (or domain name) and port of the API server.

As this mechanism allows for a node to initiate the join, we will want to ensure that this action is secure. For obvious reasons, we do not want just any node to be able to join the cluster, and similarly, we will want the worker node to be able to verify the authenticity of the control plane. This is where the `--token` and `--discovery-token-ca-cert-hash` parameters come into play.

The `--token` parameter is a bootstrap token that has been predefined with the control plane. In our simple use case above, a bootstrap token has been automatically allocated by way of the `kubeadm init` invocation. Users may also create these bootstrap tokens on the fly:

```
$ kubeadm token create [--ttl  <duration>]
```

This mechanism is especially handy when adding new worker nodes to the cluster. In this case, the steps would simply be to use `kubeadm token create` to define a new bootstrap token, and then use that token in a `kubeadm join` command on the new worker node.

The `--discovery-token-ca-cert-hash` provides the worker node a mechanism to validate the CA of the control plane. By pre-sharing the SHA256 hash of the CA, the worker node may validate that the credentials that it received, were, in fact, from the intended control plane.

The whole command may look something like this:

```
$ kubeadm join --token 878b76.ddab3219269370b2 10.1.2.15:6443 \
    --discovery-token-ca-cert-hash \
```

```
sha256:312ce807a9e98d544f5a53b36ae3bb95cdcbe50cf8d1294d22ab5521ddb54d68
```

Add-ons

Once you have installed the control plane and have brought up a few worker nodes, the obvious next step will be to get some workloads deployed. Before we can do so, we will need to deploy a few add-ons.

Minimally, we will need to install a Container Networking Interface (CNI) plugin. This plugin will provide Pod-to-Pod (also known as “east-west”) network connectivity. There are a multitude of options out there, each with their own specific lifecycles, so `kubeadm` stays out of the business of trying to manage these. In the simplest of cases, this will simply be a matter of applying the DaemonSet manifest outlined by your CNI provider.

Additional add-ons that you might want in your production clusters would probably include log aggregation, monitoring, and maybe even service mesh capabilities. Again, as these can be complex, `kubeadm` does not attempt to manage them.

The one *special* add-on that `kubeadm` will manage is that of cluster DNS. `kubeadm` currently supports `kube-dns` and `CoreDNS`, with `kube-dns` being the default. As with all parts of `kubeadm` you may even choose to forego these standard options, and install the cluster DNS provider of your choosing.

Phases

As we alluded to earlier in the chapter, `kubeadm` serves as the basis for a variety of other Kubernetes installation tools. As you might imagine, if we are trying to make use of `kubeadm` in this way, we may want some parts of the installation to be managed by `kubeadm` and others to be handled by the wrapping installer framework. `kubeadm` supports this use case as well with a feature called `phases`.

With `phases`, a user may leverage `kubeadm` to perform discreet actions undertaken in the installation process. For instance, maybe the wrapping tool would like to use `kubeadm` for its ability to generate PKI assets. Or, perhaps that tool wants to leverage `kubeadm`'s pre-flight checks in order to ensure that a cluster has best practices in place. All of this, and more, is available with `kubeadm phases`.

High Availability

If you have been paying close attention, you probably noticed that we haven't spoken about a highly available control plane. That is somewhat intentional.

As the purview of `kubeadm` is primarily from the perspective of a single node at a time, evolving `kubeadm` into a general-use tool for managing highly available installs, would be relatively complicated. Doing so would start to blur the lines of the Unix philosophy of "doing one thing, and doing it well".

That said, `kubeadm` can (and is) used to provide the components necessary for a highly available control plane. While there are a number of precise (and sometimes nuanced) actions that a user will need to take in order to create a highly available control plane, the basic steps are: 1. create a highly-available etcd cluster 2. initialize a primary control plane node with `kubeadm init` and a configuration that makes use of the etcd cluster created in step 1. 3. transfer the PKI assets securely to all of the other control plane nodes 4. front the control plane API servers with a load balancer 5. join all workers nodes to the cluster by way of the load balanced endpoints.

Note

If this is your use case, and you would like to use `kubeadm` to install a production-grade, highly-available cluster, be sure to consult `kubeadm` high-availability documentation. This documentation is maintained with each release of Kubernetes.

Upgrades

As with any deployment, there will come a time when you will want to take advantage of all new features that Kubernetes has to offer. Similarly, if you require a critical security update, you will want the ability to enable it with as little disruption as possible. Fortunately, Kubernetes provides for zero-downtime upgrades. Your applications will continue to run, while the underlying infrastructure is modified.

While there are countless ways to upgrade a cluster, we will focus on the `kubeadm` use case; a feature that has been available since version 1.8.

There are a lot of moving parts in any Kubernetes cluster, and this can make orchestrating the upgrade complicated. `kubeadm` simplifies this significantly as it is able to track well-tested version combinations for the kubelet, etcd, and the container images that serve the Kubernetes conntrol plane.

The order of operations when performing an upgrade are straightforward. First we will `plan` our upgrade, and then `apply` our determined plan.

During the `plan` phase, `kubeadm` will analyze the running cluster, and determine the possible upgrade paths. In the simplest case, we will upgrade to a minor or patch release (ie. from 1.10.3 to 1.10.4). Slightly more complicated will be the upgrade to a whole new minor release that is two (or more) releases forward (ie. 1.8 to 1.10). In this case, we will need to walk the upgrades through each successive minor version until we reach our desired state.

Under the covers, `kubeadm` will perform a number of pre-flight checks to ensure that the cluster is healthy, and then examine the `kubeadm-config` ConfigMap in the `kube-system` Namespace. This ConfigMap will help `kubeadm` determine the available upgrade paths, and will ensure that any custom configuration items are carried forward.

While much of the heavy lifting will happen automatically, you may recall from above that the kubelet (and kubeadm itself) is not managed by kubeadm. When performing the `plan kubeadm` will indicate which unmanaged components also need to be upgraded:

```
root@control1:~# kubeadm upgrade plan
[preflight] Running pre-flight checks.
[upgrade] Making sure the cluster is healthy:
[upgrade/config] Making sure the configuration is correct:
[upgrade/config] Reading configuration from the cluster...
[upgrade/config] FYI: You can look at this config file with
'kubectl -n kube-system get cm kubeadm-config -oyaml'
[upgrade/plan] computing upgrade possibilities
[upgrade] Fetching available versions to upgrade to
[upgrade/versions] Cluster version: v1.9.5
[upgrade/versions] kubeadm version: v1.10.4
[upgrade/versions] Latest stable version: v1.10.4
[upgrade/versions] Latest version in the v1.9 series: v1.9.8

Components that must be upgraded manually after you have upgraded
the control plane with 'kubeadm upgrade apply':
COMPONENT   CURRENT      AVAILABLE
Kubelet     4 x v1.9.3   v1.9.8

Upgrade to the latest version in the v1.9 series:
```

```
COMPONENT            CURRENT   AVAILABLE
API Server           v1.9.5    v1.9.8
Controller Manager   v1.9.5    v1.9.8
Scheduler            v1.9.5    v1.9.8
Kube Proxy           v1.9.5    v1.9.8
Kube DNS             1.14.8    1.14.8

You can now apply the upgrade by executing the following command:

        kubeadm upgrade apply v1.9.8

_____________________________________________________________________

Components that must be upgraded manually after you have upgraded
the control plane with 'kubeadm upgrade apply':
COMPONENT   CURRENT      AVAILABLE
Kubelet     4 x v1.9.3   v1.10.4

Upgrade to the latest stable version:

COMPONENT            CURRENT   AVAILABLE
API Server           v1.9.5    v1.10.4
Controller Manager   v1.9.5    v1.10.4
Scheduler            v1.9.5    v1.10.4
Kube Proxy           v1.9.5    v1.10.4
Kube DNS             1.14.8    1.14.8

You can now apply the upgrade by executing the following command:

        kubeadm upgrade apply v1.10.4

_____________________________________________________________________
```

You should upgrade system components consistent with the manner in which they were installed (typically with the OS package manager).

Once you have determined your planned upgrade approach, begin to execute the upgrades in the order specified by `kubeadm`. If there are multiple releases in the upgrade path, perform those on each node as indicated.

```
root@control1:~# kubeadm upgrade apply v1.10.4
```

Again, preflight checks will be performed, primarily to ensure that the cluster is still healthy, backups of the various static Pod manifests will be made, and the upgrade will take place.

In terms of node order, ensure that you are upgrading the control plane first, and then perform the upgrades on each worker node. Control plane nodes should be unregistered as upstreams for any front-facing load balancers, upgraded, and then once the entire control plane has been successfully upgraded, all control plane nodes should be re-registered as upstreams with the load balancer. This may introduce a short-lived period of time where the API is unavailable, but it will ensure that all clients have a consistent experience.

If you are performing upgrades for each worker in-place, the workers may be upgraded in parallel. Note that this may result in a period of time where there are no kubelets available for scheduling Pods. Alternatively, you may upgrade workers in a rolling fashion. This will ensure that there is always a Node that may be deployed to.

Note

If you upgrade also involves simultaneously performing disruptive upgrades on the worker nodes

(ie. upgrading the kernel), it is advisable to use the `kubectl cordon` and/or `kubectl drain` semantics to ensure that your user workloads are rescheduled prior to the maintenance

Summary

In this chapter we took a look at how to easily install Kubernetes, across a number of use cases. While we have only scratched the surface with regard to what `kubeadm` is capable of, we hope we have demonstrated what a versatile tool it can be.

As many of the deployment tools available today have kubeadm as an underpinning, understanding how it works should help you understand what higher-order tools are doing "under the covers." And, if you are so inclined, this understanding will help you develop your own in-house deployment tooling.

[1] *https://github.com/kubernetes/minikube*

Chapter 7. Authentication and User Management

Now that we have successfully installed Kubernetes, one of the most fundamental aspects of a successful deployment will center around consistent user management. As with any multi-tenant, distributed system, user management will form the basis for how Kubernetes will ultimately authenticate identities, determine appropriate levels of access, enable self-service capabilities, and maintain auditability.

In this chapter and the next we will explore how we can make best use of the authentication and access control capabilities of Kubernetes. But to get a true understanding of how these constructs work, it is important for us to first understand the lifecycle of an API request.

Every API request that makes its way to the API server will need to successfully navigate a series of challenges before the server will accept (and subsequently act) on the request. Each of these tests will fall into one of three groups: authentication, access control, and admission control.

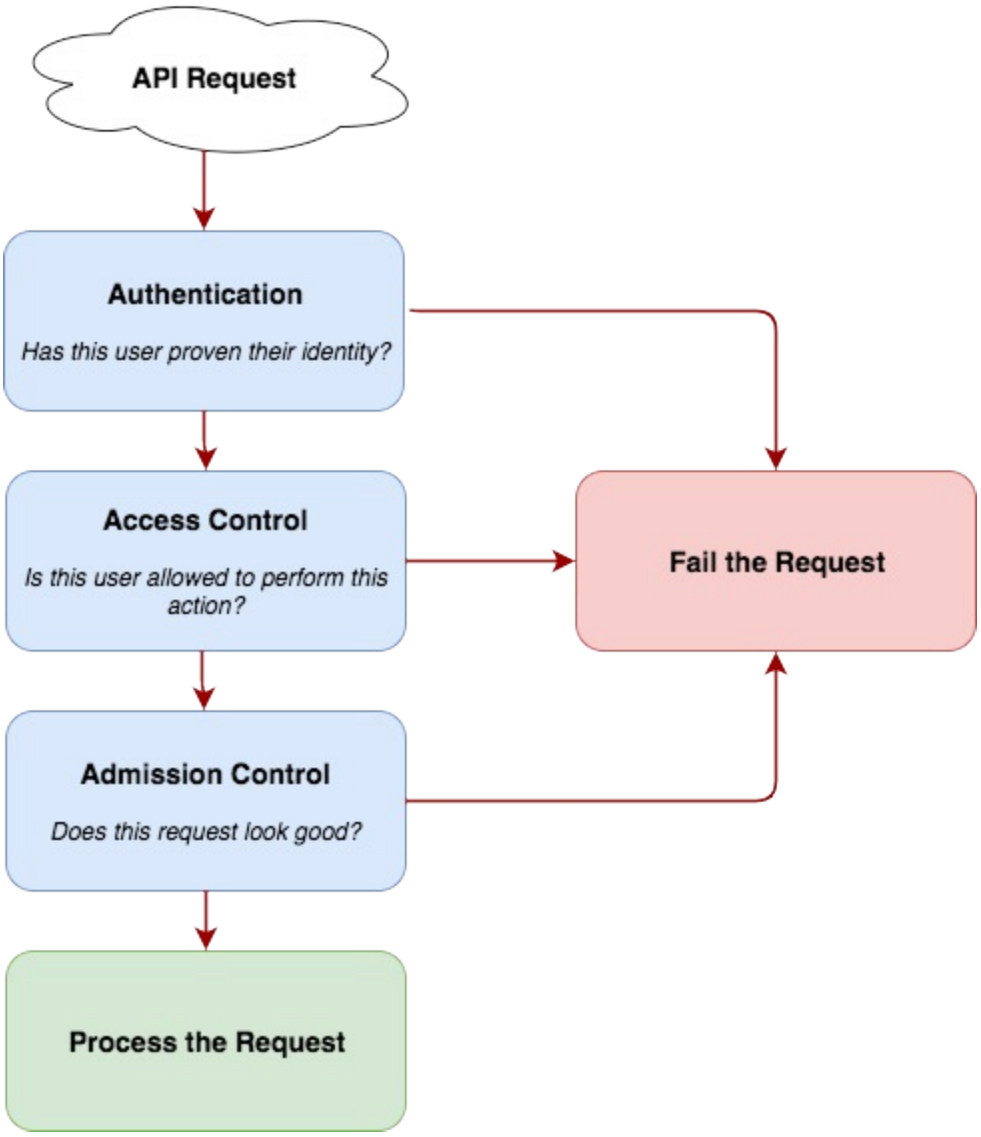


Figure 7-1. Kubernetes API Request Flow

How many of these challenges are in place, and their individual complexity will depend on how the Kubernetes API server is configured, but best practices call for production clusters to implement all three in some form or fashion.

The first two phases of servicing an API request (authentication and access control) focus on what we know about a user. In this chapter we will develop an understanding of what a user is from the perspective of the API server and ultimately how to how to leverage user resources to provide secure API access to the cluster.

users

The term “users” will pertain to how you, and I (and maybe even how your continuous delivery tooling) will connect and gain access to the Kubernetes API. Users, in the most common case, are often connecting to the Kubernetes API from some external place, and often by way of the `kubectl` command line interface. But, as the Kubernetes API forms the basis for all interactions with the cluster, these controls are also in place for all kinds of access; your custom scripts and controllers, the web user interface, everything. This provides a consistent, secure position from which to start.

You may have noticed that, until now, we have refrained from using a capital *U* when referring to “users.” What many newcomers to Kubernetes are often surprised to learn is that amongst the wide array of resources provided by the API, users are, in fact, not a top-level supported resource. They are not manipulated directly by way the Kubernetes API, but, most commonly, they are defined in an external user identity management system.

There is good reason for this as it stands in support of good user management hygiene. If you are like the vast majority of organizations that have deployed Kubernetes, you will almost certainly already have some form of user management in place. Whether this comes in the form of a corporate-wide Active Directory cluster or a one-off LDAP server, how you manage your users should remain consistent across your organization, regardless of the systems consuming it. Kubernetes supports this design tenet by providing the connectivity to leverage these existing systems, thus enabling consistent and effective user management across your infrastructure.

Note

The absence of such systems does not mean that you won’t be able to use Kubernetes, it just may mean that you may need to leverage a different mechanism for authenticating users, as we will discover in the following section.

Authentication

At the time of this writing Kubernetes supports multiple ways of authenticating against the API. As with any authentication mechanism, this will serve as the first gatekeeper for any kind of programatic access. The questions we are evaluating here are “Who is this user?” and, “do their credentials match our expectations?” At this point in the API flow we are not *yet* concerned about whether the request *should* be granted based on the user’s role, or even if the request conforms to our standards. The question here is simple, and the answer binary: “Is this a genuine user?”

Just as with many well-designed REST-based APIs, there are multiple strategies that Kubernetes may employ for authenticating users. We can think about each of these strategies as belonging to one of three major groups:

1. Basic authentication
2. X509 client certificates
3. Bearer tokens

How a user ultimately arrives at obtaining credentials will depend on the identity provider enabled by the cluster administrator, but the mechanism will adhere to one of these broad groups. And, while each of these mechanisms are vastly different in terms of how they are implemented, we will come to see how each will ultimately provide the API server with the data it needs to verify the authenticity and access levels of a user (by way of the `UserInfo` resource).

Basic Authentication

Basic Authentication is perhaps the most primitive of the authentication plugins available to a Kubernetes cluster. As you may have done on your own at some point Basic Authentication is a mechanism whereby the API client (typically `kubectl`), will set the HTTP Authorization header to a base64 hash of the combined username and password. As base64 is merely a hash, and provides no level of encryption for the transmitted credentials, it is imperative that any user Basic Authentication is used in conjunction with HTTPS.

To configure Basic Authentication on the API server, the administrator will need to supply a static file of usernames, passwords, a user ID, and a list of groups that this user should be associated with. The format is as such:

```
password,username,uid,"group1,group2,group3"
password,username,uid,"group1,group2,group3"
...
```

Note that the format of these lines matches the fields of the `UserInfo` resource.

This file is supplied to the Kubernetes API server by way of the `--basic-auth-file` command line

parameter. As the API server does not currently monitor this file for changes, any time a user is added, removed, or updated, the API server will need to be restarted in order for these changes to take effect. Because of this constraint, Basic Authentication would not normally be recommended for production clusters. This file may certainly be managed by an external entity (ie. configuration management tooling) in order to get to production-like configurations, but experience shows that this quickly becomes unsustainable.

These shortcomings aside, it should be noted that Basic Authentication can be a excellent tool for a quick-and-dirty test of a Kubernetes cluster. In the absence of a more elaborate authentication configuration, Basic Authentication allows a administrator to quickly experiment with features such as access control.

X509 Client Certificates

An authentication mechanism that is typically enabled by most of the Kubernetes installers is X509 client certificates. The reasons for this may be many, but this is almost certainly due to the fact that they are secure, ubiquitous, and may be generated (relatively) easily. If there is access to a signing certificate authority, new users may be created in short order.

When installing Kubernetes in a production-quality manner, we will want to be sure that not only are user-initiated requests transmitted securely, but that service-to-service communication is encrypted as well. X509 client certificates make perfect sense for this use case. So, if this is already a requirement, why not use it to authenticate users as well?

Well, this is precisely how the many of the installation tools work. For example, `kubeadm`, the community-supported installer, will create a self-signed root CA certificate and then use this for signing various certificates for the service components as well as single administrative certificate that it creates.

