Kubernetes Administration

Ghazela Tech

Agenda

1. Introduction
2. Installation and configuration
3. Kubernetes Architecture
4. APIS
5. Managing State With Deployments
6. Services
7. Volumes And Data
8. INGRESS
9. SCHEDULING
10. Logging And Troubleshooting
11. Logging And Troubleshooting
12. Custom Resource Definitions
13. Kubernetes Federations
14. Security

Introduction

What is Kubernetes?

Running a container on a laptop is relatively simple. But, connecting containers across multiple hosts, scaling them, deploying applications without downtime, and service discovery among several aspects, can be difficult.

Kubernetes addresses those challenges from the start with a set of primitives and a powerful open and extensible API. The ability to add new objects and controllers allows easy customization for various production needs.

According to the kubernetes.io website, **Kubernetes** is:

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

A key aspect of Kubernetes is that it builds on 15 years of experience at Google in a project called **borg**.

Google's infrastructure started reaching high scale before virtual machines became pervasive in the datacenter, and containers provided a fine-grained solution for packing clusters efficiently. Efficiency in using clusters and managing distributed applications has been at the core of Google challenges.

In Greek, **κνβερνητης** means *the Helmsman*, or pilot of the ship. Keeping with the maritime theme of Docker containers, Kubernetes is the pilot of a ship of containers.

Due to the difficulty in pronouncing the name, many will use a nickname, **K8s**, as Kubernetes has eight letters. The nickname is said like **Kate's**.

Components of Kubernetes

kubernetes

Deploying containers and using Kubernetes may require a change in the development and the system administration approach to deploying applications. In a traditional environment, an application (such as a web server) would be a monolithic application placed on a dedicated server. As the web traffic increases, the application would be tuned, and perhaps moved to bigger and bigger hardware. After a couple of years, a lot of customization may have been done in order to meet the current web traffic needs.

Instead of using a large server, Kubernetes approaches the same issue by deploying a large number of small web servers, or microservices. The server and client sides of the application expect that there are many possible agents available to respond to a request. It is also important that clients expect the server processes to die and be replaced, leading to a transient server deployment. Instead of a large Apache web server with many httpd daemons responding to page requests, there would be many nginx servers, each responding.

The transient nature of smaller services also allows for decoupling. Each aspect of the traditional application is replaced with a dedicated, but transient, microservice or agent. To join these agents, or their replacements together, we use services and API calls. A service ties traffic from one agent to another (for example, a frontend web server to a backend database) and handles new IP or other information, should either one die and be replaced.

Communication to, as well as internally, between components is API call-driven, which allows for flexibility. Configuration information is stored in a JSON format, but is most often written in YAML. Kubernetes agents convert the YAML to JSON prior to persistence to the database.

Kubernetes is written in Go Language, a portable language which is like a hybridization between C++, Python, and Java. Some claim it incorporates the best (while some claim the worst) parts of each.

Challenges

Containers have seen a huge rejuvenation in the past few years. They provide a great way to package, ship, and run applications - that is the **Docker** motto.

The developer experience has been boosted tremendously thanks to containers. Containers, and **Docker** specifically, have empowered developers with ease of building container images, simplicity of sharing images via **Docker** registries, and providing a powerful user experience to manage containers.

However, managing containers at scale and architecting a distributed application based on microservices' principles is still challenging.

You first need a continuous integration pipeline to build your container images, test them, and verify them. Then, you need a cluster of machines acting as your base infrastructure on which to run your containers. You also need a system to launch your containers, and watch over them when things fail and self-heal. You must be able to perform rolling updates and rollbacks, and eventually tear down the resource when no longer needed.

All of these actions require flexible, scalable, and easy-to-use network and storage. As containers are launched on any worker node, the network must join the resource to other containers, while still keeping the traffic secure from others. We also need a storage structure which provides and keeps or recycles storage in a seamless manner.

One of the biggest challenges to adoption is the applications themselves, inside the container. They need to be written, or re-written, to be truely transient. If you were to deploy Chaos Monkey, which would terminate any containers, would your customers notice?

Other solutions

Built on open source and easily extensible, **Kubernetes** is definitely a solution to manage containerized applications.

There are other solutions as well, including:

- **Docker Swarm** is the Docker Inc. solution. It has been re-architected recently and is based on SwarmKit. It is embedded with the **Docker Engine**.
- **Apache Mesos** is a data center scheduler, which can run containers through the use of *frameworks*. Marathon is the framework that lets you orchestrate containers.
- **Nomad** from HashiCorp, the makers of **Vagrant** and **Consul**, is another solution for managing containerized applications. **Nomad** schedules tasks defined in *Jobs*. It has a **Docker** driver which lets you define a running container as a task.
- **Rancher** is a container orchestrator-agnostic system, which provides a single pane of glass interface to managing applications. It supports **Mesos**, **Swarm**, **Kubernetes**.

The Borg Heritage

What primarily distinguishes **Kubernetes** from other systems is its heritage. **Kubernetes** is inspired by **Borg** - the internal system used by Google to manage its applications (e.g. **Gmail**, **Apps**, **GCE**).

With Google pouring the valuable lessons they learned from writing and operating **Borg** for over 15 years into **Kubernetes**, this makes **Kubernetes** a safe choice when having to decide on what system to use to manage containers. While a powerful tool, part of the current growth in Kubernetes is making it easier to work with and handle workloads not found in a Google data center.