A single certificate for all of your users is not the best way to manage users within Kubernetes, but it will do for getting things up and running. When the need to onboard additional users arises, administrators may sign additional client certificates from this signing authority.

Note

As `kubeadm` is intended to be both an easy on-ramp for users to stand up a cluster, as well as a tool for building production-grade clusters, it is highly configurable. For instance, if the user requires the use of their own CA, they may configure `kubeadm` to sign certificates for both the service and user authentication requirements.

There are a variety of tools that can help an administrator create and manage client certificates. Some of the more popular choices are the openssl client tools and a utility from Cloud Flare named `cfssl`:[*https://github.com/cloudflare/cfssl*]. If you are already somewhat familiar with these tools, you will know that the command line options can sometimes be a bit cumbersome. We will focus on `cfssl` here as it has, in our opinion, a workflow that is a bit easier to grok.

We will assume that you already have an existing signing certificate authority. The first step is to

create a certificate signing request (CSR), that will be used to generate the client certificate. Again, we need to map a user's identity to a `UserInfo` resource. We can do so with the signing request. Here our specification of Common Name (CN) will map to username, and all Organization fields (O) will map to the groups that the user is a member of.

```
cat > joesmith-csr.json <<EOF
{
  "CN": "joesmith",
  "key": {
    "algo": "rsa",
    "size": 2048
  },
  "names": [
    {
      "C": "US",
      "L": "Boston",
      "O": "qa",
      "O": "infrastructure",
      "OU": "Acme Sprockets Company",
      "ST": "MA"
    }
  ]
}
```

In this case the user "joesmith" is a member of both "qa" and "infrastructure".

We can generate the certificate as follows:

```
cfssl gencert \
  -ca=ca.pem \
  -ca-key=ca-key.pem \
  -config=ca-config.json \
  -profile=kubernetes \
  joesmith-csr.json | cfssljson -bare admin
```

Enabling X509 client certificate authentication on the API server is a simple as specifying the `--client-ca-file=` the value of which will point at the certificate file on disk.

Even though `cfssl` eases the task of creating client certificates for an administrator, this means of authentication can still be a bit unwieldy. Just as with Basic Authentication, there are some drawbacks when a user is on-boarded, removed, or when a change is required (ie. adding the user to a new group). If certificates are chosen as an authentication option, administrators will, minimally, want to be sure that this process is automated in some fashion. Be sure that this automation includes a process for rotating certificates over time.

If the number of expected end users is quite low, or if the majority of users will interact with the cluster by way of some intermediary (ie. continuous delivery tools), X509 client certificates may very well be an adequate solution. If this is not the case, however, as an administrator you may find some of the token-based options to be a bit more flexible.

OpenID Connect

OpenID Connect (OIDC) is an authentication layer built on top of Oauth 2.0. With this authentication provider, the user will independently authenticate with a trusted identity provider. If this user successfully authenticates, the provider will then, through a series of web requests, provide the user with one or more tokens.

Note

Because this exchange of codes and tokens is somewhat complex and is not really pertinent to *how* Kubernetes will authenticate and authorize the user, we will focus on the desired state: where the user has authenticated and both an `id_token` and a `refresh_token` have been granted.

The tokens that are provided to the user will be provided in the RFC 7519 JSON Web Token (JWT) format. This open standard allows for the representation of user claims between multiple parties. Put more simply, with a trivial amount of human-parsable JSON, we can share information such as username, user ID, and groups this user may belong to. These tokens are authenticated with a Hash-based Message Authentication Code (HMAC), and are not encrypted. So, again, be sure that all communication including a JWT token is encrypted, preferably with TLS.

A typical token payload looks something like this:

```
{
  "iss": "https://auth.example.com",
  "sub": "Ch5hdXRoMHwMTYzOTgzZTdjN2EyNWQxMDViNjESBWF1N2Q2",
  "aud": "dDblg7xO7dks1uG6Op976jC7TjUZDCDz",
  "exp": 1517266346,
  "iat": 1517179946,
  "at_hash": "OjgZQ0vauibNVcXP52CtoQ",
  "username": "user",
  "email": "user@example.com",
  "email_verified": true,
  "groups": [
    "qa",
    "infrastructure"
  ]
}
```

The fields in this JSON document are called “claims” and they serve to identify various attributes of the user. While many of these claims are standardized (ie. `iss`, iat`, `exp`), identity providers may also add their own custom claims. Fortunately, the API server allows us to indicate how these claims will map back to our `UserInfo` resource.

To enable OIDC authentication on the API server we will need to add the `--oidc-issuer-url` and `--oidc-client-id` parameters on the command line. These are the URL of the identity provider and the ID of the client configuration, respectively, and both of these values will be provided by your provider. The two other options that we will likely want to configure, though this is not mandatory, are `--oidc-username-claim` (default:`sub`) and `--oidc-group-claim` (default: `groups`). If these defaults match the structure of your tokens, fantastic. If not, each of these will allow you to map claims on the identity provider to their respective `UserInfo` attributes.

Note

A fantastic tool for examining the structure of JWT tokens can be found at *https://jwt.io*. This tool from Auth0, not only allows you to paste your token for exploration of its contents, but also offers an in-depth reference of open source JWT signing and verification libraries.

This type of authentication is a bit different than the others that we have looked at in that it involves an intermediary. With Basic Authentication and X509 client certificates the Kubernetes API server is able to perform all of the steps required for authentication. With OIDC, the end

user will authenticate against our mutually trusted identity provider, and then use the tokens they have received to subsequently prove their identity to the API server. The flow looks something like this:

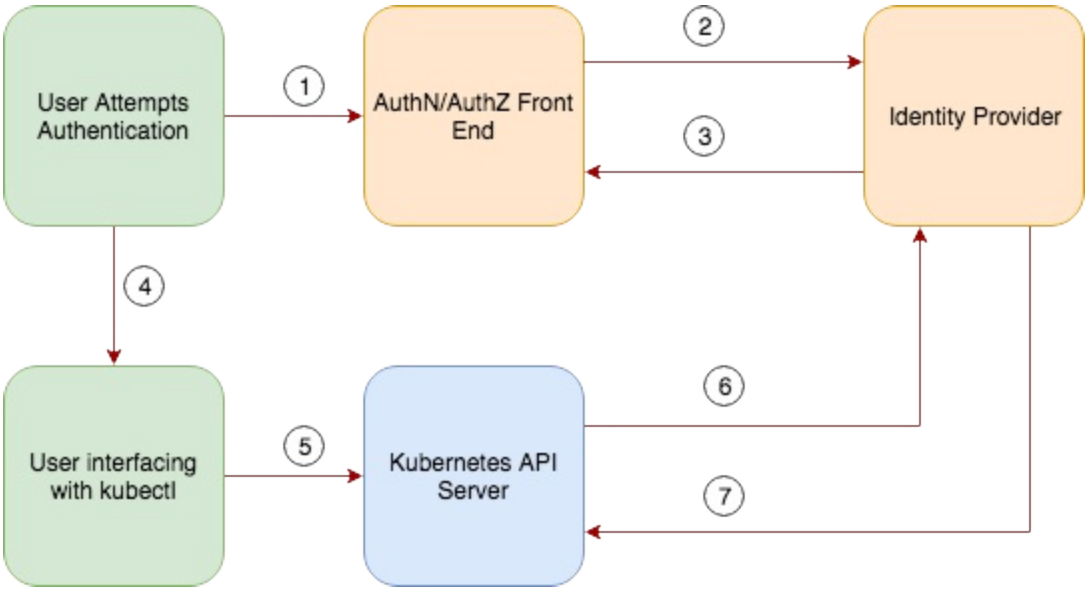


Figure 7-2. Kubernetes OIDC Flow

1. The user authenticates and authorizes the Kubernetes API server application.
2. The authentication front-end will pass the user's credentials on to the identity provider.
3. If the identity provider is able to authenticate the user, the provider will return an access code. This access code is then returned to the identity provider and exchanged for an identity token and (usually) a refresh token.
4. The user adds these tokens to their `kubeconfig` configuration.
5. Now that the `kubeconfig` file contains OIDC identity information, `kubectl` will attempt to inject the token as a bearer token on each Kubernetes API request. If the token is expired, it will first attempt to get a new identity token by exchanging the expired identity token with the issuer.
6. The Kubernetes API server ensures that this token is legitimate by requesting user information from the identity provider, based on the token credentials.
7. If the token is validated, the identity provider will return user information and the Kubernetes API server will allow the original Kubernetes API request to continue its flow.

Webhook

There are some scenarios wherein an administrator already has access to systems that are capable

of generating bearer tokens. You might be able to imagine a scenario where an in-house system grants a user a long-lived token that they may be able to use to authenticate to any number of systems within the environment. It may not be as elaborate or standards-compliant as OpenID Connect, but as long as we are able to programmatically challenge the authenticity of that token, Kubernetes will be able to verify the identity of a user.

With Webhook authentication in place, the API server will extract any bearer token present on an inbound request, and subsequently build a client POST request to the authentication service. The body of this request will be a JSON serialized `TokenReview` resource with the original bearer token embedded within.

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "spec": {
    "token": "some-bearer-token-string"
  }
}
```

Once the authenticating service evaluates this token for authenticity, it is then required to formulate its own response, again, with a `TokenReview` as the body. This response will indicate with a simple `true` or `false` whether the bearer token is legitimate. If the request fails authentication, the response is a simple one:

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": false
  }
}
```

Note

If there was an error in authenticating the user for some reason, the service may also respond with an `error` string field as a sibling to `authenticated`.

Conversely, if the response is that the authentication is successful, the provider should respond, minimally, with data about the user with an embedded `UserInfo` resource object. This object has fields for `username`, `uid`, `groups`, and even one for `extra` data the service may want to pass along.

```
{
  "apiVersion": "authentication.k8s.io/v1beta1",
  "kind": "TokenReview",
  "status": {
    "authenticated": true,
    "user": {
      "username": "janedoe@example.com",
      "uid": "42",
      "groups": [
        "developers",
        "qa"
      ],
      "extra": {
        "extrafield1": [
          "extravalue1",
          "extravalue2"
        ]
      }
    }
  }
}
```

Once the API has initiated the request and has received a response, the API server will grant or deny the Kubernetes API request in accordance with the guidance provided by the authentication service.

Note

One thing to keep in mind with nearly all of the token-based authentication mechanisms is that verification of the token often involves an additional request and response. In the case of both OpenID Connect and Webhook authentication, for instance, this additional round-trip to authenticate the token may become a performance bottleneck for the API request if the identity provider does not respond in timely fashion. With any of these plugins in-play be sure that you have low latency and performant providers.

Featured Project: dex

What happens when none of the above are appropriate for your use case? You may have noticed that commonly-utilized directory services are not included in the list of natively supported authentication plugins for Kubernetes. For example, at the time of this writing there are no connectors for Active Directory, LDAP, and others.

Of course you *could* always write your own authenticating proxy that would interface with these systems, but that would quickly become yet another piece of infrastructure to develop, manage and maintain.

Enter `dex`:[*https://github.com/coreos/dex*];

`dex` is a project from CoreOS that may be used as an OpenID Connect broker. In other words, `dex` provides a standards-compliant OIDC front-end to a variety of common backends. There is support for LDAP, Active Directory, SQL, SAML, and even SaaS service providers such as Github, Gitlab, and LinkedIn. Just imagine your delight when you receive that invite from your Kubernetes administrator:

```
I'd like to add you to my professional Kubernetes cluster network on LinkedIn.
```

Note

It is important to note that the authentication mechanisms configured in a Kubernetes cluster are not mutually exclusive. In fact, multiple plugins being enabled simultaneously is recommended.

As an administrator you may, for instance, configure both TLS client certificate and OIDC authentication simultaneously. While probably not appropriate to use multiple mechanisms on a daily basis, such a configuration may prove valuable when the need to debug a failing secondary API authentication mechanism. In this scenario an administrator would be able to leverage a well-known (and hopefully protected) certificate to garner additional data on the failure.

Note that when multiple authentication plugins are active at the same time, the first plugin to successfully authenticate a user will short-circuit the authentication process.

kubeconfig

With all of the authentication mechanisms described above, we will want to craft a `kubeconfig` file that records the details of how we authenticate. `kubectl` will use this configuration file to determine where and how to issue requests to the API server. This file is typically located in your home directory under `~/.kube/config`, but may also be specified explicitly on the command line with the `--kubeconfig` parameter or by way of the `KUBECONFIG` environment variable.

Whether or not you embed your credentials in your `kubeconfig` will depend on which authentication mechanism you will be using and possibly even your security stance. Remember that if you do embed credentials into this configuration file, they may be used by anyone who has access to this file. Treat this file as if it were a highly sensitive password, because it effectively is!

For someone who may not be familiar with a `kubeconfig` file, it is important to understand its three top-level structures: `users`, `clusters`, and `contexts`. With `users` we name a user and provide the mechanism by which they will authenticate to a cluster. The `clusters` attribute provides all of the data necessary to connect to a cluster. This, minimally, will include the IP or fully-qualified domain name for the API server, but may also include items like the CA bundle for a self-signed certificate. And, `contexts` is where we associate users with clusters as a single named entity. The `context` will serve as the means by which `kubectl` will connect and authenticate to an API server.

All of your credentials for all of your clusters may be represented with a single `kubeconfig` configuration. Best of all, this is all manipulated by way of few `kubectl` commands.

```
$ export KUBECONFIG=mykubeconfig
$ kubectl config set-credentials cluster-admin --username=admin --password=somepassword
User "cluster-admin" set.
$ kubectl config set-credentials regular-user --username=user --password=someotherpassword
User "regular-user" set.
$ kubectl config set-cluster cluster1 --server=https://10.1.1.3
Cluster "cluster1" set.
$ kubectl config set-cluster cluster2 --server=https://192.168.1.50
Cluster "cluster2" set.
$ kubectl config set-context cluster1-admin --cluster=cluster1 --user=cluster-admin
Context "cluster1-admin" created.
$ kubectl config set-context cluster1-regular --cluster=cluster1 --user=regular-user
Context "cluster1-regular" created.
$ kubectl config set-context cluster2-regular --cluster=cluster2 --user=regular-user
Context "cluster2-regular" created.
$ kubectl config view
apiVersion: v1
clusters:
- cluster:
    server: https://10.1.1.3
  name: cluster1
- cluster:
    server: https://192.168.1.50
  name: cluster2
contexts:
- context:
    cluster: cluster1
    user: cluster-admin
  name: cluster1-admin
- context:
    cluster: cluster1
    user: regular-user
  name: cluster1-regular
- context:
```

```
    cluster: cluster2
    user: regular-user
  name: cluster2-regular
current-context: ""
kind: Config
preferences: {}
users:
- name: cluster-admin
  user:
    password: somepassword
    username: admin
- name: regular-user
  user:
    password: someotherpassword
    username: user
```

Here we have created two user definitions, two cluster definitions, and three contexts. And, now, with just one more `kubectl` we can reset our context on-the-fly.

```
$ kubectl config use-context cluster2-regular
Switched to context "cluster2-regular".
```

This makes it extraordinarily simple to change from one cluster to the next, switch both cluster and user, or even impersonate a different user on the same cluster (something that is quite useful to have in an administrator's tool box).

While this was a very simple example utilizing basic authentication, users and clusters may be configured with all kinds of options. And, these configurations can become relatively complex. That said, this is a powerful tool made simple with a few command line operations. Utilize the contexts that make the most sense for your use case.

Service Accounts

So far in this chapter we have discussed how users will authenticate with the API. And, in that time, we have only really focused on authentication as it applies to a user that is external to a cluster. Perhaps this is you executing a `kubectl` command from your console or even with a click through the web interface.

There is another important use case to consider though, and this pertains to how the processes running inside a Pod will access the API. At first blush you might ask yourself, why a process running in the context of a Pod might require API access.

A Kubernetes cluster is a state machine comprised of a collection of controllers. Each of these controllers is responsible for reconciling state of the user-specified resources. So, in the most fundamental case, we will need to provide API access for any custom controllers that we intend to implement. But, access to the Kubernetes API from a controller is not the only use case. There are countless reasons why a Pod might require self-awareness or even awareness about the cluster as a whole.

The way that Kubernetes handles these use cases is by way of the `ServiceAccount` resource.

```
$ kubectl create sa testsa
$ kubectl get sa testsa -oyaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: testsa
  namespace: default
secrets:
- name: testsa-token-nr6md
```

You can think of `ServiceAccounts` as namespaced user accounts for all `Pod` resources.

In the output above you will notice that when we created the `ServiceAccount`, a `Secret` named `testsa-token-nr6md` was also created automatically. Just as with the end-user authentication we discussed earlier, this is the token that will be included as a `Bearer` token on every API request. These credentials are mounted into the `Pod` in a well-known location that is accessible by the various Kubernetes clients.

```
$ kubectl run busybox --image=busybox -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # ls -al /var/run/secrets/kubernetes.io/serviceaccount
total 4
drwxrwxrwt    3 root     root           140 Feb 11 20:17 .
drwxr-xr-x    3 root     root          4096 Feb 11 20:17 ..
drwxr-xr-x    2 root     root           100 Feb 11 20:17 ..2982_11_02_20_17_08.558803709
lrwxrwxrwx    1 root     root            31 Feb 11 20:17 ..data ->
..2982_11_02_20_17_08.558803709
lrwxrwxrwx    1 root     root            13 Feb 11 20:17 ca.crt -> ..data/ca.crt
lrwxrwxrwx    1 root     root            16 Feb 11 20:17 namespace -> ..data/namespace
lrwxrwxrwx    1 root     root            12 Feb 11 20:17 token -> ..data/token
```

Even though we are attempting to authenticate a process, we will again use JWT tokens, and the claims within will look a lot like what we saw in the end-user token scenarios. Recall that one of the API server's objectives is to map data about this user to a `UserInfo` resource, and this case is

no different.

```
{
  "iss": "kubernetes/serviceaccount",
  "kubernetes.io/serviceaccount/namespace": "default",
  "kubernetes.io/serviceaccount/secret.name": "testsa-token-nr6md",
  "kubernetes.io/serviceaccount/service-account.name": "testsa",
  "kubernetes.io/serviceaccount/service-account.uid": "23fe204f-0f66-11e8-85d0-080027da173d",
  "sub": "system:serviceaccount:default:testsa"
}
```

Every `Pod` that is launched will have an associated `ServiceAccount`.

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
spec:
  serviceAccountName: testpod-sa
```

If none is specified in the `Pod` manifest, a default service account will be used. This `default ServiceAccount` is available on a namespace-wide basis, and is created automatically when a `Namespace` is, itself, created.

Note

There are many scenarios where it is inappropriate from a security perspective to provide a `Pod` with access to the Kubernetes API. While it is not possible to prevent a `Pod` from having an associated `ServiceAccount`, in the next chapter will will explore how these use cases may be secured.

Summary

In this chapter we have covered the most commonly deployed end-user authentication mechanisms in Kubernetes. Hopefully one or more of these stood out as something you would be interested in enabling in your environment. If not, don’t despair as there are a handful of others (ie. static token files, authenticating proxies, etc.) that may be implemented. One, or more, of these will almost certainly fit your needs.

While you will want to perform the due diligence upfront to onboard your users in a secure and scalable manner, remember that just as with nearly everything in Kubernetes, these configurations may evolve over time. Use the solution that makes sense for your organization today, knowing that you may adopt additional capabilities seamlessly in the future.

Chapter 8. Authorization

Authentication is only the first challenge for a Kubernetes API request. As we introduced in the previous chapter, there are two additional tests for every request: access control and admission control. While authentication is a critical component for ensuring that only trusted users can effect change on a cluster, as we will come to learn in this chapter, authentication will also become the enabler for fine-grained control concerning *what* those users may do.

Beyond just verifying a user's authenticity and determining their levels of access, we will also want to be sure that every request will conform to our business needs. Every organization has some degree of standards that they have implemented. These policies and procedures help us make sense of the complex infrastructures that are required to bring applications to production environments, and we will see how Kubernetes stands in support of this with admission controllers.

REST

As we have already covered, the Kubernetes API is a RESTful API. The advantageous properties of a RESTful APIs are many (ie. scalability and portability), but its simple structure will be the enabler for how we will determine levels of access within Kubernetes.

For readers who may not be familiar with REST, the semantics are straightforward: resources are manipulated by way of verbs. Just as with natural language if we were to ask someone to "delete the Pod," we do so with a noun and a verb. REST API's function in the same way.

To illustrate this concept, let's look precisely at how `kubectl` requests information about a `Pod`. By simply increasing the log level using the `-v` option, we can get an in-depth view of the API calls that `kubectl` is making on our behalf.

```
$ kubectl -v=6 get po testpod
I0202 00:28:31.933993   17487 loader.go:357] Config loaded from file
    /home/ubuntu/.kube/config
I0202 00:28:31.994930   17487 round_trippers.go:436] GET
    https://10.0.0.1:6443/api/v1/namespaces/default/pods/testpod 200 OK in 41 milliseconds
```

In this case of this simple Pod information request, we can see that `kubectl` has issued a `GET` request (this is the verb) for the `pods/testpod` resource. You may also notice that there are other elements of the URL path, such as the version of the API as well as the namespace that we are querying (`default` in this case). These elements add additional context for our request, but it suffices to say that the resource and the verb are the primary actors here.

Those who have encountered REST before will be familiar with the four most basic verbs: create, read, update, and delete (affectionately known as "CRUD," for short). These four actions map directly to the HTTP verbs `POST`, `GET`, `PUT`, and `DELETE`, respectively, and these, in turn, make up the vast majority HTTP requests normally found on the Internet.

You may also notice that these verbs also look somewhat like the verbs we would use when dealing with Kubernetes resources, and you would be right. We can certainly create, delete, update, and even gather information about a Pod, for instance. Just as with HTTP, these four verbs constitute the most basic elements of how we would interact with Kubernetes resources, but in our case we are not limited to just these four. Within the Kubernetes API in addition to the `get`, `update`, `delete`, and `patch`, we also have access to the verbs, `list`, `watch`, `proxy`, `redirect`, and `deletecollection` when dealing with resources. These are the verbs that `kubectl` (and any client for that matter) is using under the covers on our behalf.

Resources in Kubernetes are all of those constructs that you know and love: `Pods`, `Services`, `Deployments`, etc. These are the constructs that we will manipulate by way of those verbs.

Authorization

Just because a user is authenticated does not mean that we should give equal access rights for all users. For example, we may have a scenario where we would like members of the web development team to have the ability to manipulate the `Deployments` serving web requests, but not the underlying `Pods` that serve as the units of compute for those `Deployments`. Or perhaps even within the web team itself, there is a contingent who may create resources and another group which may not. In short, we would like to determine which actions are permissible based upon who the user is and/or which groups they are a member of.

What we are describing here is known as authorization, and this will be the next challenge that Kubernetes will test for every API request. “Is this user allowed to perform this action?” is the question we are asking here.

Just as with authentication, all authorization challenges will be the responsibility of the API server. The API server may be configured to implement various authorization modules by way of the aptly-named, `--authorization-mode` argument to the `kube-apiserver` executable. This is a comma-delimited list of modules that will be evaluated in the order that they are specified.

The API server will pass each request to these modules in the order defined by the comma-delimited `--authorization-mode` argument. Each module, in turn, may either weigh-in on the decision making process or choose to abstain. In the case of abstinence, the API request will simply move on to the next module for evaluation. If, however, a module does make a decision, the authorization flow will be terminated and will reflect the decision of the authorizing module. If the module denies the request, the user will receive an appropriate `HTTP 403 Forbidden` response, and if the request is allowed, the request will make its way on to the final step of API flow: admission controller evaluation.

At the time of this writing, there are six authorization modules that may be configured. The simplest and most direct are the `AlwaysAllow` and `AlwaysDeny` modules, and just as the names suggest, these modules will allow or deny a request, respectively. Both of these modules are really only suited for test environments.

The `Node` authorization module is responsible for the authorization rules that we would like to apply to API requests made by worker nodes. Just like end-users, the `kubelet` processes on each of the nodes will perform a variety of API requests. For example, the `Node` status that is presented when you execute a `kubectl get nodes` is possible because the `kubelet` has provided its state to the API server with a `PATCH` request.

```
PATCH https://k8s.example.com:6443/api/v1/nodes/node1.example.com/status 200 OK
```

Obviously the `kubelet` should not have access to resources like our web service `Pods`. This module restricts the capabilities of the `kubelet` to the subset of requests necessary to maintain a functional worker `Node`.

Role-Based Access Control

The most effective means of user authorization in Kubernetes happens by way of the `RBAC` module. Short for Role-Based Access Control, this module allows for the implementation of dynamic access control polices at runtime.

Those who may be accustomed to this type of authorization from other frameworks maybe be groaning by now. How some of these frameworks have implemented Role-Based Access Control is all too often a complicated and convoluted process. When the means by which an administrator defines levels of access is tedious, there is the temptation to just provide very coarse grained access controls, if any at all. Worse, when the configuration of these controls is static or inflexible, you can almost guarantee that they will not be implemented effectively.

Fortunately, Kubernetes makes the definition and implementation of RBAC policies extraordinarily simple. To put the process succinctly, Kubernetes will map the attributes of the `UserInfo` object to which resources and verbs the user will have access to.

Role and ClusterRole

With the RBAC module, authorization to perform an action on a resource will be defined with the `Role` or `ClusterRole` resource types (we will dive into the difference between these resources shortly). To start, let's first focus only on the `Role` resource. An implementation of the policy outlined above (where a user has read-write access to `Deployments`, but only read access to `Pods`) might look something like this:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: web-rw-deployment
  namespace: some-web-app-ns
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list", "watch"]
- apiGroups: ["extensions", "apps"]
  resources: ["deployments"]
  verbs: ["get", "list", "watch", "create", "update", "patch", "delete"]
```

In this `Role` configuration we have created a policy that allows for read-type actions to be applied to `Pods` and for full read-write access rights for `Deployments`. This `Role` would enforce that all changes that happen to the child `Pods` of a `Deployment` happen at the `Deployment` level (ie. rolling-updates, scaling, etc.)

The `apiGroups` field of each rule simply indicates to the API server the namespace of the API that it should act on. (This will reflect the API namespace defined in the `apiVersion` field of your resource definition.)

In the next two fields, `resources` and `verbs`, we encounter those REST constructs we discussed earlier. And, in the case of RBAC, we will be explicitly allowing these types of API requests for a user with this `web-rw-deployment Role`. As `rules` is an array, we may add as many combinations of permissions as are appropriate. All of these permissions will be additive. With RBAC, we can only *grant* actions and this module will otherwise deny by default.

`Role` and `ClusterRole` are identical in functionality, and differ in their scope alone. In the above example you may notice that this policy is bound to the resources in the `some-web-app-ns` namespace. That means that this policy will only be applied to resources in that namespace.

If we would like to grant a permission that has cross-namespace capabilities, we will use the `ClusterRole` resource. These resources, in the same manner, grant fine-grained control but on a cluster-wide basis.

You might be wondering why someone would ever want to implement policies like this. `ClusterRoles` would typically be employed for two primary use cases: 1. to easily grant cluster administrators a wide degree of freedom 2. to grant very specific permissions to a Kubernetes controller

The first case is simple. We often want administrators to have broad access so that they can easily debug problems. Of course we could create a `Role` policy for every namespace that we

eventually create, but it may be more expedient to just grant this access with a `ClusterRole`. As these permissions are far-reaching, use this construct with caution.

Most Kubernetes controllers are interested in watching resources across namespaces, and then reconciling cluster state appropriately. We can use `ClusterRole` policies to ensure that controllers only have access to the resources they care about.

Note

All Kubernetes controllers (`Deployments`, `StatefulSets`, etc. are all examples) have the same basic structure. They are a state machine that watches the Kubernetes API for changes (additions, modification, and deletions), and seek to reconcile from the current state to the user-specified desired state.

Imagine a scenario where we wanted to create DNS records based upon a user-specified annotation on a `Service` or `Ingress` resource. Our controller would need to watch these resources and take action upon some sort of change. It would be insecure to give this controller access to other resources and inappropriate verbs (ie. `DELETE` on `Pods`). We can use a `ClusterRole` policy to provide the correct level of access as such:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: external-dns
rules:
- apiGroups: [""]
  resources: ["services"]
  verbs: ["get", "watch", "list"]
- apiGroups: ["extensions"]
  resources: ["ingresses"]
  verbs: ["get", "watch", "list"]
```

And this is exactly how the external-dns Kubernetes incubator project works: [https://github.com/kubernetes-incubator/external-dns]

With this `ClusterRole` policy in place, an `external-dns` controller can watch for additions, modifications, or even deletions of `Service` and `Ingress` resources and act on them accordingly. And, most importantly, these controllers will *not* have access to any other aspects of the API.

Note

Be especially careful to understand all implications when granting users access rights with RBAC. Always seek to give *only* the rights that are necessary, as this will reduce your security exposure significantly. Also understand that some rights grant implicit, and perhaps unintentional, rights to other resources. In particular, you should understand that granting `create` rights to a `Pod` effectively grants read access to more sensitive and related resources like `Secrets`. Because `Secrets` mounted or exposed via environment to a `Pod`, the `Pod create` rights will allow a `Pod` owner to read those `Secrets` unencrypted.

RoleBinding and ClusterRoleBinding

You'll notice that both `Role` and `ClusterRole` do not specify which users or groups to target with their rules. Polices alone are useless unless they are applied to a user or a group. To associate these policies with users, groups, or `ServiceAccounts`, we can use the `RoleBinding` and `ClusterRoleBinding` resources. The only difference here is whether we are trying to bind a `Role` or `ClusterRole`, respectively. Again, `RoleBindings` are namespaced.

`RoleBinding` and `ClusterRoleBinding` associate a policy with a `subject`.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: web-rw-deployment
  namespace: some-web-app-ns
subjects:
- kind: User
  name: "joesmith@example.com"
  apiGroup: rbac.authorization.k8s.io
- kind: Group
  name: "webdevs"
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: web-rw-deployment
  apiGroup: rbac.authorization.k8s.io
```

In this example we have associated the `web-rw-deployment Role` in the `some-web-app-ns` namespace to `joesmith@example.com` as well as to a group with the name `webdevs`.

As you'll recall from the previous chapter, the objective of every different type of authentication mechanism is two-fold: first ensure that the user's credentials match our expectations, and secondarily, obtain information about an authenticated user. This information is conveyed with the aptly-named `UserInfo` resource. The string values that we specify here are reflective of the user information obtained during authentication.

When it comes to authorization, there are three `subject` types that we may apply policy to: `Users`, `Groups`, and `ServiceAccounts`. In the case of `Users` and `Groups`, these will be defined by the `UserInfo username` and `groups` fields, respectively.

Note

The values of these fields are strings in the case of `username` and a list of strings for `groups`, and the comparison used for inclusion is a simple string match. These string values are up to you, and they can be any unique string values that your authorization system provides to identity a user or group.

`ServiceAccounts` will be specified explicitly with the (appropriately named) `ServiceAccount subject` type.

```
...
subjects:
- kind: ServiceAccount
  name: testsa
```

```
  namespace: some-web-app-ns
```

Remember that `ServiceAccounts` supply the Kubernetes API credentials for all running `Pod` processes. Every `Pod` will have an associated `ServiceAccount`, regardless of whether we specify which `serviceAccountName` to use in the `Pod` manifest. Left unattended, this could pose a significant security concern.

This concern can be largely mitigated with RBAC policies. As RBAC policies are default deny, it is recommended that any `Pod` that requires API capabilities have its own (or possibly shared) `ServiceAccount` with an associated fine-grained RBAC policy. Only grant this `ServiceAccount` the actions and resources that it requires to function properly.

Recall the `ClusterRole` `external-dns` example from above. Because the controller state machine will be issuing requests to the Kubernetes API from a `Pod` context, we can use a `ClusterRoleBinding` with a `ServiceAccount` subject to enable this functionality:

```
apiVersion: rbac.authorization.k8s.io/v1beta1
kind: ClusterRoleBinding
metadata:
  name: external-dns-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: external-dns
subjects:
- kind: ServiceAccount
  name: external-dns
  namespace: default
```

Testing Authorization

As the number of users, groups, and workloads on a Kubernetes cluster increases, so too does the complexity. While RBAC is a simple mechanism by which to apply authorization policies to a collection of subjects, implementing and debugging access rights can sometimes be tough. Fortunately, `kubectl` provides a handy resource for verifying our policies without needing to effect any real change on the cluster.

To test access, simply use `kubectl` to set the context of the user you'd like verify (or a user who is part of a group you'd like to check).

```
$ kubectl use-context admin
$ kubectl auth can-i get pod
yes
```

Here we can see that the `admin` user has `GET` access to the `Pod` resource in the `default` namespace.

After creating a much more restricted policy, where the user is restricted from creating any `Namespace` resources (which are scoped at the cluster), we use `can-i` to confirm this policy:

```
$ kubectl use-context basic-user
$ kubectl create namespace mynamespace
Error from server (Forbidden): namespaces is forbidden: User "basic-user"
cannot create namespaces at the cluster scope
$ kubectl auth can-i create namespace
no
```

Note that the user received the appropriate `Forbidden` when they first attempted to create the `Namespace mynamespace`.

Note

In this chapter we do not cover the Attribute-Based Access Control (ABAC) module. This module is semantically very similar to the RBAC module, with the exception that these policies are statically defined with a configuration file on each of the Kubernetes API servers. Just like some of the other file-based configuration items we have discussed, this policy is not dynamic. An administration will need to restart the `kube-apiserver` process each time they would like to modify this policy. This aspect makes it impractical for the more robust and production-ready RBAC module.

Summary

In this chapter we have covered the RESTful nature of the Kubernetes API and how it's structure lends itself well to policy enforcement. We have also come to understand how authorization directly relates to what we have already learned about authentication.

Authorization is one of the critical components necessary for the deployment of a secure, multi-tenant distributed system. With RBAC, Kubernetes affords us the capability to enforce both very coarse-grained, sweeping policies as well as those that are extremely specific to a user or group. And, because Kubernetes makes the definition and maintenance of these policies so trivial to implement, there is really no reason why even the most basic of deployments cannot make use of them. This is a perfect first step towards happy users, administrators, and auditors alike.

Chapter 9. Admission Control

As we have mentioned in the previous two chapters, Admission Control is the third phase of API request on-boarding. By the time we have reached this phase of an API request lifecycle, we have already determined that the request has come from a real, authenticated user, and that the user is authorized to perform this request. What we care about now, is whether the request meets the criteria for what we consider to be a valid request, and, if not, take an appropriate action. Should we flat-out reject the request or should we alter the request to meet our business standards? For those that may be familiar with the concept of API middleware, Admission Controllers are very similar in function.

While both Authentication and Admission control are both critical to a successful deployment, Admission Control is where, you, as an administrator, can really start to wrangle your user's workloads. Here you will be able to limit resources, enforce policies, and enable advanced features. This will help to drive up utilization, add some sanity to diverse workloads, and seamlessly integrate new technology.

Fortunately, just as with the other two phases, Kubernetes provides a wide array of admission capabilities right out of the box. While authentication and authorization don't change much between releases, admission control is quite the opposite. There is a seemingly never-ending list of capabilities that users are looking for when it comes to how they administer their clusters. And, because admission control is where most of that magic happens, it is no surprise that this componentry is continually evolving.

We could write books on the native admission control capabilities of Kubernetes. But, because that is not really practical, here we will focus on some of the more popular controllers as well as demonstrate how you can implement your own.

Configuration

Enabling admission control is extremely simple. As this is an API function, we will add `--enable-admission-plugins` flag to the `kube-apiserver` runtime parameters. This, like other configuration items, will be a comma-delimited list of the admission controllers that we want to enable.

Note

Prior to Kubernetes 1.10, the order that Admission Controllers were specified mattered. With the introduction of the `--enable-admission-plugins` command line parameter, this is no longer the case. For versions 1.9 and earlier, use the order-dependent `--admission-control` parameter.

Common Controllers

Much of the functionality that users take for granted in Kubernetes actually happens by way of admission controllers. If you have experienced the scenario where Pods that you have declared were automatically allocated a ServiceAccount, this happens by way of the ServiceAccount admission controller. Similarly, if you have tried to add new resources to a Namespace that is currently in a Terminating state, your request was likely rejected by the NamespaceLifecycle controller.

The Admission Controllers that are available from Kubernetes itself are often of two types: ensuring that sane defaults are utilized in the absence of being specified by an end-user, and for ensuring that users do not have more capabilities than they need. Many of the actions that a user is authorized to perform will be controlled with RBAC, but Admission Controllers allow an administrator to define additional fine-grained policies that extend beyond the simplistic resource, action, and subject policies offered by authorization.

Pod Security Policies

One of the most widely-utilized admission controllers is the PodSecurityPolicy controller. With this controller, administrators are able to specify the constraints that they demand of the processes under Kubernetes' control. With PodSecurityPolicies, administrators may enforce that Pods are not able to run in a privileged context, that they cannot bind to the hostNetwork, must run as a particular user, and that they are constrained by a variety of other security-focused attributes.

When PodSecurityPolicies are enabled, users will be unable to onboard new Pods unless there are authorized policies in place. Policies may be as permissive or restrictive as required by your organization's security posture. In production multi-user environments, administrators will want to use most of the policies offered by PodSecurityPolicies, as these will significantly improve overall cluster security.