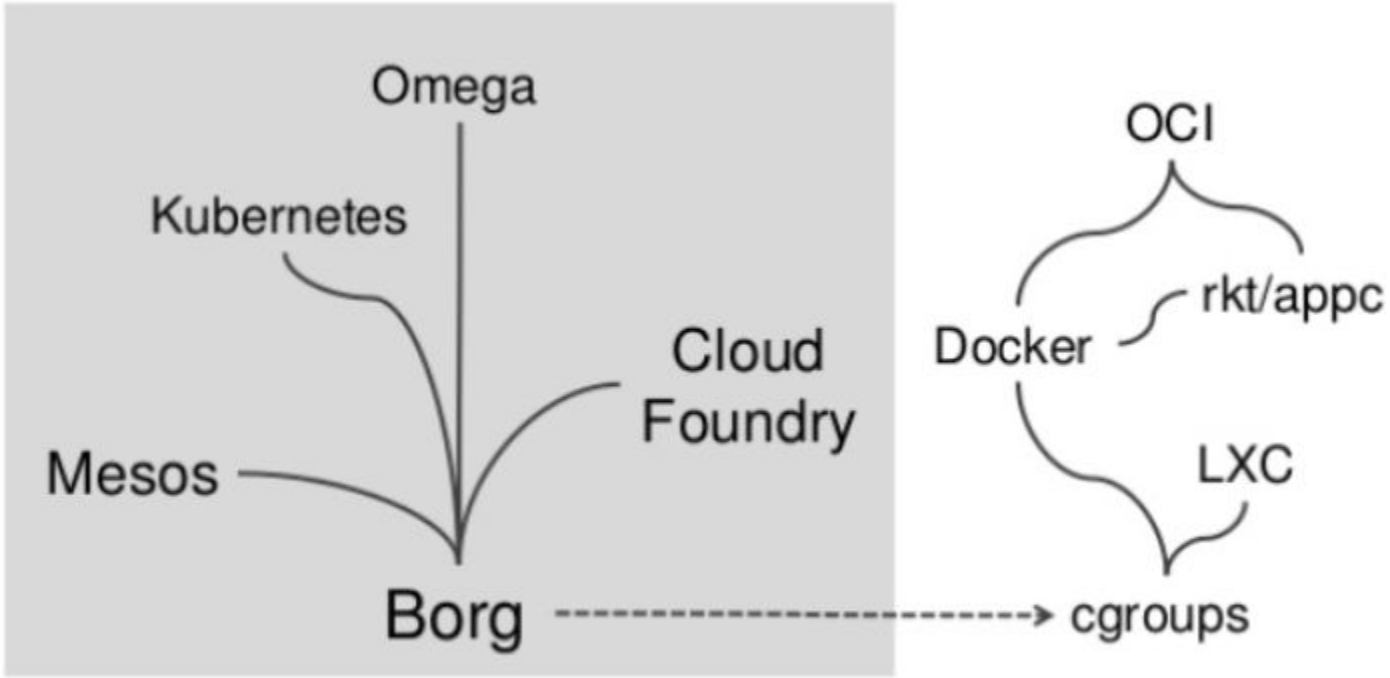
To learn more about the ideas behind Kubernetes, you can read the *Large-scale cluster management at Google with Borg paper*.

Borg has inspired current data center systems, as well as the underlying technologies used in container runtime today. **Google** contributed **cgroups** to the Linux kernel in 2007; it limits the resources used by collection of processes. Both **cgroups** and **Linux namespaces** are at the heart of containers today, including **Docker**.

Mesos was inspired by discussions with **Google** when **Borg** was still a secret. Indeed, **Mesos** builds a multi-level scheduler, which aims to better use a data center cluster.

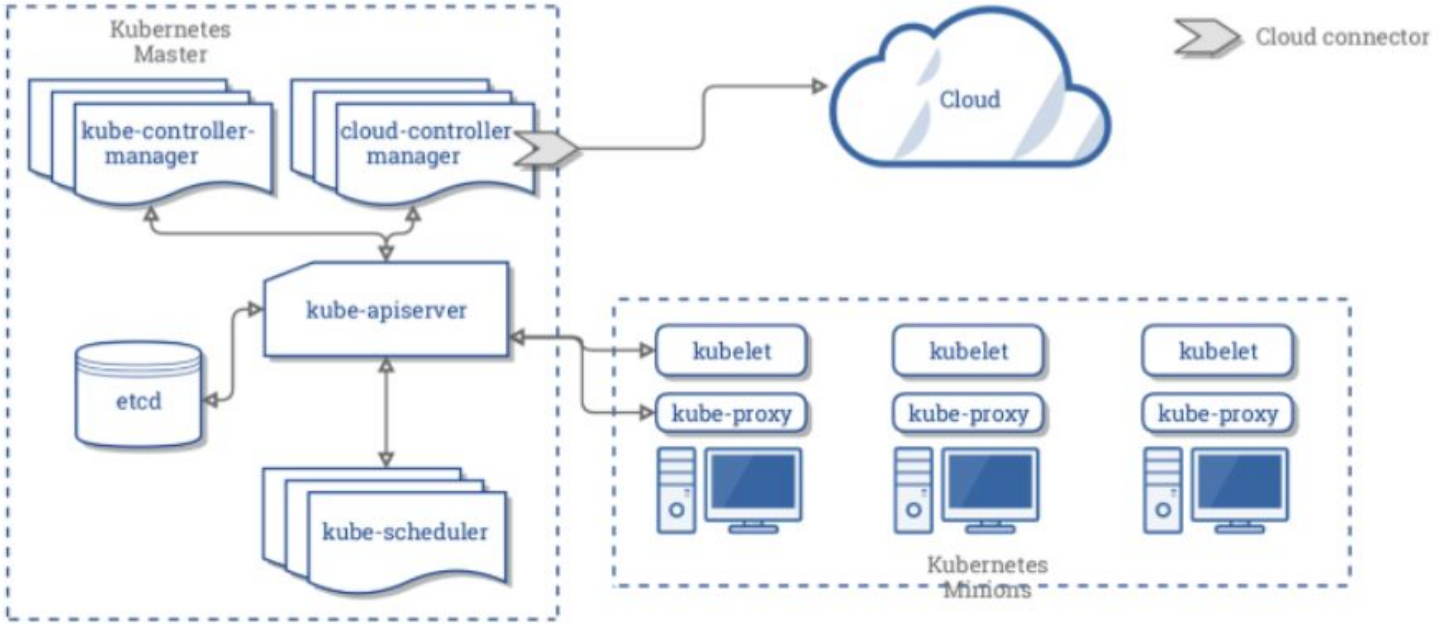
The Cloud Foundry Foundation embraces the 12 factor application principles. These principles provide great guidance to build web applications that can scale easily, can be deployed in the cloud, and whose build is automated. **Borg** and **Kubernetes** address these principles as well.

The Borg Heritage



K8S architecture

To quickly demistify **Kubernetes**, let's have a look at the *Kubernetes Architecture* graphic, which shows a high-level architecture diagram of the system components. Not all components are shown. Every node running a container would have **kubelet** and **kube-proxy**, for example.



K8S architecture

In its simplest form, **Kubernetes** is made of a central manager (aka master) and some worker nodes, once called minions (we will see in a follow-on chapter how you can actually run everything on a single node for testing purposes). The manager runs an API server, a scheduler, various controllers and a storage system to keep the state of the cluster, container settings, and the networking configuration.

Kubernetes exposes an API via the API server. You can communicate with the API using a local client called **kubectl** or you can write your own client and use **curl** commands. The **kube-scheduler** is forwarded the requests for running containers coming to the API and finds a suitable node to run that container. Each node in the cluster runs two processes: a **kubelet** and **kube-proxy**. The **kubelet** receives requests to run the containers, manages any necessary resources and watches over them on the local node. **kubelet** interacts with the local container engine, which is Docker by default, but could be rkt or cri-o, which is growing in popularity.

The **kube-proxy** creates and manages networking rules to expose the container on the network.