Let's consider a simple, yet typical case, where we would like to ensure that Pods are not able to run in a privileged context. Defining the policy happens, as usual, by way of the Kubernetes API:

```
apiVersion: policy/v1beta1
kind: PodSecurityPolicy
metadata:
  name: non-privileged
spec:
  privileged: false
```

If you were to create this policy, apply it to the API server, and then attempt to create a conformant Pod, the request would be rejected as your user and/or the Service Account will not have permission to `use` the policy. To rectify this situation, simply create an RBAC Role that allows either of those Subject types to use this PodSecurityPolicy:

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: non-privileged-user
  namespace: user-namespace
rules:
- apiGroups: ['policy']
  resources: ['podsecuritypolicies']
  verbs:     ['use']
  resources:
  - non-privileged
```

and its RoleBinding:

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: non-privileged-user
  namespace: user-namespace
roleRef:
  kind: Role
  name: non-privileged-user
  apiGroup: rbac.authorization.k8s.io
subjects:
- kind: ServiceAccount
  name: some-service-account
  namespace: user-namespace
```

Once a user is authorized to `use` a PodSecurityPolicy, the Pod may be declared so long as it conforms to the policies defined.

Note

As PodSecurityPolicies are implemented as an Admission Controller (which enforce policy during the API request flow) Pods that have been scheduled prior to PodSecurityPolicy being enabled may be out of conformance. Keep this in mind, as restarts of those Pods may render them unable to be scheduled. Ideally, the PodSecurityPolicy admission controller is enabled at installation time.

ResourceQuota

Generally speaking, it is good practice to enforce quotas on your cluster. Quotas will ensure that no one user is able to utilize more than they have been allocated and is a critical component in driving up overall cluster utilization. If you intend to enforce user quotas, you will also want to enable the ResourceQuota controller.

This controller ensures that any new Pod's that are declared are first evaluated against the current quota utilization for the given namespace. By performing this check during workload onboarding we give immediate notice to a user that their Pod will or will not fit within the quota. Note, too, that when a Quota is defined for a namespace, all Pod definitions (even if originating from another resource such as Deployments or ReplicaSets) will be required to specify resource requests and limits.

Quotas may be implemented for an ever expanding list of resources, but some of the most common include cpu, memory, and volumes. It is also possible to place quotas on the number of distinct Kubernetes resources (ie. Pods, Deployments, Jobs, etc.) within a namespace.

Configuring quotas is straightforward:

```
$ cat quota.yml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: memoryquota
  namespace: memoryexample
spec:
  hard:
    requests.memory: 256Mi
    limits.memory: 512Mi
```

Now, if we try to exceed the limit, even with a single Pod, we will have our declaration immediately rejected by the ResourceQuota admission controller:

```
$ cat pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: memoryexample
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
    resources:
      limits:
        memory: 1Gi
      requests:
        memory: 512Mi
$ kubectl apply -f pod.yml
Error from server (Forbidden): error when creating "pod.yml": pods "nginx" is forbidden:
exceeded quota: memoryquota, requested: limits.memory=1Gi,requests.memory=512Mi,
used: limits.memory=0,requests.memory=0, limited: limits.memory=512Mi,requests.memory=256Mi
```

While somewhat less obvious, the same holds true for Pods created by way of higher order

resources, such as Deployments:

```
$ cat deployment.yml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: memoryexample
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
        resources:
          limits:
            memory: 256Mi
          requests:
            memory: 128Mi
$ kubectl apply -f deployment.yml
deployment.apps "nginx-deployment" configured
$ kubectl get po -n memoryexample
NAME                                READY     STATUS    RESTARTS   AGE
nginx-deployment-55dd98c6c8-9xmjn   1/1       Running   0          25s
nginx-deployment-55dd98c6c8-hc2pf   1/1       Running   0          24s
```

Even though we have specified 3 replicas, we could only satisfy 2 based upon Quota. If we `describe` the resulting ReplicaSet, we see the failure:

```
Warning  FailedCreate      3s (x4 over 3s)  replicaset-controller
(combined from similar events): Error creating: pods
"nginx-deployment-55dd98c6c8-tkrtz" is forbidden: exceeded quota:
memoryquota, requested: limits.memory=256Mi,requests.memory=128Mi,
used: limits.memory=512Mi,requests.memory=256Mi, limited:
limits.memory=512Mi,requests.memory=256Mi
```

Again, this error is originating from the ResourceQuota admission controller, but this time the error is somewhat hidden as it is being return to the Deployment's ReplicaSet (which is the creator of the Pods).

By now it is probably becoming clear that Quotas will quickly help you manage your resources effectively.

LimitRanger

Complementary to ResourceQuota, the LimitRanger admission controller is necessary if you have defined any LimitRange policies against a Namespace. A LimitRange, put simply, allows an administrator to place default resource limits for Pods that are declared as a member of a particular Namespace.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: default-mem
spec:
  limits:
  - default:
      memory: 1024Mi
    defaultRequest:
      memory: 512Mi
    type: Container
```

This capability becomes important in scenarios where Quotas have been defined. When Quotas are enabled, a user that has not defined resource limits on their Pod will have their request rejected. With the LimitRanger admission controller, a Pod with no resource limits defined will, instead, be given defaults (as defined by an administrator) and the Pod will be accepted.

Dynamic Admission Controllers

So far we have focused on the admission controllers that are available from Kubernetes itself. There may, however, be times where the native functionality just doesn't cut it. In scenarios like this, we will want to develop additional functionality that will help us meet our business objectives. Fortunately, Kubernetes supports a wide array of extensibility points, and this is also true for admission controllers.

Dynamic Admission Control is the mechanism by which we inject custom business logic into the admission control pipeline. There are two types of Dynamic Admission Control: validating and mutating.

With validating admission control, our business logic will simply accept or reject a user's request based upon our requirements. In the event of failure, an appropriate HTTP status code and reason for failure will be returned to the user. We will be placing the onus on the end user to declare conformant resource specifications, and hopefully doing so in a way that does not cause consternation.

In the mutating admission controller case, we will again be evaluating requests against the API server, but in this case we will be selectively altering the declaration to meet our objectives. In simplistic cases this may be something as simple as applying a series of well-known labels to the resource. In more elaborate cases, we may go so far as to transparently inject a sidecar container. While in this case we are taking on much of the burden for the end user, it can sometimes become a bit confusing for the user when they discover that some additional magic is happening under the covers. That said, this capability, if well-documented, can be critical for implementing advanced architectures.

In both cases, this functionality is implemented by way of user-defined webhooks. These downstream webhooks are called by the API server when it observes that a qualifying request has been made. (As we will see in the examples below, users are able to qualify requests in a fashion similar to how RBAC policies are defined.) The API server will `POST` an AdmissionReview object to these webhooks. The body of this request will include the original request, status of the object, and metadata about the requesting user.

In turn, the webhook will provide a simple AdmissionResponse object. This object includes fields for whether this request is allowed, a reason and code for failure, and even a field for what a mutating patch would look like.

In order to utilize Dynamic Admission Controllers, you must first configure the API server with a change the the `--enable-admission-plugins` parameter:

```
--enable-admission-plugins=...,MutatingAdmissionWebhook,ValidatingAdmissionWebhook
```

Note

Note that dynamic admission control, while extraordinarily powerful, is still somewhat early in

its maturity cycle. These features were alpha as of 1.8 and beta in 1.9. As with all new functionality, be sure to consult the Kubernetes documentation for additional recommendations regarding these extension points.

Validating Admission Controllers

Let’s take a look at how we can implement our own validating admission controller. We will reuse an example from above. Our controller will inspect all Pod `CREATE` requests to enforce that each Pod has an *environment* label, and that the label must have a value of *dev* or *prod*.

To demonstrate that you can write dynamic admission controllers in the language of your choice, we will use a Python Flask application for this example.

```
import json
import os

from flask import jsonify, Flask, request

app = Flask(__name__)

@app.route('/', methods=['POST'])
def validation():
    review = request.get_json()
    app.logger.info('Validating AdmissionReview request: %s',
                    json.dumps(review, indent=4))

    labels = review['request']['object']['metadata']['labels']
    response = {}
    msg = None
    if 'environment' not in list(labels):
        msg = "Every Pod requires an 'environment' label."
        response['allowed'] = False
    elif labels['environment'] not in ('dev', 'prod',):
        msg = "'environment' label must be one of 'dev' or 'prod'"
        response['allowed'] = False
    else:
        response['allowed'] = True

    status = {
        'metadata': {},
        'message': msg
    }
    response['status'] = status

    review['response'] = response
    return jsonify(review), 200

context = (
    os.environ.get('WEBHOOK_CERT', '/tls/webhook.crt'),
    os.environ.get('WEBHOOK_KEY', '/tls/webhook.key'),
)
app.run(host='0.0.0.0', port='443', debug=True, ssl_context=context)
```

We will containerize this application and make it available internally with a `ClusterIP` Service:

```
---
apiVersion: v1
kind: Pod
metadata:
  name: label-validation
  namespace: infrastructure
  labels:
    controller: label-validator
spec:
  containers:
  - name: label-validator
    image: label-validator:latest
    volumeMounts:
    - mountPath: /tls
      name: tls
  volumes:
```

```
  - name: tls
    secret:
      secretName: admission-tls
---
kind: Service
apiVersion: v1
metadata:
  name: label-validation
  namespace: infrastructure
spec:
  selector:
    controller: label-validator
  ports:
  - protocol: TCP
    port: 443
```

In this case, the webhook will be hosted on-cluster. For simplicity's sake we have used a standalone Pod, but there is no reason why this couldn't be deployed with something a bit more robust, like a Deployment. And, just as with any web service, we will secure it with TLS.

Once this Service becomes available, we need to direct the API server to call our webhook. We will indicate which resources and operations we care about, and the API server will only call this webhook when a request that meets this qualification is observed.

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: label-validation
webhooks:
- name: admission.example.com
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
  clientConfig:
    service:
      namespace: infrastructure
      name: label-validation
    caBundle: <base64 encoded bundle>
```

With a ValidatingWebhookConfiguration in place, we can now verify that our policy is working as expected. Attempting to apply a Pod without an *environment* label yields:

```
# kubectl apply -f pod.yaml
Error from server: error when creating "pod.yaml": admission webhook
"admission.example.com" denied the request: Every Pod requires an 'environment' label.
```

Similarly with an `environment=staging` label:

```
# kubectl apply -f pod.yaml
Error from server: error when creating "pod.yaml": admission webhook
"admission.example.com" denied the request: 'environment' label must be one of 'dev' or 'prod'
```

It is only when we add an environment label according to the specification that we are able to successfully create a new Pod.

Note

Notice that our application is being served over TLS. As API requests may certainly contain

sensitive information, all traffic should be encrypted.

Mutating Admission Controllers

If we modify the example above, we can easily develop a mutating webhook. Again, with a mutating webhook we will be attempting to alter the resource definition transparently for the user.

In this example we will inject a proxy sidecar container. While this sidecar is simply a helper nginx process, we could modify any aspect of the resource.

Note

Exercise care when modifying resources at runtime, as there may be existing logic that depends on well-defined and/or previously-defined values. A general rule of thumb is to only set previously unset fields. Always avoid altering any namespaced values (ie. resource annotations).

Our new webhook looks like this:

```
import base64
import json
import os

from flask import jsonify, Flask, request

app = Flask(__name__)

@app.route("/", methods=["POST"])
def mutation():
    review = request.get_json()
    app.logger.info("Mutating AdmissionReview request: %s",
                    json.dumps(review, indent=4))

    response = {}
    patch = [{
        'op': 'add',
        'path': '/spec/containers/0',
        'value': {
            'image': 'nginx',
            'name': 'proxy-sidecar',
        }
    }]
    response['allowed'] = True
    response['patch'] = base64.b64encode(json.dumps(patch))
    response['patchType'] = 'application/json-patch+json'

    review['response'] = response
    return jsonify(review), 200

context = (
    os.environ.get("WEBHOOK_CERT", "/tls/webhook.crt"),
    os.environ.get("WEBHOOK_KEY", "/tls/webhook.key"),
)
app.run(host='0.0.0.0', port='443', debug=True, ssl_context=context)
```

Here we use the JSON Patch syntax to add the proxy-sidecar to the Pod.

Just as with the validating webhook, we containerize the application and then dynamically configure the API server to forward requests on to the webhook. The only difference being that we will use a MutatingWebhookConfiguration and, naturally, point to the internal `ClusterIP` Service:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: pod-mutation
webhooks:
- name: admission.example.com
  rules:
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
  clientConfig:
    service:
      namespace: infrastructure
      name: pod-mutator
    caBundle: <base64 encoded bundle>
```

Now, when we apply a very simple, single-container Pod, we will get something a bit more:

```
# cat pod.yaml
---
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  labels:
    app: testpod
    environment: prod
#staging
spec:
  containers:
  - name: busybox
    image: busybox
    command: ['/bin/sleep', '3600']
```

Even though our Pod only declared the busybox container, we now have two containers at runtime:

```
# kubectl get pod testpod
NAME      READY     STATUS    RESTARTS   AGE
testpod   2/2       Running   0          1m
```

And a deeper inspection reveals that our sidecar was injected properly:

```
...
spec:
  containers:
  - image: nginx
    imagePullPolicy: Always
    name: proxy-sidecar
    resources: {}
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
  - command:
    - /bin/sleep
    - "3600"
    image: busybox
...
```

With mutating webhooks we have an extremely powerful tool for standardizing our user's declarations. Use this power with caution.

Summary

Admission Control is yet another tool for sanitizing your cluster's state. As this functionality is ever-evolving, be sure to check for new capabilities with every Kubernetes release. Be sure to implement those controllers that will help secure your environment and drive up utilization. And, where appropriate, don't be afraid to roll up your sleeves, implementing the logic that makes the most sense for your particular use cases.

Chapter 10. Networking

Just as with any distributed system, Kubernetes relies on the network in order provide connectivity between services as well as for connecting external users to exposed workloads.

Managing networking in traditional application architectures has always proven quite difficult. In many organizations there was a segregation of duties: developers would create their applications, and operators would be responsible for running them. As the application evolved, many times the needs from the networking infrastructure would drift. In the best of scenarios, the application simply would not operate, and an operator would take corrective action. But, in the worst of scenarios, significant gaps in areas like network security would arise.

Kubernetes allows developers to define network resources and policies that can live co-resident with their application deployment manifests. These resources and policies may be well-scoped by cluster administrators and can leverage any number of best-of-breed technology implementations by way of common abstraction layers. By removing developers from the nuts and bolts of how the network works, and by colocating the demands of the infrastructure with those of the application, we can have better assurances that our applications can be delivered in a consistent and secure manner.

Container Network Interface

Before we can speak about how to connect users with containerized workloads, we need to understand how Pods are able to communicate with other Pods. These Pods may be colocated on the same Node, across Nodes in the same subnet, and even on Nodes in different subnets that are, perhaps, even located in different data centers. Regardless of what the network plumbing looks like, we aim to connect Pods in a seamless, routable manner.

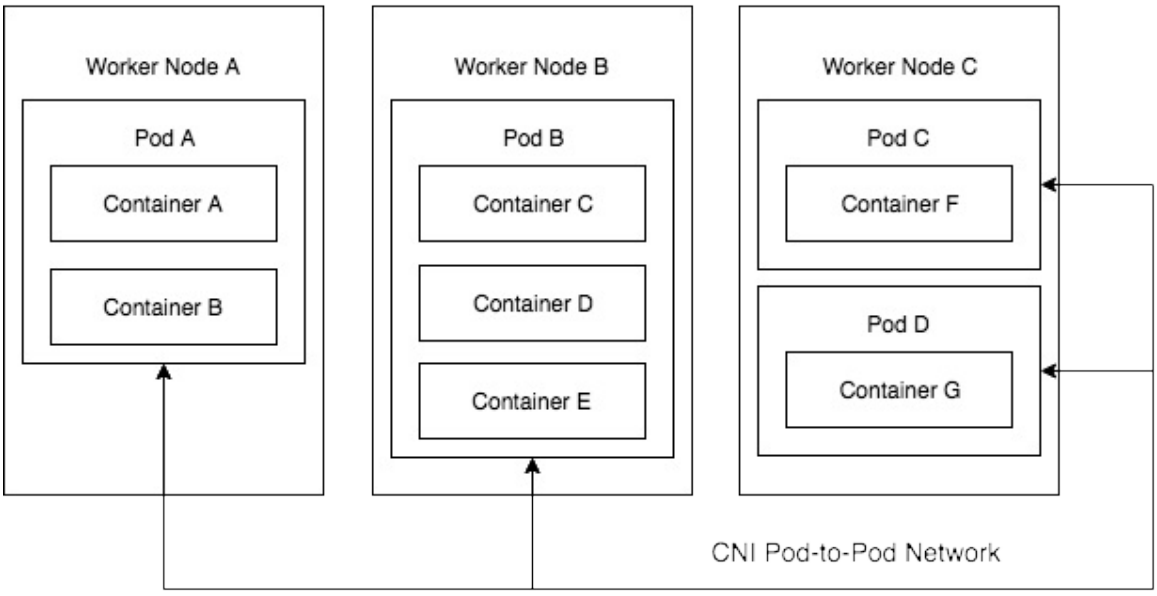


Figure 10-1. CNI Networking

Kubernetes interfaces with the network by way of the Container Network Interface (CNI) specification. The objective of this open specification is to standardize how container orchestration platforms connect containers with the underlying network, and to do so in a pluggable way. There are dozens of solutions, each with their own architectures and capabilities. Most are open source solutions, but there are also proprietary solutions from a number of different vendors within the Cloud Native ecosystem. Regardless of the environment in which you are deploying your cluster, there will certainly be a plugin that will meet your needs.

While there are multiple aspects to networking within Kubernetes, the the role of CNI is simply to facilitate Pod-to-Pod connectivity. The manner in which this happens is relatively simple. The container runtime (ie. Docker) will call the CNI plugin executable (ie. Calico) to add or remove an interface to or from the container's networking namespace. These are termed "sandbox" interfaces.

As you recall, every Pod is allocated an IP address, and the CNI plugin is responsible for it's allocation and assignment to a Pod.

Note

You may be asking yourself, “If a Pod can have multiple containers, how does CNI know which one to connect?” If you have ever interrogated Docker to list the containers running on a given Kubernetes Node, you may have noticed a number of `pause` containers associated with each of your Pods. These `pause` containers do nothing meaningful computationally. They merely serve as the placeholder for each Pod’s container network. As such, they are the first container to be launched and the last to die in the lifecycle of an individual Pod.

Once the plugin has executed the desired task on behalf of the container runtime, it returns the status of the execution just like any other Linux process: `0` for success and any other return code to indicate a failure. As part of a successful operation, the CNI plugin will also return the details of the IPs, routes, and DNS entries that were manipulated by the plugin in the process.

In addition to connecting a container to a network, CNI has capabilities for IP Address Management (IPAM). IPAM ensures that CNI always has a clear picture of which addresses are in use as well as those that are available for configuration of new interfaces.

Choosing a Plugin

When choosing a CNI plugin for use in your environment, there are two primary of considerations to keep in mind:

- What is the topology of your network?

The topology of your network will dictate a large part of what you will ultimately be able to deploy within your environment. For instance, if you are deploying to multiple availability zones within a public cloud, you will likely need to implement a plugin that has support for some form of encapsulation (also known as an overlay network).

- Which features are imperatives for your organization?

You will want to consider which features are important for your deployment. If there are hard requirements for mutual TLS between Pods, you may want to use a plugin that provides such a capability. By the same token, not every plugin provides support for NetworkPolicy. Be sure to evaluate the features that are offered by the plugin before you deploy your cluster.

Note

CNI is not the only mechanism for enforcing mutual TLS between Pods. With a sidecar pattern termed “Service Mesh” cluster administrators can require that workloads only communicate by a TLS enabled local proxy. Service mesh not only provides end-to-end encryption but may also enable higer-level features such as circuit breaking, blue-green deployments, and distributed tracing. It may also be enabled transparently for the end user.

kube-proxy

Even with Pod-to-Pod networking in place, Kubernetes would still be relatively primitive in terms of connectivity if it did not provide some additional abstractions over direct IP-to-IP connectivity. How would we handle the case where a Deployment has multiple replicas, and therefore, multiple serving IP's? Do we just pick one of the IP's and hope it doesn't get removed at some point in the future? Wouldn't it be nice to reference these replicas by a virtual IP? And, taking things one step further, wouldn't it be nice to have a DNS record?

All of this is possible with the Kubernetes Service resource that we covered in Chapter 2. With the Service resource we assign a virtual IP for network services exposed by a collection of Pods. The backing Pods are discovered and connected by way of a Pod selector.

Note

Many newcomers to Kubernetes typically think of the relationship between a collection of Pods (ie. a Deployment) and a Service as being one-to-one. Because Services are connected to Pods by way of label selectors, any Pod with the appropriate label will be considered a Service Endpoint. This functionality allows you to mix and match backing Pods, and can even enable advanced deployments such as blue-green and canary rollouts.

Under the covers, the Kubernetes component that is making all of this possible is the `kube-proxy` process. `kube-proxy` typically runs as a privileged container process, and it is responsible for managing the connectivity for these virtual Service IP addresses.

The name "proxy" is a misnomer of historical origin, wherein `kube-proxy` was originally implemented with a userspace proxy. This has since changed, and in the most common scenario, `kube-proxy` is simply manipulating `iptables` rules on every Node. These rules redirect traffic that is destined for a Service IP to any one of the backing Endpoint IP's. As `kube-proxy` is a controller, it will watch for state changes, and reconcile to the appropriate state upon any modifications.

If we take a look at a Service that is already defined in our cluster, we can get a sense of how `kube-proxy` works under the covers:

```
$ kubectl get svc -n kube-system kubernetes-dashboard
NAME                   TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes-dashboard   ClusterIP   10.104.154.139   <none>        443/TCP   40d
$ kubectl get ep -n kube-system kubernetes-dashboard
NAME                   ENDPOINTS                                     AGE
kubernetes-dashboard   192.168.63.200:8443,192.168.63.201:8443   40d
$ sudo iptables-save | grep KUBE | grep "kubernetes-dashboard"
-A KUBE-SEP-3HWS5OGCGRHMJ23K -s 192.168.63.201/32 -m comment --comment \
    "kube-system/kubernetes-dashboard:" -j KUBE-MARK-MASQ
-A KUBE-SEP-3HWS5OGCGRHMJ23K -p tcp -m comment --comment "kube-system/kubernetes-dashboard:" \
    -m tcp -j DNAT --to-destination 192.168.63.201:8443
-A KUBE-SEP-XWHZMKM53W55IFOX -s 192.168.63.200/32 -m comment --comment \
    "kube-system/kubernetes-dashboard:" -j KUBE-MARK-MASQ
-A KUBE-SEP-XWHZMKM53W55IFOX -p tcp -m comment --comment "kube-system/kubernetes-dashboard:"
    -m tcp -j DNAT --to-destination 192.168.63.200:8443
-A KUBE-SERVICES ! -s 192.168.0.0/16 -d 10.104.154.139/32 -p tcp -m comment --comment \
    "kube-system/kubernetes-dashboard: cluster IP" -m tcp --dport 443 -j KUBE-MARK-MASQ
-A KUBE-SERVICES -d 10.104.154.139/32 -p tcp -m comment --comment \
    "kube-system/kubernetes-dashboard: cluster IP" -m tcp --dport 443 \
```

```
    -j KUBE-SVC-XGLOHA7QRQ3V22RZ
-A KUBE-SVC-XGLOHA7QRQ3V22RZ -m comment --comment "kube-system/kubernetes-dashboard:" \
    -m statistic --mode random --probability 0.50000000000 -j KUBE-SEP-XWHZMKM53W55IFOX
-A KUBE-SVC-XGLOHA7QRQ3V22RZ -m comment --comment "kube-system/kubernetes-dashboard:" \
    -j KUBE-SEP-3HWS5OGCGRHMJ23K
```

This might be a bit hard to follow, so let us break it down. In the scenario we are looking at the “kubernetes-dashboard” ClusterIP Service. We see that it has a ClusterIP of 10.104.154.139 and Pod Endpoints at 192.168.63.200:8443 and 192.168.63.201:8443. What `kube-proxy` has done here is to create a number of `iptables` rules to reflect this state on each node. These rules, in effect, say that any packets coming from the Pod CIDR (192.168.0.0/16) destined for the dashboard ClusterIP (10.104.154.139/32) on TCP port 443 should be redirected, randomly, to one of the downstream Pods hosting the dashboard container on container port 8443.

In this way, every Pod on every Node will be able to communicate with defined Services by way of the `kube-proxy` daemon’s manipulation of `iptables` rules.

Note

`iptables` is the implementation most commonly found in the wild. With Kubernetes 1.9, a new IP Virtual Server (IPVS) implementation has been added. This is not only more performant, but also affords a variety of load balancing algorithms that may be utilized.

Service Discovery

In any environment where there is a high degree of dynamic process scheduling, we will want a means by which to reliably discover where service endpoints are located. This is true of many clustering technologies, and Kubernetes is no different. Fortunately, with the Service resource we have a good place from which to enable service discovery.

DNS

The most common way to discover Services within Kubernetes is via DNS. While there are no native DNS controllers within the Kubernetes componentry itself, there are add-on controllers that may be utilized for providing DNS records for Service resources.

The two most widely deployed add-ons in this space are the kube-dns and CoreDNS controllers that are maintained by the community. These controllers watch the Pod and Service state from the API server and, in turn, automatically define a number of different DNS records. The difference between these two controllers is primarily one of implementation: the CoreDNS controller uses CoreDNS (surprising, right?) as its implementation, whereas kube-dns leverages dnsmasq.

Every Service, upon creation, gets a DNS A record associated with the virtual Service IP that will take the form of `<service name>.<namespace>.svc.cluster.local`:

```
# kubectl get svc
NAME         TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)   AGE
kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP   35d
# kubectl run --image=alpine dns-test -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup kubernetes
Server:    10.96.0.10
Address 1: 10.96.0.10 kube-dns.kube-system.svc.cluster.local

Name:      kubernetes
Address 1: 10.96.0.1 kubernetes.default.svc.cluster.local
```

For headless Services the records are slightly different:

```
# kubectl run --image=alpine headless-test -it -- /bin/sh
If you don't see a command prompt, try pressing enter.
/ # nslookup kube-headless
Name:      kube-headless
Address 1: 192.168.136.154 ip-192-168-136-154.ec2.internal
Address 2: 192.168.241.42 ip-192-168-241-42.ec2.internal
```

in this case, instead of an A record for the Service clusterIP, the user is presented with a list of A records that they may use at their discretion.

Note

“Headless” Services are ClusterIP Services with `clusterIP=None`. These are used when you would like to define a Service but do not require that it is managed by `kube-poxy`. As you will still have access to the Endpoints for the Service, you may leverage this if you would like to implement your own service discovery mechanisms.

Environment Variables

In addition to DNS, a lesser used feature, but one to be aware of nonetheless, is service discovery by way of automatically injected environment variables. When a Pod is launched a collection of variables describing the ClusterIP Services in the current Namespace will be added to the process environment.

```
# kubectl get svc test
NAME      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
test      ClusterIP   10.102.163.244   <none>        8080/TCP   9m

TEST_SERVICE_PORT_8080_8080=8080
TEST_SERVICE_HOST=10.102.163.244
TEST_PORT_8080_TCP_ADDR=10.102.163.244
TEST_PORT_8080_TCP_PORT=8080
TEST_PORT_8080_TCP_PROTO=tcp
TEST_SERVICE_PORT=8080
TEST_PORT=tcp://10.102.163.244:8080
TEST_PORT_8080_TCP=tcp://10.102.163.244:8080
```

This mechanism may be used in the absence of DNS capabilities, but there is one important caveat to keep in mind. Because the process environment is populated at Pod startup time, any service discovery using this method would require that the required Service resources are defined *before* the Pod has been. This method does not account for any updates to a Service once the Pod has been started.

Network Policy

A critical aspect of securing user workloads, whether with Kubernetes or not, involves ensuring that services are only exposed to the appropriate consumers. If, for instance, you were developing an API that required a database backend, a typical deployment pattern would be to expose only the API endpoint to external consumers. Accessing the database would only be possible from the API service itself. This type of service isolation at layer 3 and layer 4 of the OSI model, helps to ensure that the surface area for attack is limited. Traditionally, these types of restrictions have been implemented with some type of firewall, and on Linux systems this policy is typically enforced with IPTables.

IPTables rules, under normal circumstances, are only manipulated by a server administrator and are local to the node that they are implemented on. This poses a bit of a problem for Kubernetes users that would like to have self-service capabilities for securing their services.

Fortunately, Kubernetes provides the NetworkPolicy resource for users to define layer 3 and layer 4 rules as they pertain to their own workloads. The NetworkPolicy resource offers both ingress and egress rules that may be applied to Namespaces, Pods, and even regular CIDR blocks.

Note

Note well that NetworkPolicy can only be defined in environments where the CNI plugin supports this functionality. The Kubernetes API server will gladly accept your NetworkPolicy declaration, but as there is no controller to reconcile the declared state, no policies will be enacted. For instance, Flannel is able to provide an overlay network for Pod-to-Pod communication, but it does not include a policy agent. For this reason, many who want the functionality of Flannel's overlay with NetworkPolicy capabilities have turned to Canal. Canal combines the overlay of Flannel with the policy engine from Calico.

A typical NetworkPolicy manifest may look something like so:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: backend-policy
  namespace: api-backend
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          project: api-midtier
    - podSelector:
        matchLabels:
          role: api-management
    ports:
    - protocol: TCP
```

```
      port: 3306
  egress:
  - to:
    - ipBlock:
        cidr: 10.3.4.5/32
    ports:
    - protocol: TCP
      port: 22
```

Reading and crafting these NetworkPolicy resources can take a bit of getting used to, but once you master the schema, this can be an extremely powerful tool at your disposal.

In the example above, we are declaring a policy that will be placed on all Pods with the `role=db` labels in the `api-backend` Namespace. The ingress rules in place allow for traffic to port 3306 from either a Namespace with the `project=api-midtier` label or from a Pod with the `role=api-management` label. Additionally, we are limiting the outbound, or egress, traffic from the `role=db` Pods to an ssh server at 10.3.4.5. Perhaps we use this for rsyncing backups to an externally available location.

While these rules are relatively specific, we can also create broad allow-all or deny-all policies, for both ingress and egress traffic, for any given Namespace. For example, the following policy (and perhaps the most interesting) creates default deny ingress policy for a Namespace:

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

Note

It is important to note that, by default, there are no network restrictions for Pods. It is only through NetworkPolicy that we can begin to lock down Pod interconnectivity.

Service Mesh

Understanding the network flows between workloads can be a complicated endeavor. In the simplest of cases, a single Pod replica with a single container is fronted by a Service resource. With this scenario we simply need to analyze where traffic is originating from; typically by looking at the container's application logs.

In a microservices environment, however, it is often typical for traffic to the enter the cluster via an Ingress, which is backed by a Service, which is then backed by any number of Pod replicas. Further, these Pods might, themselves, connect to other cluster Services and their respective backing Pods. As you can probably see, these flows get intricate very quickly, and this is where service mesh solutions may help.

A service mesh is simply a collection of "smart" proxies that are able to help users with a variety of east-west or Pod-to-Pod networking needs. These proxies may operate as sidecar containers in the application Pods, or may operate as DaemonSets where they are Node-local infrastructure components that may be utilized by any of the Pods on a given Node. Simply configure your Pods to proxy their traffic to these service mesh proxies (typically with environment variables), and your Pods are now a part of the mesh.

Note

Whether you deploy as a sidecar or as a DaemonSet will typically be determined by the service mesh technology that you choose and/or the availability of resources on your cluster. As these proxies run as Pods they do consume cluster resources and, as such, you will want to make decisions about whether these resources should be shared or associated with a Pod.

Service mesh solutions typically provide common functionality.

- Traffic Management - most service mesh solutions include some features targeted at driving incoming requests at particular Services. This can enable advanced patterns such as canary and blue/green deployments. Additionally, some solutions are protocol-aware. Instead of acting as a "dumb" layer 4 proxy, they have the ability to introspect higher-level protocols and make intelligent proxying decisions. For example, if a particular upstream were to respond slowly to HTTP requests, the proxy could weight that backend lower than a responsive upstream.

- Observability - when deploying microservices to a Kubernetes cluster, the interconnectivity between Pods can quickly become difficult to understand. As more and more Pods communicate with one another, how will you debug a user-reported connectivity issue? How will you find that application that is slow to respond? Most service mesh solutions provide automatic mechanisms for distributed tracing (and commonly based on the OpenTracing standard). In a transparent way you can uniquely trace the flow of individual requests.

- Security - in environments where the underlying network provides no default encryption (which is common for most CNI plugins), service mesh can intercede by offering mutual TLS for all east-west traffic. This can be advantageous because policies may be enforced such that all connectivity is secure by default.

Projects like Istio [1], Linkerd [2], and Conduit [3] are commonly-utilized service mesh solutions. If the features mentioned above speak to your use case, definitely give these projects some consideration.

Summary

Networking in any distributed system is always complex. Kubernetes simplifies this critical capability by offering well-conceived abstractions over multiple layers of the OSI networking stack. Often these abstractions are implemented with tried and tested networking technologies that have been reliably utilized for decades. But, because these abstractions are intended to provide a common interface for functionality, you as the cluster administrator, are free to utilize the implementations that best suit your needs. By coupling your application's networking requirements with its deployment manifests it will be much easier to deploy complex, stable and secure application architectures.

[1] *https://istio.io*

[2] *https://linkerd.io*

[3] *https://conduit.io*

Chapter 11. Monitoring Kubernetes

It’s all well and good to setup or use a Kubernetes cluster from a public cloud vendor, but without the right strategy for monitoring metrics and logs from that cluster and firing appropriate alerts when something goes wrong, the cluster that you have created is a disaster waiting to happen. While Kubernetes makes it easy for developers to build and deploy their applications, it also creates applications that are dependent on Kubernetes for successful operation. This means that when a cluster fails, your user’s applications will often fail as well. And if a cluster fails to often users will lose trust in the system and begin to question the value of the cluster and its operators. This chapter discusses approaches to developing and deploying monitoring and alerting for your Kubernetes cluster to prevent this from happening. Additionally, we’ll describe how you can add monitoring onto your cluster so that application developers can automatically take advantage of it for their own applications.

Goals for Monitoring

Before we step into the details of how to monitor your cluster, it's important to go over the goals for this monitoring. All of the specifics of how to deploy and manage monitoring are in service of these goals, and thus a crystal clear sense of "why?" will help in understanding the what.

Obviously, the first and foremost goal of monitoring is reliability. In this case, reliability is both the reliability of the Kubernetes cluster, and also the applications that are running on top of the cluster. As an example of this relationship, consider a binary like the controller-manager. If it stops operating correctly, service discovery will start to slowly go out of date. Existing services will have already been properly propogated to the DNS server in the cluster, but new services, or services that change due to roll-outs or scaling operations won't have their DNS updated.

Note

This failure shows the importance of understanding the underlying Kubernetes architecture. If you can't crisply explain the role of the controller manager in the overall Kubernetes cluster, this might be a good time to step back and review chapter three which covers Kubernetes components and architecture.

This sort of failure, actually won't be reflected in the correct operation of the Kubernetes cluster itself. All of it's service discovery is largely static after cluster initialization. However it will be reflected in the correctness of the applications running on top of the Kubrernetes cluster, as their Service discovery and failover, will itself start failing.

This example demonstrates two important points about Kubernetes monitoring. The first is that in many cases the cluster itself appears to be operating correctly, but it is actual failing. This points to the importance of not just monitoring the cluster pieces, but also the cluster functionality that users require. In this case, the best monitoring to catch this problem would be a "blackbox" monitor which continuously deploys a new Pod and Service, and validates that Service discovery works as expected.

Note

Throughout this chapter we will refer to two different types of monitoring. *Whitebox* monitoring looks at signals that applications produce, and uses these signals to find problems. *Blackbox* or *Probe* monitoring uses the public interfaces (e.g. the Kubernetes) to take actions with known expected outcomes (e.g. "create a ReplicaSet of size three leads to three pods") and fires alerts if the expected outcome does not occur.

The other important lesson of this DNS example is how important it is to have proper alerting in place. If you only notice that there is a problem with your Kubernetes cluster when you have a user complaining about failures of their application, you have a monitoring gap. While live site incidents (LSI) are an inevitable part of running a service, customer reported incidents (CRI) should be non-existent in a well monitored system.

In addition to reliability, another significant feature of a monitoring system is providing observability into your Kubernetes cluster. There are lots of reasons for why observing your cluster is important and relevant.

It is one thing to be able fire monitoring alerts that indicate that there is a problem with your cluster. Its another to be able to determine exactly what is going wrong to cause the alert, and yet another to be able to see and correct problems before they become end-user facing problems. The ability to observe, visualize and query your monitoring data is a critical tool in both determining what problems are happening, as well as identifying problems before they become incidents.