Using an API-based communication scheme allows for non-Linux worker nodes and containers. Support for Windows Server 2019 was graduated to *Stable* with the 1.14 release. Only Linux nodes can be master on a cluster.

Terminology

We have learned that Kubernetes is an orchestration system to deploy and manage containers. Containers are not managed individually; instead, they are part of a larger object called a **Pod**. A **Pod** consists of one or more containers which share an IP address, access to storage and namespace. Typically, one container in a Pod runs an application, while other containers support the primary application.

Orchestration is managed through a series of watch-loops, or **controllers**. Each controller interrogates the **kube-apiserver** for a particular object state, modifying the object until the declared state matches the current state. These controllers are compiled into the **kube-controller-manager**. The default, newest, and feature-filled controller for containers is a **Deployment**. A **Deployment** ensures that resources declared in the PodSpec are available, such as IP address and storage, and then deploys a **ReplicaSet**. The **ReplicaSet** is a controller which deploys and restarts pods, which declared to the container engine, Docker by default, to spawn or terminate a container until the requested number is running. Previously, the function was handled by the **ReplicationController**, but has been obviated by **Deployments**. There are also **Jobs** and **CronJobs** to handle single or recurring tasks, among others.

To easily manage thousands of Pods across hundreds of nodes can be a difficult task to manage. To make management easier, we can use **labels**, arbitrary strings which become part of the object metadata. These can then be used when checking or changing the state of objects without having to know individual names or UIDs. Nodes can have **taints** to discourage Pod assignments, unless the Pod has a **toleration** in its metadata.

There is also space in metadata for **annotations** which remain with the object but cannot be used by Kubernetes commands. This information could be used by third-party agents or other tools.

The CNCF

Kubernetes is an open source software with an Apache license. Google donated **Kubernetes** to a newly formed collaborative project within **The Linux Foundation** in July 2015, when **Kubernetes** reached the v1.0 release. This project is known as the **Cloud Native Computing Foundation** (**CNCF**).

CNCF is not just about **Kubernetes**, it serves as the governing body for open source software that solves specific issues faced by cloud native applications (i.e. applications that are written specifically for a cloud environment).

CNCF has many corporate members that collaborate, such as Cisco, the Cloud Foundry Foundation, AT&T, Box, Goldman Sachs, and many others.

Note: Since CNCF now owns the **Kubernetes** copyright, contributors to the source need to sign a contributor license agreement (CLA) with CNCF, just like any contributor to an Apache-licensed project signs a CLA with the **Apache Software Foundation**.

Installation and configuration

Installation tools

This chapter is about Kubernetes installation and configuration. We are going to review a few installation mechanisms that you can use to create your own Kubernetes cluster.

To get started without having to dive right away into installing and configuring a cluster, there are two main choices.

One way is to use **Google Container Engine** (**GKE**), a cloud service from the **Google Cloud Platform**, that lets you request a Kubernetes cluster with the latest stable version.

Another easy way to get started is to use **Minikube**. It is a single binary which deploys into **Oracle VirtualBox** software, which can run in several operating systems. While Minikube is local and single node, it will give you a learning, testing, and development platform.

In both cases, to be able to use the Kubernetes cluster, you will need to have installed the Kubernetes command line, called **kubectl**. This runs locally on your machine and targets the API server endpoint. It allows you to create, manage, and delete all Kubernetes resources (e.g. Pods, Deployments, Services). It is a powerful CLI that we will use throughout the rest of this course. So, you should become familiar with it.

We will use **kubeadm**, which is a newer tool coming up in the Kubernetes project, that makes installing Kubernetes easy and avoids vendor-specific installers. Getting a cluster running involves two commands: `kubeadm init`, that you run on a Master, and then, `kubeadm join`, that you run on your Worker Nodes, and your cluster bootstraps itself. The flexibility of these tools allows Kubernetes to be deployed in a number of places. We will be using this method on AWS nodes.

We will also talk about other installation mechanisms that can be found in the community, such as **kubespray** or **kops**, another way to create a Kubernetes cluster on AWS. We will note you can create your **systemd** unit file in a very traditional way. Additionally, you can use a container image called **hyperkube**, which contains all the key Kubernetes binaries, so that you can run a Kubernetes cluster by just starting a few containers on your nodes.

Installing Kubectl

To configure and manage your cluster, you will probably use the **kubectl** command. You can use **RESTful** calls or the **Go** language, as well.

Enterprise Linux distributions have the various Kubernetes utilities and other files available in their repositories. For example, on RHEL 7/CentOS 7, you would find **kubectl** in the **kubernetes-client** package.

You can (if needed) download the code from Github, and go through the usual steps to compile and install **kubectl**.

This command line will use `~/.kube/config` as a configuration file. This contains all the Kubernetes endpoints that you might use. If you examine it, you will see cluster definitions (i.e. IP endpoints), credentials, and contexts.

A *context* is a combination of a cluster and user credentials. You can pass these parameters on the command line, or switch the shell between contexts with a command, as in:

```
$ kubectl config use-context foobar
```

This is handy when going from a local environment to a cluster in the cloud, or from one cluster to another, such as from development to production.

Using Google Kubernetes Engine (GKE)

Google takes every **Kubernetes** release through rigorous testing and makes it available via its **GKE** service. To be able to use **GKE**, you will need the following:

- An account on Google Cloud.
- A method of payment for the services you will use.
- The **gcloud** command line client.

There is an extensive documentation to get it installed. Pick your favorite method of installation and set it up. For more details, you can visit the Installing Cloud SDK web page.

You will then be able to follow the GKE quickstart guide and you will be ready to create your first **Kubernetes** cluster:

```
$ gcloud container clusters create linuxfoundation
$ gcloud container clusters list
$ kubectl get nodes
```

By installing `gcloud`, you will have automatically installed **kubectl**. In the commands above, we created the cluster, listed it, and then, listed the nodes of the cluster with **kubectl**.

Once you are done, **do not forget to delete your cluster**, otherwise you will keep on getting charged for it:

```
$ gcloud container clusters delete linuxfoundation
```

Using Minikube

You can also use Minikube, an open source project within the GitHub Kubernetes organization. While you can download a release from **GitHub**, following listed directions, it may be easier to download a pre-compiled binary. Make sure to verify and get the latest version.

For example, to get the v.0.22.2 version, do:

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

With Minikube now installed, starting Kubernetes on your local machine is very easy:

```
$ minikube start
$ kubectl get nodes
```

This will start a VirtualBox virtual machine that will contain a single node Kubernetes deployment and the Docker engine. Internally, minikube runs a single Go binary called `localkube`. This binary runs all the components of Kubernetes together. This makes Minikube simpler than a full Kubernetes deployment. In addition, the Minikube VM also runs Docker, in order to be able to run containers.

Installing With Kubeadm

Once you become familiar with Kubernetes using Minikube, you may want to start building a real cluster. Currently, the most straightforward method is to use **kubeadm**, which appeared in Kubernetes v1.4.0, and can be used to bootstrap a cluster quickly.

The Kubernetes website provides documentation on how to use kubeadm to create a cluster.

Package repositories are available for **Ubuntu 16.04** and **CentOS 7.1**.

Run `kubeadm init` on the head node. The token is returned by the command, `kubeadm init`. Create a network for IP-per-Pod criteria. Run `kubeadm join --token token head-node-IP` on the worker nodes. You can also create the network with **kubectl**, by using a resource manifest of the network.

For example, to use the **Weave** network, you should do the following:

```
$ kubectl create -f https://git.io/weave-kube
```

Once all the steps are completed, you will have a functional multi-node Kubernetes cluster, and you will be able to use **kubectl** to interact with it.

Installing a Pod Network

Prior to initializing the Kubernetes cluster, the network must be considered and IP conflicts avoided. There are several Pod networking choices, in varying levels of development and feature set:

- Calico
  A flat Layer 3 network which communicates without IP encapsulation, used in production with software such as **Kubernetes**, **OpenShift**, **Docker**, **Mesos** and **OpenStack**. Viewed as a simple and flexible networking model, it scales well for large environments. Another network option, **Canal**, also part of this project, allows for integration with **Flannel**. Allows for implementation of network policies.
- Flannel
  A Layer 3 IPv4 network between the nodes of a cluster. Developed by **CoreOS**, it has a long history with **Kubernetes**. Focused on traffic between hosts, not how containers configure local networking, it can use one of several backend mechanisms, such as VXLAN. A **flanneld** agent on each node allocates subnet leases for the host. While it can be configured after deployment, it is much easier prior to any Pods being added.
- Kube-router
  Feature-filled single binary which claims to "*do it all*". The project is in the alpha stage, but promises to offer a distributed load balancer, firewall, and router purposely built for **Kubernetes**.
- Romana
  Another project aimed at network and security automation for cloud native applications. Aimed at large clusters, IPAM-aware topology and integration with **kops** clusters.
- Weave Net
  Typically used as an add-on for a CNI-enabled **Kubernetes** cluster.

Many of the projects will mention the Container Network Interface (CNI), which is a CNCF project. Several container runtimes currently use CNI. As a standard to handle deployment management and cleanup of network resources, it will become more popular.

Installation Consideration

To begin the installation process, you should start experimenting with a single-node deployment. This single-node will run all the **Kubernetes** components (e.g. API server, controller, scheduler, kubelet, and kube-proxy). You can do this with **Minikube** for example.

Once you want to deploy on a cluster of servers (physical or virtual), you will have many choices to make, just like with any other distributed system:

- Which provider should I use? A public or private cloud? Physical or virtual?
- Which operating system should I use? **Kubernetes** runs on most operating systems (e.g. **Debian**, **Ubuntu**, **CentOS**, etc.), plus on container-optimized OSes (e.g. **CoreOS**, **Atomic**).
- Which networking solution should I use? Do I need an overlay?
- Where should I run my **etcd** cluster?
- Can I configure Highly Available (HA) head nodes?

To learn more about how to choose the best options, you can read the *Picking the Right Solution* article.

With **systemd** becoming the dominant **init** system on **Linux**, your **Kubernetes** components will end up being run as *systemd unit files* in most cases. Or, they will be run via a kubelet running on the head node (i.e. `kubadm`).

Main Deployment Configurations

At a high level, you have four main deployment configurations:

- Single-node
- Single head node, multiple workers
- Multiple head nodes with HA, multiple workers
- HA **etcd**, HA head nodes, multiple workers.

Which of the four you will use will depend on how advanced you are in your **Kubernetes** journey, but also on what your goals are.

With a single-node deployment, all the components run on the same server. This is great for testing, learning, and developing around Kubernetes.

Adding more workers, a single head node and multiple workers typically will consist of a single node **etcd** instance running on the head node with the API, the scheduler, and the controller-manager.

Multiple head nodes in an HA configuration and multiple workers add more durability to the cluster. The API server will be fronted by a load balancer, the scheduler and the controller-manager will elect a leader (which is configured via flags). The **etcd** setup can still be single node.

The most advanced and resilient setup would be an HA **etcd** cluster, with HA head nodes and multiple workers. Also, **etcd** would run as a true cluster, which would provide HA and would run on nodes separate from the Kubernetes head nodes.

The use of Kubernetes Federations also offers high availability. Multiple clusters are joined together with a common control plane allowing movement of resources from one cluster to another administratively or after failure.

Systemd Unit File for Kubernetes

In any of these configurations, you will run some of the components as a standard system daemon. As an example, below is a sample **systemd** unit file to run the `controller-manager`:

```
 - name: kube-controller-manager.service
    command: start
    content: |
      [Unit]
      Description=Kubernetes Controller Manager
Documentation=https://github.com/kubernetes/kubernetes
      Requires=kube-apiserver.service
      After=kube-apiserver.service
      [Service]
      ExecStartPre=/usr/bin/curl -L -o /opt/bin/kube-controller-manager -z /opt/bin/kube-controller-manager
https://storage.googleapis.com/kubernetes-release/release/v1.7.6/bin/linux/amd64/kube-controller-manager
      ExecStartPre=/usr/bin/chmod +x /opt/bin/kube-controller-manager
      ExecStart=/opt/bin/kube-controller-manager \
      --service-account-private-key-file=/opt/bin/kube-serviceaccount.key \
      --root-ca-file=/var/run/kubernetes/apiserver.crt \
      --master=127.0.0.1:8080 \
      ...
```