In addition to insight in the service of reliability, another important use case for cluster monitoring data is providing users with insight into the operation of the cluster. For example, a user may be curious to know on average how long it takes to pull and begin running their image. A user may be wondering how fast in practice a Kubernetes DNS record is created, or a person from finance may want to track if users are really using all of the compute resources they are requesting. All of this information is available from a cluster monitoring system.

Differences between logging and monitoring

One important topic to cover before we delve into the details of monitoring Kubernetes is the difference between logging and monitoring. Though closely related, they are actually quite different, used for different problems and often stored in different infrastructure.

Logging records events (e.g. a Pod being created, or an API call failing) while monitoring records statistics (the latency of a particular request, the CPU used by a process or the number of requests to a particular endpoint). Logged records by their nature are discrete, whereas monitoring data is a sampling of some continuous value.

Logging systems are generally used to search for relevant information (“Why did creating that Pod fail?” “Why didn’t that Service work correctly?”. For this reason, log storage systems are oriented around storing and querying vast quantities of data, whereas monitoring systems are generally geared around visualization (“Show me the CPU usage over the last hour.”) and thus are stored in systems that can efficiently store time-series data.

It is worth noting that neither logging nor monitoring alone are sufficient to understand your cluster. Monitoring data can give you a good sense for the overall health of your cluster, and help you identify anomalous events that may be occuring. Logging on the other-hand is critical for diving in and understanding what is actually happening, possibly across many machines, to cause such anomalous behavior.

Building a monitoring stack

Now that you have some understanding of why and what you may need to monitor your Kubernetes cluster, let’s take a look at how you might accomplish it.

Getting data from your cluster and applications

Monitoring begins with exposing data to the monitoring system. Some of this data is obtained generically from the kernel about the cgroups and namespaces that make up containers in your cluster, but the bulk of the information that is useful to monitor is added to the applications themselves by the developer. There are numerous different ways to integrate metrics into your application, but one of the most popular, and the choice of Kuberentes for exposing metrics, is the Prometheus (*https://prometheus.io*) monitoring interface.

Every server in Kubernetes exposes monitoring data via an HTTP(S) endpoint that serves the monitored data using the Prometheus protocol. If you have a Kubernetes kubelet server that is up and running, you can access this data via a standard http client like `curl` at the following url: *http://localhost:9093*

To integrate new application metrics into your code, you need to link in the relevant Prometheus libraries. This both adds the right HTTP server to your application, but it also exposes the specific metrics for scraping from that server. Prometheus has official libraries for Go, Java, Python and Ruby, and unofficial libraries for numerous other languages.

Here's an example of how you would instrument a Go application. First you add the prometheus server to your code:

```
import  "github.com/prometheus/client_golang/prometheus/promhttp"
...
func main() {
...
    http.Handle("/metrics", promhttp.Handler())
...
}
```

Once you have the server running, you need to define a metric and observe values:

```
        "github.com/prometheus/client_golang/prometheus"
        ...
           histogram := prometheus.NewHistogram(...)
        ...
        histogram.Observe(someLatency)
 ...
```

In addition to the monitoring information that Kubernetes makes available, each of the Kubernetes binaries logs a great deal of information to the `stdout` file stream. Often this output is captured and redirected to a log-rotated file such as /var/lib/kubernetes-apiserver.log. If you SSH into the master node running the Kubernetes API server, you can find the API server log file in /var/lib/kube-apiserver.log, and you can watch the log lines in action using the `tail -f …` command. The Kubernetes components use the `github.com/google/glog` library to log data to the file at various different severity levels. When looking through the file you can detect these severity levels by looking at the first letter that was logged. For example an Error level log looks like:

```
E0610 03:39:40.323732    1753 reflector.go:205] ...
```

You can see the *E* for the error log level, the time of the log, as well as the file that logged.

The Kubernetes components also log data at different levels of verbosity. Most installations of Kubernetes set the verbosity at two, which is also the default. This produces a good balance between verbosity and spam. If you need to increase or decrease the verbosity of the logging, you can use the `--v` flag and set it between zero and ten, where `ten` indicates maximum verbosity, and indeed it can be quite spammy. You can also use the `-vmodule` flag to set the verbosity for a particular file or set of files.

Note

Setting the verbosity of your logs to a higher level increases visibilty, but it comes with cost both financial and in performance. Because absolute number of logs is higher, it increases storage and retention costs, and can also make your querying slower. When increasing the logging level, generally only do it for a short period of time, and also ensure that you bring the logging back to a standard level as soon as possible.

Aggregating metrics and logs from multiple sources

Once you have your components generating data, you need a place to group it together, or aggregate it and once aggregated, store it for querying and introspection.. This section deals with aggregation of logging and monitoring data, while later sections deal with choices in terms of storage.

When it comes to aggregation there are two different styles of aggregation. There is pull aggregation where the aggregation system pro-actively reaches out to each monitored system and pulls the monitoring information into the aggregate store. An alternate approach is a push based monitoring system where the system that is being monitored is responsible for sending its metrics on to the aggregation system. Each of these two different monitoring designs has their proponents, and are easier or harder to implement depending on the details of the system. When you look at the two systems that we examine in detail for logging and monitoring aggregation, Prometheus and fluentd, the two systems have made different choices for this design. Understanding each and why they made those decisions helps you understand the design trade-off.

Prometheus is a pull based aggregator for monitoring metrics. As we have seen in the earlier section, when you expose a prometheus metric it is exposed as a web page that can be scraped and aggregated into the prometheus server. Prometheus chooses this design because it makes the adding more systems to monitor quite trivial. As long as the system implements the expected interface, it is as simple as adding an additional URL to the Prometheus configuration and that server's data will start being monitored. Because Prometheus is responsible for scraping the data and aggregating it at it's own pace, the Prometheus system doesn't need to worry about bursty or lagging clients sending it data at different intervals. Instead Prometheus always knows exactly what time it was when it requests the monitoring data, and it also is in control of the rate at which it samples data, ensuring that the sample rate is consistent and evenly distributed across all systems being monitored.

In contrast the `fluentd` daemon (*https://www.fluentd.org*) is a push based log aggregator. Just as Prometheus chose the pull model for a variety of pragmatic reasons, fluentd choses the push model for a number of real design considerations. The primary reason for fluent selecting a push based model is that nearly every system that logs does so to a file or to a stream on the local machine. As a result of that to integrate into the logging stack, fluentd needs to read from a large variety of files on disk to get the latest logs. It is generally not very possible to inject custom code into the system being monitored (e.g. to print to a logging system instead of stdout). Consequently, fluentd's chose to take control of the logging information, read it from each of the different files and push it into an aggregate log system, means that adding fluentd to an existing binary is quite straight-forward. You don't change the binary at all, instead you configure fluentd to load in data from a file in a specific path, and forward those logs on to a log storage system (more in for following section). To illustrate this, here is an example fluentd configuration to monitor and push kubernetes API-server audit logs:

```
<source>
       @type tail
       format json
```

```
        path /var/log/audit
        pos_file /var/log/audit.pos
        tag audit
        time_key time
        time_format %Y-%m-%dT%H:%M:%S.%N%z
    </source>
….
```

You can see from this example that fluentd is highly configurable, taking the file location, the file format, as well as an expression for parsing dates from the logs and a tag to add to the data. Because the Kubernetes servers use the `glog` package, log lines follow a consistent format, thus you can expect (and extract) structured data from every line that Kubernetes logs.

Note

Just because Prometheus is often linked into an application doesn’t mean that you can’t use it with off the shelf software. There are a wide variety of Prometheus adapters which can be run as side-cars next to your application which can be ambassadors betweeen the application and the expected prometheus interfaces. In many cases (e.g. Redis, Java) the adapter knows directly how to talk to the application to expose its data in a format which prometheus can understand. Additionally there are adapters from common monitoring protocols (e.g. StatsD) such that Prometheus can also scrape metrics which were originally intended for some other system.

Storing data for retreival and querying

Once monitoring and logging data has been aggregated by Prometheus or fluentd, the data still needs to be stored somewhere for retention across a period of time. How long you retain monitoring data is somewhat up to the needs of your system and the costs you're willing to pay in terms of storage space for the data. However, in our experience the minimum you should have is 30-45 days worth of data. At first blush, this might seem like a lot, but the truth is that many problems begin slowly and take a long time before they become apparent. Having a historical perspective allows you to see differences before they became significant problems, this allows you to more easily identify the source of the problem.

Consider for example, a release four weeks ago could have introduced some additional latency in request processing. That might not have been significant enough to cause problems, but when combined with a more recent increase in request traffic, now requests are being processed far to slowly and your alerts are firing. Without historical data to pinpoint the initial increase in latency four weeks ago, you wouldn't be able to identify the release (and thus the changes) that caused the problem, you would be stuck searching through the code looking for the issue, which can take significantly longer.

As with aggregators, there are a number of different choices for storing monitoring and logging data. Many of the storage options run as cloud services. These can be good choices as they eliminate operations for the storage part of your cluster, but there are also good reasons for running your own storage, e.g. being able to control data location and retention precisely. Even in the space of open source storage for logging and monitoring, you have multiple choices, but in the interests of time and space we discuss two of them here: InfluxDB for maintaining time series data and ElasticSearch for storing log-structured data.

InfluxDB

InfluxDB is a time-series database that is capable of storing large amounts of data in a compact and searchable format. It is an open source project that is freely available and can be installed on a variety of operating systems. InfluxDB is distributed as a binary package that can easily be installed, you can find those packages at *https://portal.influxdata.com/downloads#influxdb*

Note

A time-series is a collection of data-pairs where one member is a value and the other is an instant in time. For example, you might have a time series that represents the CPU usage of a process over time. Each pair would combine the CPU usage and the instant at which that CPU usage was observed.

One important question when running InfluxDB is whether or not to run it as a container on the Kubernetes cluster itself. In general this is not a recommended setup. You are using InfluxDB to monitor the cluster, thus you want monitoring data to continue to be accessible, even if the cluster itself is having problems.

ElasticSearch

ElasticSearch is a system for ingesting and searching log-based data. Unlike InfluxDB which is oriented towards storing time-series data, Elastic Search is designed to ingest large quantitys of unstructured or semi-structured log files and make them available via a search interface. ElasticSearch can be installed from binary packages available here *https://www.elastic.co/downloads/elasticsearch*.

Visualizing and interacting with your data

Of course, storing the information isn't very useful, if you can't then access it in interesting ways to analyze and understand what is going on in your system. To that end, visualization is a critical component in a complete monitoring stack. For logging and metric data, visualization is different. Metric monitoring data is generally visualized as graphs, either as a time-series that shows a few metrics over time, or a histogram which summarizes the statistics for a value across a time window. Sometimes it is visualized as an aggregate across a window of time (sum of all errors each hour for a week). One of the most popular interfaces for visualizing metrics is the open source Grafana dashboard (*https://grafana.com*) which can interface with Prometheus and other metric sources and enable you to build your own dashboards or import dashboards created by other users.

For logged data, the search interface is more oriented around ad-hoc queries and exploration of the data that has been logged. One of the popular interfaces for viewing logging data is the Kibana web frontend (*https://www.elastic.co/products/kibana*) which allows you to search, browse and introspect data that has been logged to elastic search.

What to monitor?

Now that you have assembled your monitoring stack, there is still one important part left unanswered: What to monitor? and correspondingly what to alert on?

When assembling monitoring information, as with nearly all software, it is valuable to take a layered approach. The layers to monitor are machines, cluster basics, cluster add-ons and finally user applications. In this way, beginning with the basics, each layer in the monitoring stack builds on top of the layer below it. When built this way, identifying a problem is an exercise in diving down through the layers until the cause is identified, but correspondingly if a healthy layer is reached (e.g. all of the machines in the cluster appear to be operating correctly) then it becomes obvious that the problem lies in the layer above (e.g. the cluster infrastructure)

The monitoring that was described in the previous paragraph was all *white box* monitoring, by which we mean that the monitoring was based upon detailed knowledge of the system and how it is assembled. Each part of the system is monitored for deviations from the expected and such deviations are reported.

The contrast to white box monitoring is *black box* or *prober* based monitoring. In black box monitoring you don't assume or know any details of how the system is constructed. Instead you simply consume the external interface, as a customer or user would, and observe if your actions have the expected results. For example, a simple prober for a Kubernetes cluster, might schedule a Pod onto the cluster and verify that the Pod was successfully created and the application running in the Pod (e.g. `nginx`) can be reached via a Kubernetes service. If a black-box monitor succeeds it can generally be assumed that the system is healthy. If the black-box fails, then clearly the system is not.

The value of black-box monitoring is that it gives you a very clear signal about the health of your system. The downside is that it gives you very little visibility into *why* your system has failed. Consequently it is essential to combine both white-box and black-box monitoring to have a robust, useful monitoring system.

Monitoring Machines

The machines (physical or virtual) that make up your cluster are the foundation of your Kubernetes cluster. If the machines in your cluster are overloaded or mis-behaving, all other operation within the cluster is suspect. Monitoring the machines is essential to understanding if your basic infrastructure is operating correctly.

Fortunately, monitoring machine metrics with Prometheus is quite straight-forward. The Prometheus project has a *Node Exporter* daemon (*https://github.com/prometheus/node_exporter*) which can run on each machine and which exposes basic information gathered from the kernel and other system sources so that Prometheus can scrape. This data includes:

- CPU Usage
- Network usage
- Disk usage and free space available
- Memory usage
- … and much, much more

You can download the node exporter from Github, or build it yourself. Once you have the node exporter binary in your system, you can set it to run automatically as a daemon using this simple `systemd` unit file:

```
[Unit]
Description=Node Exporter

[Service]
User=node_exporter
EnvironmentFile=/etc/sysconfig/node_exporter
ExecStart=/usr/sbin/node_exporter $OPTIONS

[Install]
WantedBy=multi-user.target
```

Once you have the node exporter up and running you can configure prometheus to scrape metrics from each machine in your cluster, using the following scrape configuration in Prometheus:

```
- job_name: "node"
   scrape_interval: "60s"
   static_configs:
   - targets:
     - 'server.1:9100'
     - 'server.2:9100'
     - ...
     - 'server.N:9100'
```

Monitoring Kubernetes

Fortunately all of the pieces of the Kubernetes infrastructure expose metrics using the Prometheus API, and there is also a Kubernetes service discovery which you can use to automatically discover and monitor the kubernetes components in your cluster:

```
- job_name: 'kubernetes-apiservers'
   kubernetes_sd_configs:
   - role: endpoints
 ….
```

You can re-use this service discovery implementation to add scraping for multiple different components in the cluster like the api servers and the kubelets.

Monitoring applications

Finally, you can also use the Kubernete service discovery to find and scrape metrics from Pods themselves. This means that you will automatically scrape metrics from pieces of the Kubernetes cluster designed to run as Pods (for example the kube-dns servers) and you will also automatically scrape all metrics from Pods that are run by users. Assuming, that is, that the users integrate and expose Prometheus compatible metrics. However, once your users see how easy it is to get automated metric monitoring via Prometheus, they are unlikely to use any other monitoring.

Black-box monitoring

As mentioned earlier, black-box monitoring will probe the external API of the system, ensuring that it responds correctly. In this case, the external API of the system is the Kubernetes API. Probing of the system can be performed by an agent that makes calls against the Kubernetes API. It is an interesting challenge to decide if this agent runs inside the Kubernetes cluster or not. Running it inside the cluster makes it significantly easier to manage, but it also makes it vulnerable to cluster failures. If you choose to monitor the cluster from within the cluster, it is essential to also have a “watchdog alert” which will fire if a prober hasn’t been run to completion in the previous N minutes. There are many different black-box tests you can design for the Kubernetes API.

As a simple example, you can imagine writing a small script:

```
#!/bin/bash

# exit on all failures
set -e

NAMESPACE=blackbox

# tear down the namespace no matter what
function teardown {
  kubectl delete namespace ${NAMESPACE}
}
trap teardown ERR

# Create a probe namespace
kubectl create namespace ${NAMESPACE}
kubectl create -f my-deployment-spec.yaml

# Test connectivity to your app here, wget etc.

teardown
```

You could run this script every five minutes (or some other interval) to validate that the cluster was working properly. A more complete example might be to write an application which continuously tested the Kubernetes API, similar to the script above, but also exported Prometheus metrics so that you could scrape the black box monitoring data into Prometheus.

Ultimately, the limits of what you want to black-box test are really the limits of your imagination, and willingness to design and build tests. As of the current writing, there are no good off-the-shelf black box probers for the Kubernetes API. You’re on your own to design and build such tests.

Streaming Logs

In addition to all of the metric monitoring data, it's also important to get the logs from your cluster. Thus includes things like the Kubelet logs from each node, as well as the API server, scheduler and controller manager logs from the master. These are generally located in `/var/log/kube-*.log`, you can set them up for export with a simple fluentd configuration like:

```
<source>
  @type tail
  path /var/log/kube-apiserver.log
  pos_file /var/log/fluentd-kube-apiserver.log.pos
  tag kube-apiserver
 ...
</source>
```

It is also useful to log anything that a container running in the cluster writes to stdout. By default Docker writes all of the logs from the containers to `/var/log/containers/*.log`, thus you can use that expression in a similar fluentd configuration to also export log data for all containers that run in the cluster.

Alerting

Once you have monitoring working correctly, its time to add alerts. Defining and implementing alerts in Prometheus or other systems is beyond the scope of this book, if you have never done monitoring before, we'd strongly recommend obtaining a full book on the subject.

However, when it comes to which alerts to define, there are two philsophies to consider. The first, similar to white-box monitoring, is to alert when signals stop being nominal. That is, have an understand of, for example, how much CPU an API server normally consumes, and alert if the CPU usage of an API server goes out of that range.

The benefits of this approach to monitoring are that you will often times notice problems before they are user impacting. Systems start to behave strangely or poorly often long before they have catastrophic failures.

The downside of this alerting strategy is that it can be quite noisy. Signals like CPU usage can be quite varied, and alerting when things change but there isn't necessarily a real problem can lead to tired, frustrated operators, who ignore pages when there are real alerts.

The alternate monitoring strategy, more similar to black-box monitoring, is to alert on the signals that your user sees. For example the latency of a request to the API server, or the number of 403 (unauthorized) responses that your API server is returning. The benefit of this alerting, is that by definition there can't be a noisy alert. Every time such an alert fires there is a real problem. The downside of such alerting is that you will not notice problems until they are customer facing.

Like everything, the best path for alerting lies with a balance of each. For signals that you understand very well, which have stable values, white-box alerting can give you a critical heads-up before a significant problem. While black-box alerting gives you high-quality alerts caused by real, user facing problems. A successful alerting strategy will combine both styles of alerts (and perhaps more critically adapt these alerts) as your understanding of your particular cluster(s) grows.

Summary

Logging and monitoring are critical components to understanding how your cluster and your applications are performing, and/or where they are having problems. Constructing a high-quality alerting and monitoring stack should be one of the very first priorities after a cluster is successfully set up. Done correctly, a logging and monitoring stack that is automatically available to the users of a Kubernetes cluster is a key differentiator that makes it feasible for developers to deploy and manage reliable applications at scale.

Chapter 12. Disaster Recovery

If you’re like most users, you have probably looked to Kubernetes, at least in part, for its ability to automatically recover from failure. And, of course, Kubernetes does a great job of keeping your workloads up and running. But, as with any complex system, there is always room for failure. Whether that failure is due to something like hardware fault on a Node, or even data loss on the etcd cluster, we will want to have systems in place to ensure that we can recover in a timely and reliable fashion.

High Availability

A first principle in any disaster recovery strategy is to design your systems such that you have minimized the possibility of failure in the first place. Naturally, designing a foolproof system is an impossibility, but we should always build with the worst-case scenarios in mind.

When building production-grade Kubernetes clusters best practices will always dictate that critical components are highly available. In some cases, as with the API server, these may have an active-active configuration, whereas with items like the scheduler and controller manager, these will operate in an active-passive manner. When these control plane surfaces are deployed properly, a user should not notice that a failure has even occurred.

Similarly, it is recommended that your etcd backing store is deployed in a 3 or 5-node cluster configuration. You may certainly deploy larger clusters (always with an odd number of members), but clusters of this size should suffice for the vast majority of use cases. The failure tolerance of the etcd cluster increases with the number of members that are present: 1 node failure tolerance for a 3-node cluster, and a 2-node tolerance for a 5-node cluster. But, as the size of the etcd cluster increases, the performance of the cluster may slowly degrade. When choosing your cluster size, always be sure that you are well within your expected etcd load.

Note

Understand that a failure of the Kubernetes control plane typically does not affect the data plane. In other words, if your API server, controller manager or scheduler fail, your Pods will often continue to operate as-is. In most scenarios you simply will not be able to effect change on the cluster until the control plane is brought back online.

State

The question at the center of every disaster recovery solution is, “How do I restore to a well-defined previous state?” We want to be sure that when disaster strikes we have copies of all of the data that we will need to return to an operational state.

Fortunately, with Kubernetes most of the cluster’s operational state is centrally located in the etcd cluster. Therefore, we will spend a good deal of time ensuring that we can reconstitute its contents, should a failure occur.

But, etcd is not the only state that we care about. We also need to be sure that we have backups of some of the static assets created (or, in some cases, provided) during deployment. Among the items that should be safely tucked away:

- all PKI assets used by the Kubernetes API server: These are typically located in the `/etc/kubernetes/pki` directory.
- any Secret encryption keys: These keys are stored in a static file that is specified with the `--experimental-encryption-provider-config` in the API server parameter. If these keys are lost, any Secret data would not be recoverable.
- any administrator credentials: Most deployment tools (kubeadm included) will create static administrator credentials and provide them in a kubeconfig file. While these may be recreated, securely storing them off-cluster might reduce recovery time.

Application Data

In addition to all of the state necessary to reconstitute Kubernetes itself, recovering stateful Pods would be useless unless we also recovered any persistent data associated with those Pods.

Persistent Volumes

As you are aware, there are a variety of ways that a user may persist data from within Kubernetes. How you back this data up will be contingent on your environment.

For instance, in a cloud provider it may be as simple as simply reattaching any persistent volumes to their respective Pods, with the working assumption that your Kubernetes failure is unrelated to the availability of persistent volumes. You might also rely on the underlying architecture backing the volumes themselves. For instance, with Ceph-based volumes, having multiple replicas of your data may be enough.

How you implement the backup of application data will depend heavily on the implementation that you have chosen for how volumes are presented to Kubernetes. Just keep this in mind as you develop a wider disaster recovery strategy.

Note

Kubernetes does not currently have a mechanism for defining volume snapshots, but this is a feature that seems to be getting traction in recent community conversations.

Local Data

One aspect of data backup that is often overlooked is that users sometimes unknowingly persist critical data to a Node's local disk. This is particularly common in on-premises environments, where network attached storage may not always be present. Without appropriate guard rails in place (ie. PodSecurityPolicies and/or more generic Admission Controllers), users might make use of `emptyDir` or `hostPath` volumes, possibly holding incorrect assumptions about the longevity of this data.

Note

Recall our discussion about Admission Control in Chapter 7. If you would like to enforce restrictions on local disk access, these may be implemented with PodSecurityPolicies, primarily with the `volumes` and `allowedHostPaths` controls.

It may not even be a failure scenario where this issue is encountered. Because worker nodes are widely considered to be ephemeral in nature, even a planned maintenance or retirement of a Node may yield a poor experience for your users. Always be sure to have the appropriate controls in place.

Worker Nodes

We can think of worker nodes as being replaceable. When designing our disaster recovery strategy for worker nodes, we simply need to have a process in place whereby we can recreate a worker node reliably. If you have deployed Kubernetes to a cloud provider, this task is often as simple as launching a new instance and joining that worker to the control plane. In bare metal environments (or those without API-directed infrastructure), this process may be a bit more onerous, but the process will be mostly identical.

In the event that you are able to identify that a Node is approaching failure, or in cases where you need to perform maintenance, Kubernetes offers two commands that may be of assistance.

First, and particularly important in high-churn clusters, `kubectl cordon` will render a Node unschedulable. This can help stem the tide of new Pods affecting our ability to perform a recovery action on a worker node. Second, the `kubectl drain` command will allow us to remove and reschedule all running Pods from a target node. This is useful in scenarios where we intend to remove a Node from the cluster.

etcd

Because an etcd cluster retains multiple replicas of its data set, complete failure is relatively rare. However, backing up etcd is always considered a best practice for production clusters.

Just as with any other database, etcd stores its data on disk, and with that comes a variety of ways that we can backup the etcd data. At the lowest levels we can use block and filesystem snapshots, and this might work well. However, there is a significant amount of coordination that would need to take place when attempting the backup. In both cases you would want to be sure that etcd has been quiesced, typically by stopping the etcd process on the etcd member you intend to perform the backup on. Further, to ensure that all in-flight data has been saved, you will want to be sure to first freeze the underlying filesystem. As you can see this is become pretty onerous.

Where this technique may start to make sense though, is with network attached block devices that are backing etcd. Many clusters that are built in public cloud environments choose to use this technique as it shortens the time to recovery. Instead of replacing the data on disk with a backup, these users simply reattach the existing etcd data volumes to the new etcd member nodes, and, fingers crossed, they are back in business. While this solution may work, there are a number of reasons why it may be less than ideal. Chief among them, are concerns surrounding data consistency, as this approach is relatively difficult to perform correctly.

The most common approach, albeit resulting in slightly longer recovery times, is to utilize the native etcd command line tools:

```
ETCDCTL_API=3 etcdctl --endpoints $ENDPOINT snapshot save etcd-`date +%Y%m%d`.db
```

This can be run against an active etcd member, and the resulting file should be offloaded from the cluster, and stored in a reliable location such as an object store.

In the event that you need to restore, you simply need to execute the (aptly named) `restore` command:

```
ETCDCTL_API=3 etcdctl snapshot restore etcd-$DATE.db --name $MEMBERNAME
```

against each of the replacement members of a new cluster.

While all of these backup strategies are viable, there are important caveats to be aware of.

First, when backing up by either method, be cognizant of the fact that you will be backing up the entire etcd keyspace. It will be a complete copy of the state of etcd at the time of backup. While, our goal is typically to create a carbon copy, there may be scenarios where we may not actually want the entire backup. Perhaps we simply want to bring the “production” Namespace up in an expeditious manner. With this type of recovery we will be restoring indiscriminately.

Second, just as with any type of database backup, if the consuming application (in our case Kubernetes itself), is not quiesced during backup, there may be transient state that has not been consistently applied to the backing store. The likelihood of this being highly problematic is

small, but present nonetheless.

And finally, if you have enabled any Aggregate API servers or have used an etcd-backed Calico implementation (both of which use their own etcd instances), these would not be backed up if you have only targeted the cluster's primary etcd endpoints. You would need to develop additional strategies to capture and restore that data.

Note

If you are using a managed Kubernetes offering, you may not have direct access to etcd or even the disks that are backing etcd. In this case, you will need to utilize a different backup and restore methodology.

Ark

A purpose-built tool that is widely used for backup and recovery of Kubernetes clusters, is Ark [1], from Heptio. This tool is not only concerned with the management of Kubernetes resource data, but also serves as a framework for managing application data.

What makes Ark different than the methods we have already described is that it is Kubernetes-aware. Instead of blindly backing up etcd data, Ark performs backups by way of the Kubernetes API itself. This ensures that the data is always consistent, and allows for more selective backup strategies. Let's consider a few examples.

- Partial Backup and Restore: Because Ark is Kubernetes-aware, it is able to facilitate more advanced backup strategies. For instance, if you were interested in backing up only production workloads, you could use a simple label selector: `ark backup create prod-backup --selector env=prod` This would backup all resources with the label `env=prod`.

- Restoration to New Environment: Ark is capable of restoring backup to an entirely new cluster, or even to a new Namespace within an existing cluster. Beyond the topic of disaster recovery, this may also be used to facilitate interesting testing scenarios.

- Partial Restoration: In the midst of downtime it is often preferable to restore the most critical systems first. With partial restoration Ark allows you to prioritize which resources are restored.

- Persistent Data Backup: Ark is able to integrate with a variety of cloud providers to automatically snapshot persistent volumes. Additionally, it includes a hook mechanism for performing actions such as filesystem freezing prior to and after the snapshot has been taken.

- Scheduled Backups: With an on-cluster service managing state, Ark is capable of scheduling backups. This can be particularly useful for ensuring that backups are taken regularly.

- Off-Cluster Backups: Ark integrates with various S3-compatible object storage solutions. While these solutions may be run on-cluster, it would be advisable to offload these backups so that they are available in the event of failure.

You likely will not need all of these features, but with the wide degree of freedom that Ark offers, you can choose the pieces that make sense for your backup solution.

Summary

When devising a disaster recovery strategy for your Kubernetes cluster, there will be many areas to consider. How you design this strategy will depend on your selections for complementary technologies as well as the details of your particular use case. As you build this muscle be sure to regularly exercise your ability to fully restore your production systems with fully automated solutions. This will not only prepare you for failure, but will also help you think about your deployment strategy more holistically. Of course, we hope you will never need to use the techniques outlined above, but should the need arise, you will be better off for having considered these cases upfront.

[1] *https://github.com/heptio/ark*

Chapter 13. Extending Kubernetes

Kubernetes has a rich API that provides much of the functionality that you might need to build and operate a distributed system. However the API is purposefully generic, aimed at the “eighty-percent” use cases. Taking advantage of the rich ecosystem of add-ons and extensions that exist for Kubernetes can add significant new functionality and enable new experience for users of your cluster. You may even chose to implement your own custom add-ons and extensions that are suited to the particular needs of your company or environment.

Kubernetes Extension Points

There are a number of different ways to extend a Kubernetes cluster, each offers a different set of capabilities and additional operational complexity. The following sections describe these various extension points in detail and provide insight into both how they can extend the functionality of a cluster, as well as the additional operational requirements of these extensions.

The four types of extensibility are:

- Cluster daemons for automation
- Cluster assistants for extended functionality
- Extending the lifecycle of the API Server
- Adding additional APIs

The truth, of course, about some of these classifications is that they are somewhat arbitrary and there are different extensions that may combine multiple different kinds of extensibility to provide additional functionality for a cluster. The categories described here are intended to help guide your discussion and planning for extending a Kubernetes cluster. They are guidelines, not hard and fast rules.

Cluster Daemons

The simplest and most common form of cluster extensibility is the cluster daemon. Just like a daemon or agent running on a single machine adds automation (for example log rolling) to a single machine, a cluster daemon adds automation functionality to a cluster. There are two important components to what it means to be a cluster daemon. The agent needs to run on the Kubernetes cluster itself, and the agent needs to add functionality to the cluster that is automatically provided to all users of the cluster without any action on their part.

To be able to deploy a cluster daemon onto the Kubernetes cluster it helps manage, the cluster daemon itself is packaged as a container image. It is then configured via Kubernetes configuration objects and run on the cluster as either via a DaemonSet or a Deployment. Typically these cluster daemons run in a dedicated namespace so that they are not accessible to users of the cluster, though in some cases users may install cluster daemons into their own namespaces. When it comes time to monitor, upgrade or otherwise maintain these daemons they are maintained exactly like any other application running on the Kubernetes cluster. Running agents in this manner is more reliable, since they inhereit all of the same capabilities that make running any other application in Kubernetes easier, and it is also more consistent, since both agents and applications are monitored and maintained using the same tools.

In the following sections, we will see additional ways in which programs running on a Kubernetes cluster can extend or enhance that cluster, but what distinguishes cluster agents or daemons from other extensions is that the capabilities they provide apply to all objects within a cluster or within a namespace without additional user interaction to enable them. They are “automagic” and users will often gain their functionality without even being aware that they are present.

Use cases for cluster daemons

There are many different sorts of functionality that you might want to provide to a user automatically. A great example is automatic metrics collection from servers that expose Prometheus. When you run Prometheus within a Kubernetes cluster and configure it to do Kubernetes based service discovery, it will operate as a cluster daemon and automatically scan all Pods in the cluster for metrics that it should ingest. It does this by watching the Kubernetes API server to discover any new Pods as they come and go. Thus, any application that is run within a Kubernetes cluster with a Prometheus cluster agent will automatically have metrics collected without any configuration or enablement by the developer.

Another example of a cluster daemon, is an agent that scans services deployed in the cluster for XSS vulnerabilities. This cluster daemon again watches the Kubernetes API server for when new Ingress (HTTP Load Balancer) services are created. When such services are created, it automatically scans all paths in the service for XSS vulnerable web pages and sends a report to the user. Again, because it is provided by a cluster daemon, this functionality is inherited by developers who use the cluster without any requirement that they even know what XSS is or that the scanning is occuring until they deploy a service that has a vulnerability. We'll see how to build this example in a section below.

Cluster daemons are powerful, because they add automatic functionality. The less a developer has to learn, but can inherit automatically from their environment the more likely their applications are to be reliable and secure.

Installing a cluster daemon

Installation of a cluster daemon is done via container images and Kubernetes configuration files. These configurations may be developed by the cluster adminstrator, provided by a package manager like Helm, or supplied by the developer of the service (for example an open source project or independent software vendor). Typically the cluster adminstrator will use the kubectl tool to install the cluster daemon on the cluster, possibly with some additional configuration information, such as a license key, or namespaces to scan. Once installed, the daemon immediately starts operation on the cluster, and any subsequent upgrades, repair or removal of the daemon is performed via Kubernetes configuration objects just like any other application.

Operational considerations for cluster daemons

While the installation of a cluster daemon is generally trivial, often just a single command line call, the operational complexity incurred by adding such a daemon can be quite significant. The automagic nature of cluster add-ons is a double-edged sword, for users will quickly come to rely on them, and thus the operational importance of a cluster daemon add-on can be quite significant. Another way of thinking about this, is that if the value of a cluster daemon is that the user doesn't even know they are there, the user is also unlikely to notice if the cluster daemon is failing. Imagine, for example, that your security regime is based on automated XSS scanning via a cluster daemon, and that daemon gets silently stuck. Suddenly all XSS detection for your entire cluster may be disabled. Installation of a cluster daemon shifts the responsibility for the reliability of these systems from the developer to the cluster adminstrator. Generally this is the right thing to do, since it centralizes knowledge of these extensions, and it allows for a single team to build services shared by a large number of other teams, but it is critical that the cluster adminstrator know what they are signing up for. A cluster adminstrator can not just install a cluster daemon on a whim or because of a user's request, they must be fully committed to operational management and support of that cluster daemon for the lifetime of the cluster.

Hands-on: Creating a cluster daemon

Creating a cluster daemon doesn't need to be hard, in fact a simple Bash script that you might run from a single machine can easily be transformed into a cluster daemon. Consider for example the following script:

```
#!/bin/bash
for service in $(kubectl --all-namespaces get services | awk '{print $0}'); do
  python XssPy.py -u ${service} -e
done
```

This script lists all services in a cluster, and then uses an open-source XSS scanning script (*https://github.com/faizann24/XssPy/blob/master/XssPy.py*) to scan each service and print out the report.

To turn this into a cluster daemon, we simply need to place this script in a loop (with some delays of course) and give it a way to report:

```
#!/bin/bash
# Start a simple web server
mkdir -p www
cd www
python -m SimpleHTTPServer 8080 &
cd ..

# Scan every service and write a report.
while true; do
  for service in $(kubectl --all-namespaces get services | awk '{print $0}'); do
    python XssPy.py -u ${service} -e > www/${service}-$(date).txt
  done
  # Sleep ten minutes between runs
  sleep 600
done
```

If you package this script up in a Pod and run it in your cluster, you will have a collection of XSS reports available from the Pod. Of course to really productionize this, there are many other things that you might need including uploading files to a central repository, or monitoring/alerting. But this example shows that building a cluster daemon does not have to be a complicated task for Kubernetes experts. A little shell script and a Pod are all you need.

Cluster Assistants

Cluster assistants are quite similar to cluster daemons, but unlike cluster daemons, where functionality is automatically enabled for all users of the cluster, a cluster assistant requires the user to provide some configuration or other gesture to opt-in to the functionality provided by the assistant. Rather than providing “automagic” experiences, cluster assistants provide enriched, yet easily accessible functionality to users of the cluster, but it is functionality that the user must be aware of, and provide appropriate information to enable.

Use cases for cluster assistants

The uses cases for cluster assistants are generally those where a user wants to enable some functionality, but the work to enable the capabilities is significantly harder, slower or more complicated and error prone than necessary. Given such a situation, it is the job of the assistant to "assist" (obviously) and automate this process to make it easier, more automatic and less likely to suffer from "cut and paste" or other configuration errors. Assistants simplify tedious or rote tasks in a cluster to easier to consume concepts.

As a concrete example of such a process, consider what is necessary to add an SSL certificate to an HTTP service in a Kubernetes cluster. First a user must obtain a certificate. Though APIs like Let's Encrypt have made this significantly easier, this is still a non-trivial task, requiring a user to install tooling, set up a server, claim a domain with Let's Encrypt. However, once the certificate is obtained, the developer still isn't done, they need to figure out how to deploy it into their web server. Some developers may follow best practices, and know about Kubernetes Ingress, make a Kubernetes Secret and associate the certificate with the HTTP load balancer. But other developers may take the easy (and dramatically less secure) route and bake the certificate directly into their container image. Still others may balk at the complexity and decide that SSL isn't actually required for their use case. Regardless of the outcome, the extra work by developers, and the different implementations of SSL are unnecessary risks. Instead, the addition of a cluster assitant to automate the process of provisioning and deploying SSL certificates can both reduce developer complexity as well as ensure that all certificates in the cluster are obtained, deployed and rotated in a manner that follows best practices. However, to operate correctly, the cluster assistant requires the knowlege and engagement of the end user of the cluster, in this case, the domain name for the certificate, and an explicit request for SSL to be attached to the load balancer via the cluster assistant. Such an assistant is implemented by the open source cert-manager project (*https://github.com/jetstack/cert-manager*)

For a cluster administrator, cluster assistants centralize knowledge and best practices, reduce questions from users by simplifying complex cluster configurations, and ensure that all services deployed to a cluster have a common look and feel.