Kubernetes Architecture

Main Components

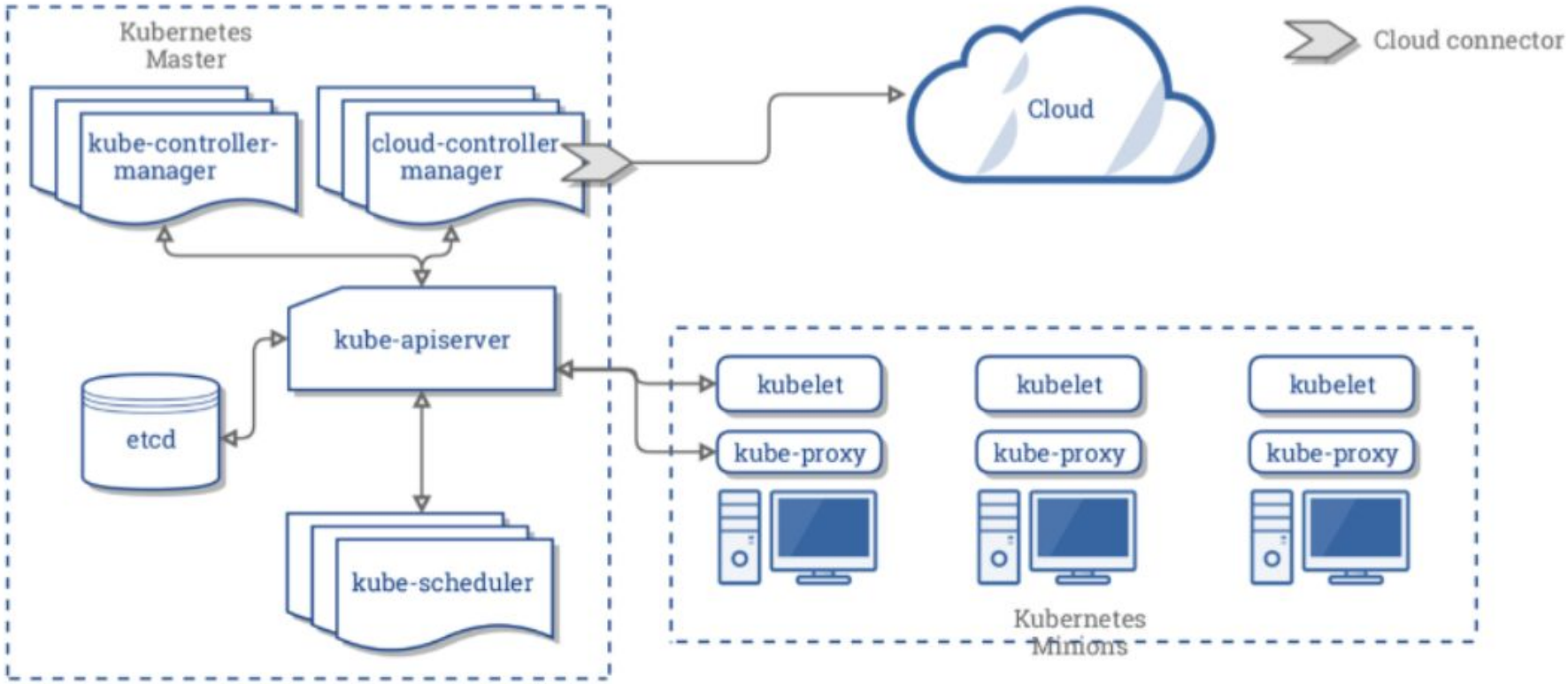
Kubernetes has the following main components:

- Master and worker nodes
- Controllers
- Services
- Pods of containers
- Namespaces and quotas
- Network and policies
- Storage.

A **Kubernetes** cluster is made of a master node and a set of worker nodes. The cluster is all driven via API calls to controllers, both interior as well as exterior traffic. We will take a closer look at these components next.

Most of the processes are executed inside a container. There are some differences, depending on the vendor and the tool used to build the cluster.

Main Components



Main Components

Master Node

The Kubernetes master runs various server and manager processes for the cluster. Among the components of the master node are the **kube-apiserver**, the **kube-scheduler**, and the **etcd** database. As the software has matured, new components have been created to handle dedicated needs, such as the **cloud-controller-manager**; it handles tasks once handled by the **kube-controller-manager** to interact with other tools, such as **Rancher** or **DigitalOcean** for third-party cluster management and reporting.

There are several add-ons which have become essential to a typical production cluster, such as DNS services. Others are third-party solutions where Kubernetes has not yet developed a local component, such as cluster-level logging and resource monitoring.

kube-apiserver

The **kube-apiserver** is central to the operation of the Kubernetes cluster.

All calls, both internal and external traffic, are handled via this agent. All actions are accepted and validated by this agent, and it is the only connection to the **etcd** database. As a result, it acts as a master process for the entire cluster, and acts as a frontend of the cluster's shared state.

Main Components

kube-scheduler

The **kube-scheduler** uses an algorithm to determine which node will host a Pod of containers. The scheduler will try to view available resources (such as volumes) to bind, and then try and retry to deploy the Pod based on availability and success.

There are several ways you can affect the algorithm, or a custom scheduler could be used instead. You can also bind a Pod to a particular node, though the Pod may remain in a pending state due to other settings.

One of the first settings referenced is if the Pod can be deployed within the current quota restrictions. If so, then the taints and tolerations, and labels of the Pods are used along with those of the nodes to determine the proper placement.

etcd Database

The state of the cluster, networking, and other persistent information is kept in an **etcd** database, or, more accurately, a *b+tree* key-value store. Rather than finding and changing an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process. It works with **curl** and other HTTP libraries, and provides reliable watch queries.

Simultaneous requests to update a value all travel via the **kube-apiserver**, which then passes along the request to **etcd** in a series. The first request would update the database. The second request would no longer have the same version number, in which case the **kube-apiserver** would reply with an error 409 to the requester. There is no logic past that response on the server side, meaning the client needs to expect this and act upon the denial to update.

There is a `master` database along with possible `followers`. They communicate with each other on an ongoing basis to determine which will be **master**, and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with new tools, such as **kubeadm**, and features like whole cluster upgrades.

Main Components

Worker Nodes

All worker nodes run the **kubelet** and **kube-proxy**, as well as the container engine, such as Docker or rkt. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.

The **kubelet** interacts with the underlying Docker Engine also installed on all the nodes, and makes sure that the containers that need to run are actually running. The **kube-proxy** is in charge of managing the network connectivity to the containers. It does so through the use of **iptables** entries. It also has the `userspace` mode, in which it monitors *Services* and *Endpoints* using a random port to proxy traffic and an alpha feature of `ipvs`.

You can also run an alternative to the Docker engine: cri-o or rkt. To learn how you can do that, you should check the documentation. In future releases, it is highly likely that Kubernetes will support additional container runtime engines.

Supervisord is a lightweight process monitor used in traditional Linux environments to monitor and notify about other processes. In the cluster, this daemon monitors both the **kubelet** and **docker** processes. It will try to restart them if they fail, and log events. While not part of a standard installation, some may add this monitor for added reporting.

Kubernetes does not have cluster-wide logging yet. Instead, another CNCF project is used, called Fluentd. When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages.

Main Components

kubelet

The **kubelet** agent is the heavy lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications (a `PodSpec` is a JSON or YAML file that describes a pod). It will work to configure the local node until the specification has been met.

Should a Pod require access to storage, *Secrets* or *ConfigMaps*, the **kubelet** will ensure access or creation. It also sends back status to the **kube-apiserver** for eventual persistence.

- Uses `PodSpec`
- Mounts volumes to Pod
- Downloads secrets
- Passes request to local container engine
- Reports status of Pods and node to cluster.