Installing a cluster assistant

Because the difference between cluster assistants and cluster daemons comes from the patter of interaction, not the implementation, the installation of a cluster assistant is more or less identical to the installation of a cluster daemon. The cluster assistant is packaged as a container image and deployed via standard Kubernetes API objects like Deployments and Pods. Like cluster daemons, maintainence, operations and removal of the cluster assistants is managed via the Kubernetes API.

Operational considerations for cluster assistants

Like cluster daemons, cluster assistants need the cluster adminstrator to take on operational responsibility for the assistant. Because the assistants hide complexity from the end-user, meaning that the end-user is ultimately unaware of the details of how a task like installing a certificate is actually implemented, it is critical that the assistant's function correctly since the end-user is unlikely to be able to achieve similar tasks on their own, due to lack of experience and knowledge. However, because the functionality is opt-in a user is far more likely to notice that something isn't working. They requested an SSL certificate and it didn't arrive. However this doesn't mean that the cluster adminstrator has less of an operational burden, they still should be pro-actively monitoring and repairing cluster assistant infrastructure, but it does mean that someone is more likely to notice when things go wrong.

Hands-on Example of cluster assistants

To make this a little bit more concrete, we'll build an example cluster assistant that automatically adds authentication to a Kubernetes Service. The basic operation of this assistant is that it continuously scans the list of `Service` objects in your cluster looking for objects with a specific annotation key: `managing-k8s.io/authentication-secret`. It is expected that the value for this key points to a Kubernetes `Secret` which contains a `.htpasswd` file. For example:

```
kind: Service
metadata:
  name: my-service
  annotations:
    managing-k8s.io/authentication-secret: my-httpasswd-secret
...
```

When the cluster assistant finds such an annotation it will create two new Kubernetes objects. First it creates a `Deployment` which contains a replicated `nginx` web server Pod. These Pods, take the `.httpasswd` file that was referenced by the `Secret` in the annotation, and configure `nginx` as a reverse proxy which forwards traffic on to `my-service` but requires a user and password as specified in the `.htpasswd` file. The cluster assistant also creates a Kubernetes `Service` named `authenticated-my-service` that directs traffic to this authentication layer. That way a user can expose this authenticated service to the external world and have authentication without having to worry about how to configure `nginx`. Of course Basic Authentication is a pretty simple example, you can easily imagine extending it to cover OAuth or other more sophisticated authentication endpoints.

Extending the lifecycle of the API Server

The previous examples were applications that ran on top of your cluster, but there are limits to what is possible with such cluster extensions. A deeper sort of extensibility comes from extending the behavior of the API server itself. These extensions can be applied to all API requests directly as they are processed by the API server itself. This enables additional extensibility for your cluster.

Use cases for extending the API lifecycle

Because API lifecycle extensions exist in the path of the API server, you can use them to enforce requirements on all API objects that are created by the service. For example, suppose that you want to ensure that all container images that run in the cluster come from your company's private registry and that a naming convention is maintained. You might, for example, want all images to be of the form registry.my-co.com/<team-name>/<server-name>:<git-hash> where registry.my-co.com is a private image registry run by your company, <team-name> and <server-name> are well known teams and applications built by those teams, and finally <git-hash> is source-control commit hash indicating the revision from which the image was built. Requiring such an image name ensures that developers don't store their production images on public (unauthenticated) image repositories, and the naming conventions ensure that any application (for example the XSS scanner we described earlier) has access to meta-data that is needed to send notifications. Requiring the git-hash ensures that developers only build images from checked in (and therefore code-reviewed) source code and that it is easy to go from a running image, to the source code that it is running.

To implement this functionality we can register a custom admissions controller. Admission controllers were described earlier in the chapter on the "life of an API request" They are responsible for determining if an API request is accepted or "admitted" into the API server. In this case, we can register an admission controller that is run for all API objects that contain an "image" field (Pods, Deployments, DaemonSets, ReplicaSets and StatefulSets). The admission controller will introspect the image field in these objects and validate that they match the naming pattern described above, and the various components of the image name are valid (e.g. the team-name is associated with a known team, and the git-hash is one in a release branch of the team's repository.

Installation of API lifecycle extensions

There are two parts to installing an extension to the API lifecycle. Creating a service to handle the webhook calls, and second creating a new Kubernetes API object that adds the extension. To create the service that handles the web hook calls from the API server you need to create a web service that can respond appropriately. There are many ways to do this, from functions as a service (FaaS) from a cloud-provider, to FaaS implementations on the cluster itself (e.g. OpenFaaS) to a standard web application implemented in your favorite programming language. Depending on the requirements for the web-hook handler and the operations/cost requirements you can make different decisions. For example using a cloud-based FaaS might be the easiest in terms of setup and operations, but each invocation will cost some money, where-as if you already have an open source FaaS implementation running on your cluster, then that is a logical place to run your webhooks, but installing and maintaining an OSS FaaS might be more work that it's worth if you only have a few webhooks, where running a simple web server might be the right choice. You will need to make such choices as your situations warrant.

Operational Considerations for lifecycle extensions

From an operational standpoint, there are two complexities. The first, and more obvious, complexity comes from having to run a service to handle the web hook. The operational responsibility here varies as described previously depending on where you run the particular webhook, but regardless you will need to monitor your webhooks for at least application level reliability (e.g. not returning 500s) and perhaps more. The second operational complexity is more subtle, and it comes from having your own code injected into the critical path for the API server. If you implement a custom admission controller and it starts crashing and returning 500s, all requests to the API server that use this admission controller will start failing. Such an event could have significant impact on the correct operation of your cluster, it could cause a wide variety of failures that could affect the correct operation of applications deployed on top of the controller. In a less extreme case, your code could add extra latency to the API calls that it effects. This added latency could cause bottlenecks in other parts of the Kubernetes cluster (for example the controller manager or scheduler), or they might just make your cluster seem flaky or slow if your extension occasionally runs slow or fails. Whatever the case, placing code in the API server call path should be done carefully, and with monitoring, thought and planning to ensure that there are not any unanticipated consequences.

Hands-on Example of lifecycle extensions

To implement an admission controller you need to implement the admission control web hook. The admission control web hook receives an HTTP `POST` with a JSON body that contains an AdmissionReview, you can find the type definition here: *https://github.com/kubernetes/kubernetes/blob/master/pkg/apis/admission/types.go#L29*

We'll implement a simple Javascript service that admits Pods.

```
const http = require('http');

const isValid = (pod) => {
  // validate pod here
};

const server = http.createServer((request, response) => {
  var json = '';
  request.on('data', (data) => {
    json += data;
  });
  request.on('end', () => {
    var admissionReview = JSON.parse(json);
    var pod = admissionReview.request.object;

    var review = {
      kind: 'AdmissionReview',
      apiVersion: 'admission/v1beta1',
      response: {
        allowed: isValid(pod)
      }
    };
    response.end(JSON.stringify(review));
  });
});

server.listen(8080, (err) => {
  if (err) {
    return console.log('admission controller failed to start', err);
  }

  console.log('admission controller up and running.');
});
```

You can see that we take `AdmissionReview` objects, extract the `Pod` from the review, validate it, and then return an `AdmissionReview` object with a response filled in.

You can then register this dynamic admission controller with Kubernetes by creating the registration:

```
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: my-admission-controller
webhooks:
- name: my-web-hook
  rules:
  # register for create of v1/pod
  - apiGroups:
    - ""
    apiVersions:
    - v1
    operations:
    - CREATE
    resources:
    - pods
```

```
  clientConfig:
    service:
      # Send requests to a Service named 'my-admission-controller-service'
      # in the kube-system namespace
      namespace: kube-system
      name: my-admission-controller-service
```

As with all Kubernetes objects you can instantiate this dynamic admission controller registration with `kubectl create -f <web-hook-yaml-file>`. But make sure that the right service is up and running before you do so, or subsequent Pod creations may fail.

Adding custom APIs to Kubernetes

Though we've shown how you can extend the lifecycle of individual API requests to the Kubernetes API Server, this level of extensibility is still not capable of all the types of extensibility you might want to perform to your cluster. In particular, while Kubernetes comes with a rich set of API types that you can use to implement your application, sometimes you want to add new API types to the Kubernetes API. This dynamic type capability in Kubernetes allows you to take an existing cluster, with a collection of built-in API types like Pods, Services and Deployments and add new types that largely look and feel exactly as if they had been built in. This sort of extensibility is quite flexible and powerful, but it is also the most abstract and complicated to understand. At the highest level, you can think of this sort of extension as adding new API objects to the Kubernetes API server which look as if they have been compiled into Kubernetes, all of the tooling for handling existing Kubernetes objects will apply natively to these extensions.

Use cases for adding new APIs

Because custom API types are so flexible, they can literally represent any object, there are a large number of potential use cases, the following ones only just scratch the surface of these possibilities. In earlier sections we discussed open source implementations of functions as a service (FaaS) that run on top of Kubernetes. When a FaaS is installed on top of Kubernetes, it adds new functionality to the cluster. With this new functionality, you need an API to create, update and delete functions in the FaaS. Though you could implement your own new API for this FaaS, you would have to implement many of the things (authorization, authentication, error handling, etc) that the Kubernetes API server already has implemented for you. Consequently, it is far easier to model the functions provided by the FaaS as Kubernetes API extensions. Indeed this is what many of the popular open source FaaS do. When you install one on Kubernetes, they turn around and register new types with the Kubernetes API. Once these new types have been registered, all of the existing kubernetes tools (e.g. kubectl) apply directly to these new Function objects. This familiarity means that in many cases users of extended clusters won't even notice they're using API extensions.

Another popular use cases for API extensions has been the Operator pattern championed by CoreOS. With an operator, a new API object is introduced into the cluster to represent the (formerly human) "operator" of a piece of software. (e.g. a database adminstrator). To achieve this, a new API object is added to the Kubernetes API server that represents this "operated" piece of software. E.g. you might add a MySQLDatabase object to Kubernetes via API extensions. When a user creates a new instance of a MySQLDatabase, the operator would then use this API object to instantiate a new MySQLDatabase including appropriate monitoring and online supervision to automatically keep the database running correctly. Thus through operators and API extensibility, users of your cluster can directly instantiate "Databases" instead of Pods that happen to run databases.

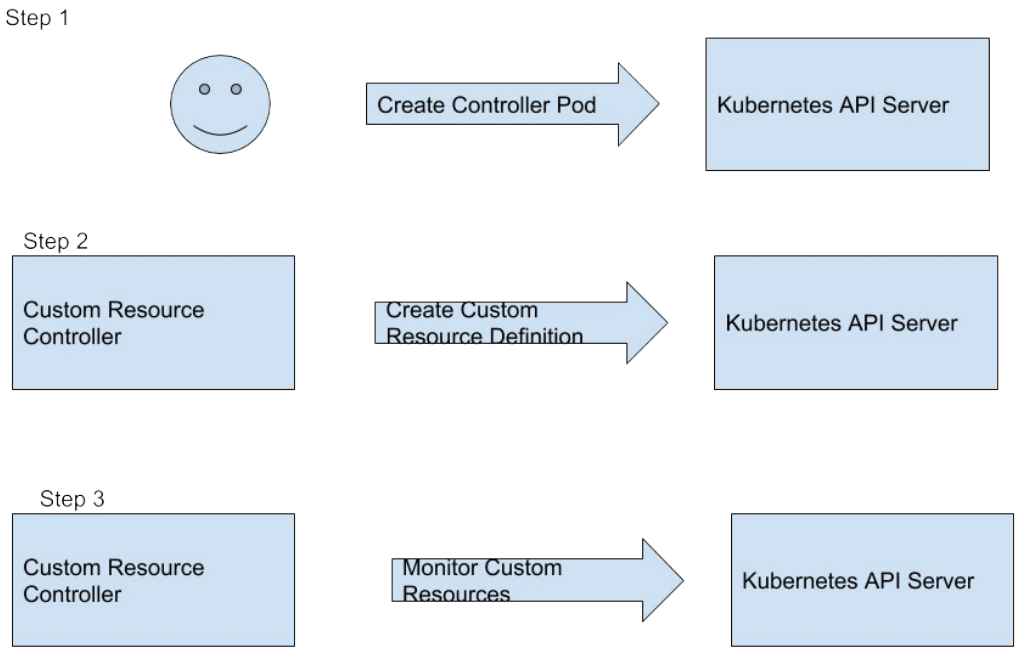
Custom resource definitions & Aggregated API Servers

Because both the API lifecycle and the process of extending the API is complicated technically, Kubernetes actually implements two separate mechanisms for adding new types to the Kubernetes API. The first is known as CustomResourceDefinitions and involves using the Kubernetes API itself to add new types to Kubernetes. All of the storage, and API serving associated with the new custom type are handled by Kubernetes itself. Because of this, custom resource definitions are by far a simpler way to extend the Kubernetes API server. On the other hand, because Kubernetes handles all of the extensibility, there are several limitations to these APIs, for example it is difficult to perform validation and defaulting for APIs added by custom resoruce definitions, however it is possible by combining custom resource definitions with a custom admission controller.

Because of these limitations, Kubernetes also supports API delegation, where the complete API call, including the storage of the resources is delegated to an alternate server. This enables the extension to implement an arbitrarily complex API, but it also comes with significant operational complexity, most especially the need to manage your own storage. Because of this complexity, most API extensions use custom resource definitions, and describing how to implement delegated API servers is beyond the scope of this book, the remainder of this section will describe how to use CustomResourceDefinitions to extend the Kubernetes API.

Architecture for custom resource definitions

There are several different pieces to implementing a custom resource definition, the first is the creation of the CustomResourceDefinition object itself. Custom resources are a built-in Kubernetes objects that carry the definition of the new type. After a CustomResourceDefinition is created, the Kubernetes API server programs itself with a new API group and resource path in the API web server, as well as new handlers that know how to serialize and deserialize these new custom resources from Kubernetes storage. If all you want is a simple CRUD (CReate, Update, Delete) API, then this may be sufficient, but in most cases, you want to actually do something when a user creates a new instance of your custom object. To do this, you need to combine the Kubernetes CustomResourceDefinition with a controller application which watches these custom definitions and then takes action based on resources the user creates, updates or destroys. In many cases this application server also is the application that registers the new custom resource definition. An diagram of this flow is show below:



The combination of a custom resource and a controller application is generally sufficient for many applications, but you may want to add even more functionality to your API, for example pre-create validation or defaulting. To do this, you can also add an admission controller for your newly defined custom resources that inserts itself in the API lifecycle as described in the previous chapter and adds these capabilities to your custom resource.

Installation of Custom resource definitions

Like all of the extensions, the code needed to manage these custom resources is run on the Kubernetes cluster itself. The custom resource controller is packaged as a container image and installed on the cluster using Kubernetes API objects. Because a custom resource is a more complicated extension, generally speaking the Kubernetes configuration consists of multiple objects packaged into a single YAML file. In many cases these files can be obtained from the open source project or software vendor supplying the extension, alternately they can be installed via a package manager like Helm. As with all of the other extensions, monitoring, maintainence and deletion of the custom resource API extensions occurs using the Kubernetes API.

Note

When a CustomResourceDefinition is deleted, *all* of the corresponding resources are also deleted from the clusters data store. They can not be recoverred. So when deleting a custom resource, be carefull and be sure to communicate with all end-users of that resource before you delete the CustomResourceDefinition.

Operational Considerations for custom resources

The operational considerations of a custom resource are generally the same as for other extensions, you are adding an application to your cluster that users will rely on that needs to be monitored and managed, furthermore, if your extension also uses an admission controller, the same operational concerns for admission controllers apply as well. However, in addition to these complexities described elsewhere, there is a significant additional complexity for custom resource definitions, and that is that they use the same storage associated with the all of the built-in Kubernetes API objects. As a result, it is possible to impact your API server and clusters operation by storing too many, or too large objects in the API server using custom resources. In general API objects in Kubernetes are intended to be simple configuration objects, they’re not intended to represent large data files. If you find yourself storing large amounts of data in custom API types, you should probably consider installing some sort of dedicated key-value store or other storage API.

Summary

Kubernetes is great, not just because of the value of the core APIs it provides, but also because of all of the dynamic extension points which allow users to customize their clusters to suit their needs. Whether it is via dynamic admission controllers to validate API objects, or new custom resource definition, there is a rich ecosystem of external add-ons that you can use to build a customized experience that fits your user's needs perfectly. And if a necessary extension doesn't exist, the knowledge in this chapter should help you design and build one.

Chapter 14. Conclusions

Kubernetes is a powerful tool that enables users to decouple from operating machines and focus on the core operations of their applications. This enables users to build, deploy and manage applications at scale significantly more easily and efficiently. Of course to achieve this, someone has to actually deploy and manage the Kubernetes cluster itself. For someone in that role, the application they focus on is Kubernetes itself.

We hope that the overview of the Kubernetes API and architecture, as well as coverage of topics like RBAC, upgrades, monitoring and extending Kubernetes give you the knowledge necessary to successfully deploy and operate Kuberentes so that your users don’t have to.

Index

About the Authors

Brendan Burns is a co-founder of the Kubernetes open source container management platform. He is currently a distinguished engineer at Microsoft running the Azure Resource Manager and Azure Container Service teams. Before Microsoft he was a senior staff engineer on the Google Cloud Platform. Prior to working in Cloud he developed web search backends that helped power Google search. Prior to that he was a Professor of Computer Science at Union College in Schenectady, NY. Brendan received a PhD in Computer Science from the University of Massachusetts Amherst and a BA from Williams College.

Craig Tracey has helped build the infrastructure that powers the Internet every day for the past 20 years. In this time he has had the opportunity to develop everything from kernel device drivers, to massive-scale cloud storage services, and even a few distributed compute platforms. Now as a Software Engineer turned Field Engineer at Heptio, he helps organizations accelerate their adoption of Kubernetes by teaching the principles of cloud native architectures through code.

Based in Boston, Massachusetts, in his free time, Craig loves playing hockey and exploring Europe. Craig holds a BS in Computer Science from Providence College.

Colophon

The animal on the cover of *FILL IN TITLE* is *FILL IN DESCRIPTION*.

Many of the animals on O’Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from *FILL IN CREDITS*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag’s Ubuntu Mono.