Main Components

Services

With every object and agent decoupled we need a flexible and scalable agent which connects resources together and will reconnect, should something die and a replacement is spawned. Each *Service* is a microservice handling a particular bit of traffic, such as a single `NodePort` or a `LoadBalancer` to distribute inbound requests among many Pods.

A *Service* also handles access policies for inbound requests, useful for resource control, as well as for security.

- Connect Pods together
- Expose Pods to Internet
- Decouple settings
- Define Pod access policy.

Main Components

Controllers

An important concept for orchestration is the use of controllers. Various controllers ship with Kubernetes, and you can create your own, as well. A simplified view of a controller is an agent, or `Informer`, and a downstream store. Using a `DeltaFIFO` queue, the source and downstream are compared. A loop process receives an `obj` or object, which is an array of deltas from the FIFO queue. As long as the delta is not of the type `Deleted`, the logic of the controller is used to create or modify some object until it matches the specification.

The `Informer` which uses the API server as a source requests the state of an object via an API call. The data is cached to minimize API server transactions. A similar agent is the `SharedInformer`; objects are often used by multiple other objects. It creates a shared cache of the state for multiple requests.

A `Workqueue` uses a key to hand out tasks to various workers. The standard **Go** work queues of rate limiting, delayed, and time queue are typically used.

The `endpoints`, `namespace`, and `serviceaccounts` controllers each manage the eponymous resources for Pods.

Main Components

Pods

The whole point of Kubernetes is to orchestrate the lifecycle of a container. We do not interact with particular containers. Instead, the smallest unit we can work with is a *Pod*. Some would say a pod of whales or peas-in-a-pod. Due to shared resources, the design of a *Pod* typically follows a one-process-per-container architecture.

Containers in a *Pod* are started in parallel. As a result, there is no way to determine which container becomes available first inside a pod. To support a single process running in a container, you may need logging, a proxy, or special adapter. These tasks are often handled by other containers in the same pod.

There is only one IP address per Pod. If there is more than one container, they must share the IP. To communicate with each other, they can either use IPC, or a shared filesystem.

While Pods are often deployed with one application container in each, a common reason to have multiple containers in a Pod is for logging. You may find the term `sidecar` for a container dedicated to performing a helper task, like handling logs and responding to requests, as the primary application container may have this ability.

Main Components

Containers

While Kubernetes orchestration does not allow direct manipulation on a container level, we can manage the resources containers are allowed to consume.

In the `resources` section of the `PodSpec` you can pass parameters which will be passed to the container runtime on the scheduled node:

```
resources:
  limits:
    cpu: "1"
    memory: "4Gi"
  requests:
    cpu: "0.5"
    memory: "500Mi"
```

Another way to manage resource usage of the containers is by creating a `ResourceQuota` object, which allows hard and soft limits to be set in a namespace. The quotas allow management of more resources than just CPU and memory and allows limiting several objects.

Main Components

Node

A node is an API object created outside the cluster representing an instance. While a master must be Linux, worker nodes can also be Microsoft Windows Server 2019. Once the node has the necessary software installed, it is ingested into the API server.

At the moment, you can create a master node with `kubeadm init` and worker nodes by passing `join`. In the near future, secondary master nodes and/or etcd nodes may be joined.

If the **kube-apiserver** cannot communicate with the kubelet on a node for 5 minutes, the default *NodeLease* will schedule the node for deletion and the *NodeStatus* will change from `ready`. The pods will be evicted once a connection is re-established. They are no longer forcibly removed and rescheduled by the cluster.

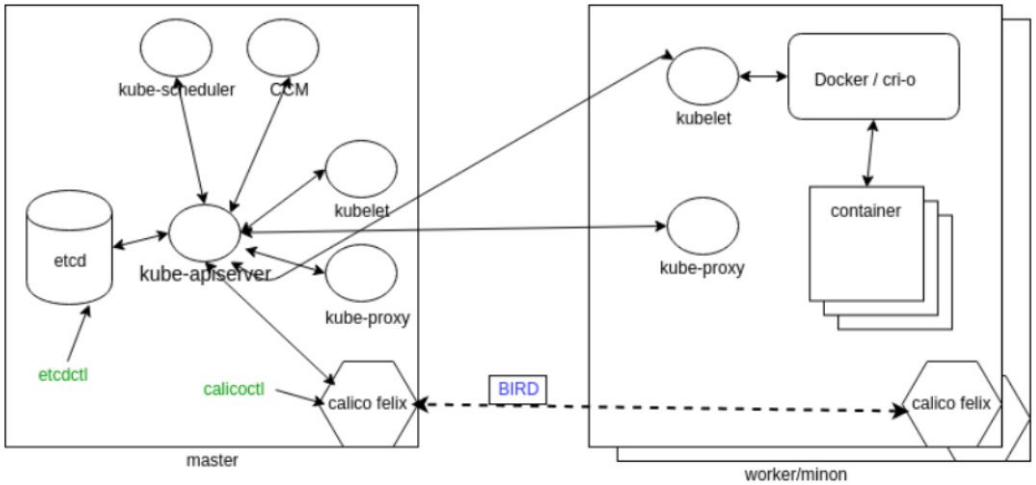
Each node object exists in the `kube-node-lease` namespace. To remove a node from the cluster, first use `kubectl delete node <node-name>` to remove it from the API server. This will cause pods to be evacuated. Then, use `kubeadm reset` to remove cluster-specific information. You may also need to remove iptables information, depending on if you plan on re-using the node.

Components Review

Now that we have seen some of the components, lets take another look with some of the connections shown. Not all connections are shown in this diagram. Note that all of the components are communicating with **kube-apiserver**. Only **kube-apiserver** communicates with the etcd database.

We also see some commands, which we may need to install separately to work with various components. There is an **etcdctl** command to interrogate the database and **calicoctl** to view more of how the network is configured. We can see Felix, which is the primary Calico agent on each machine. This agent, or daemon, is responsible for interface monitoring and management, route programming, ACL configuration and state reporting.

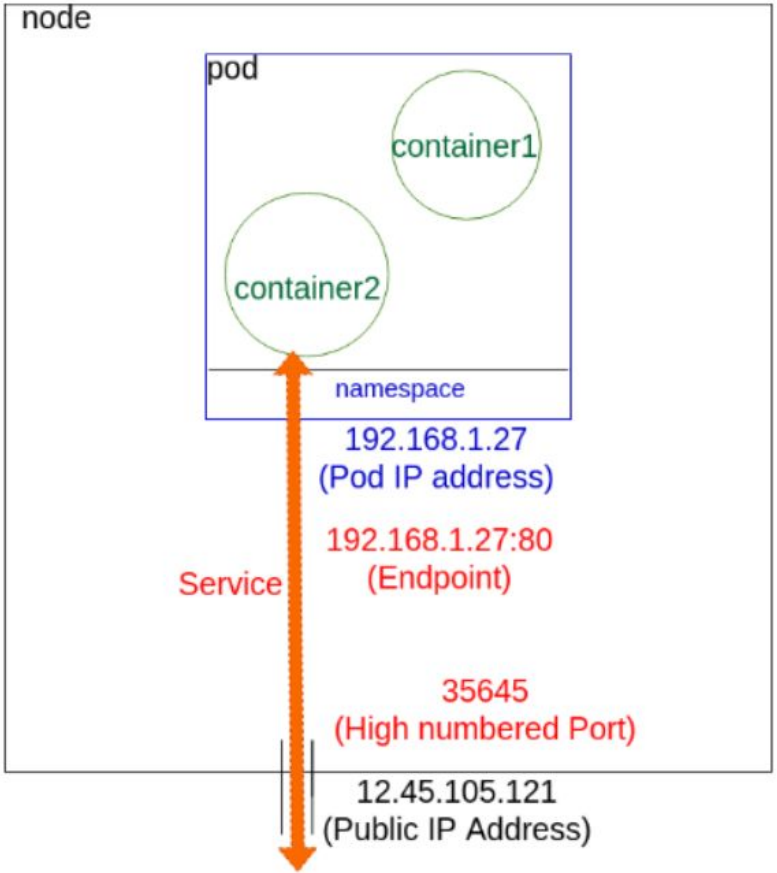
BIRD is a dynamic IP routing daemon used by Felix to read routing state and distribute that information to other nodes in the cluster. This allows a client to connect to any node, and eventually be connected to the workload on a container, even if not the node originally contacted.



Container To Outside Path

This graphic shows a node with a single, dual-container pod. A NodePort service connects the Pod to the outside network. Even though there are two containers, they share the same namespace and the same IP address, which would be configured by kubectl working with kube-proxy.

The end point is created at the same time as the service. Note that it uses the pod IP address, but also includes a port. The service connects network traffic from a node high-number port to the endpoint using iptables with ipvs on the way. The **kube-controller-manager** handles the watch loops to monitor the need for endpoints and services, as well as any updates or deletions.



Networking Setup

Getting all the previous components running is a common task for system administrators who are accustomed to configuration management. But, to get a fully functional **Kubernetes** cluster, the network will need to be set up properly, as well.

A detailed explanation about the **Kubernetes** networking model can be seen on the Cluster Networking page in the Kubernetes documentation.

If you have experience deploying virtual machines (VMs) based on IaaS solutions, this will sound familiar. The only caveat is that, in **Kubernetes**, the lowest compute unit is not a **container**, but what we call a **pod**.

A pod is a group of co-located containers that share the same IP address. From a networking perspective, a pod can be seen as a virtual machine of physical hosts. The network needs to assign IP addresses to pods, and needs to provide traffic routes between all pods on any nodes.

The three main networking challenges to solve in a container orchestration system are:

- Coupled container-to-container communications (solved by the pod concept).
- Pod-to-pod communications.
- External-to-pod communications (solved by the services concept, which we will discuss later).

Kubernetes expects the network configuration to enable pod-to-pod communications to be available; it will not do it for you.

CNI Network Configuration File

To provide container networking, **Kubernetes** is standardizing on the Container Network Interface (CNI) specification. Since v1.6.0, **kubeadm's** (the **Kubernetes** cluster bootstrapping tool) goal has been to use CNI, but you may need to recompile to do so.

CNI is an emerging specification with associated libraries to write plugins that configure container networking and remove allocated resources when the container is deleted. Its aim is to provide a common interface between the various networking solutions and container runtimes. As the CNI specification is language-agnostic, there are many plugins from Amazon ECS, to SR-IOV, to Cloud Foundry, and more.

With CNI, you can write a network configuration file:

```
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
        ]
    }
}
```

This configuration defines a standard Linux bridge named `cni0`, which will give out IP addresses in the subnet **`10.22.0.0./16`**. The `bridge` plugin will configure the network interfaces in the correct namespaces to define the container network properly.

Pod To Pod Communication

While a CNI plugin can be used to configure the network of a pod and provide a single IP per pod, CNI does not help you with pod-to-pod communication across nodes.

The requirement from Kubernetes is the following:

- All pods can communicate with each other across nodes.
- All nodes can communicate with all pods.
- No Network Address Translation (NAT).

Basically, all IPs involved (nodes and pods) are routable without NAT. This can be achieved at the physical network infrastructure if you have access to it (e.g. GKE). Or, this can be achieved with a software defined overlay with solutions like:

- Weave
- Flannel
- Calico
- Romana.

APIS

API Access

Kubernetes has a powerful **REST**-based API. The entire architecture is API-driven. Knowing where to find resource endpoints and understanding how the API changes between versions can be important to ongoing administrative tasks, as there is much ongoing change and growth. Currently, there is no process for deprecation.

As we learned in the *Architecture* chapter, the main agent for communication between cluster agents and from outside the cluster is the **kube-apiserve**r. A `curl` query to the agent will expose the current API groups. Groups may have multiple versions, which evolve independently of other groups, and follow a domain-name format with several names reserved, such as single-word domains, the empty group, and any name ending in `.k8s.io`.

Such an API group is seen here:

```
$ curl https://127.0.0.1:6443/apis -k
....
    {
      "name": "apps",
      "versions": [
        {
          "groupVersion": "apps/v1beta1",
          "version": "v1beta1"
        },
        {
          "groupVersion": "apps/v1beta2",
          "version": "v1beta2"
        }
      ],
     },
....
```

Restful

kubectl makes API calls on your behalf, responding to typical `HTTP` verbs (`GET`, `POST`, `DELETE`). You can also make calls externally, using **curl** or other program. With the appropriate certificates and keys, you can make requests, or pass JSON files to make configuration changes.

```
$ curl --cert userbob.pem --key userBob-key.pem \
  --cacert /path/to/ca.pem \
  https://k8sServer:6443/api/v1/pods
```

The ability to impersonate other users or groups, subject to RBAC configuration, allows a manual override authentication. This can be helpful for debugging authorization policies of other users.

Checking Access

While there is more detail on security in a later chapter, it is helpful to check the current authorizations, both as an administrator, as well as another user. The following shows what user `bob` could do in the `default` namespace and the `developer` namespace, using the `auth can-i` subcommand to query:

```
$ kubectl auth can-i create deployments
yes

$ kubectl auth can-i create deployments --as bob
no

$ kubectl auth can-i create deployments --as bob --namespace developer
yes
```

There are currently three APIs which can be applied to set who and what can be queried:

- `SelfSubjectAccessReview`
  Access review for any user, helpful for delegating to others.
- `LocalSubjectAccessReview`
  Review is restricted to a specific namespace.
- `SelfSubjectRulesReview`
  A review which shows allowed actions for a user within a particular namespace.

The use of `reconcile` allows a check of authorization necessary to create an object from a file. No output indicates the creation would be allowed.

Using Annotations

Labels are used to work with objects or collections of objects; *annotations* are not.

Instead, *annotations* allow for metadata to be included with an object that may be helpful outside of the Kubernetes object interaction. Similar to *labels*, they are key to value maps. They are also able to hold more information, and more human-readable information than *labels*.

Having this kind of metadata can be used to track information such as a timestamp, pointers to related objects from other ecosystems, or even an email from the developer responsible for that object's creation.

The *annotation* data could otherwise be held in an exterior database, but that would limit the flexibility of the data. The more this metadata is included, the easier it is to integrate management and deployment tools or shared client libraries.

For example, to annotate only Pods within a namespace, you can overwrite the annotation, and finally delete it:

```
$ kubectl annotate pods --all description='Production Pods' -n prod
$ kubectl annotate --overwrite pods description="Old Production Pods" -n prod
$ kubectl annotate pods foo description- -n prod
```

Simple Pod

As discussed earlier, a Pod is the lowest compute unit and individual object we can work with in Kubernetes. It can be a single container, but often, it will consist of a primary application container and one or more supporting containers.

Below is an example of a simple pod manifest in YAML format. You can see the **`apiVersion`** (it must match the existing API group), the **`kind`** (the type of object to create), the **`metadata`** (at least a name), and its **`spec`** (what to create and parameters), which define the container that actually runs in this pod:

```
apiVersion: v1
kind: Pod
metadata:
    name: firstpod
spec:
    containers:
    - image: nginx
      name: stan
```

You can use the **`kubectl create`** command to create this pod in Kubernetes. Once it is created, you can check its status with **`kubectl get pods`**. The output is omitted to save space:

```
$ kubectl create -f simple.yaml

$ kubectl get pods

$ kubectl get pod firstpod -o yaml

$ kubectl get pod firstpod -o json
```

Managing API Resources with Kubectl

Kubernetes exposes resources via RESTful API calls, which allows all resources to be managed via HTTP, JSON or even XML, the typical protocol being HTTP. The state of the resources can be changed using standard HTTP verbs (e.g. `GET`, `POST`, `PATCH`, `DELETE`, etc.).

kubectl has a verbose mode argument which shows details from where the command gets and updates information. Other output includes **curl** commands you could use to obtain the same result. While the verbosity accepts levels from zero to any number, there is currently no verbosity value greater than nine. You can check this out for `kubectl get`. The output below has been formatted for clarity:

```
$ kubectl --v=10 get pods firstpod
....
I1215 17:46:47.860958 29909 round_trippers.go:417]
   curl -k -v -XGET -H "Accept: application/json"
   -H "User-Agent: kubectl/v1.8.5 (linux/amd64) kubernetes/cce11c6"
   https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod
....
```

If you delete this pod, you will see that the HTTP method changes from `XGET` to `XDELETE`:

```
$ kubectl --v=9 delete pods firstpod
....
I1215 17:49:32.166115 30452 round_trippers.go:417]
   curl -k -v -XDELETE -H "Accept: application/json, */*"
   -H "User-Agent: kubectl/v1.8.5 (linux/amd64) kubernetes/cce11c6"
   https://10.128.0.3:6443/api/v1/namespaces/default/pods/firstpod
....
```

Access From Outside The Cluster

The primary tool used from the command line will be **kubectl**, which calls **curl** on your behalf. You can also use the **curl** command from outside the cluster to view or make changes.

The basic server information, with redacted TLS certificate information, can be found in the output of

```
$ kubectl config view
```

If you view the verbose output from a previous page, you will note that the first line references a configuration file where this information is pulled from, `~/.kube/config`:

```
I1215 17:35:46.725407 27695 loader.go:357]
     Config loaded from file /home/student/.kube/config
```

Without the certificate authority, key and certificate from this file, only insecure **curl** commands can be used, which will not expose much due to security settings. We will use **curl** to access our cluster using TLS in an upcoming lab.

~/.kube/config

Take a look at the output below:

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0tLS1CRUdF.....
    server: https://10.128.0.3:6443
  name: kubernetes
contexts:
- context:
    cluster: kubernetes
    user: kubernetes-admin
  name: kubernetes-admin@kubernetes
current-context: kubernetes-admin@kubernetes
kind: Config
preferences: {}
users:
- name: kubernetes-admin
  user:
    client-certificate-data: LS0tLS1CRUdJTib.....
    client-key-data: LS0tLS1CRUdJTi....
```

~/.kube/config

The output on the previous page shows 19 lines of output, with each of the keys being heavily truncated. While the keys may look similar, close examination shows them to be distinct:

- `apiVersion`
  As with other objects, this instructs the `kube-apiserver` where to assign the data.
- `clusters`
  This contains the name of the cluster, as well as where to send the API calls. The `certificate-authority-data` is passed to authenticate the **curl** request.
- `contexts`
  A setting which allows easy access to multiple clusters, possibly as various users, from one configuration file. It can be used to set `namespace`, `user`, and `cluster`.
- `current-context`
  Shows which cluster and user the **kubectl** command would use. These settings can also be passed on a per-command basis.
- `kind`
  Every object within Kubernetes must have this setting, in thus case a declaration of object type `Config`.
- `preferences`
  Currently not used, optional settings for the **kubectl** command, such as colorizing output.
- `users`
  A nickname associated with client credentials, which can be client key and certificate, username and password, and a token. Token and username/password are mutually exclusive. These can be configured via the `kubectl config set-credentials` command.

Namespaces

The term **namespace** is used to reference both the kernel feature and the segregation of API objects by Kubernetes. Both are means to keep resources distinct.

Every API call includes a namespace, using `default` if not otherwise declared:
`https://10.128.0.3:6443/api/v1/namespaces/default/pods`.

Namespaces, a Linux kernel feature that segregates system resources, are intended to isolate multiple groups and the resources they have access to work with via quotas. Eventually, access control policies will work on namespace boundaries, as well. One could use *Labels* to group resources for administrative reasons.

There are four namespaces when a cluster is first created.

- `default`
  This is where all resources are assumed, unless set otherwise.
- `kube-node-lease`
  The namespace where worker node lease information is kept.
- `kube-public`
  A namespace readable by all, even those not authenticated. General information is often included in this namespaces.
- `kube-system`
  Contains infrastructure pods.

Should you want to see all the resources on a system, you must pass the `--all-namespaces` option to the `kubectl` command.

Namespaces

Take a look at the following commands:

```
$ kubectl get ns
$ kubectl create ns linuxcon
$ kubectl describe ns linuxcon
$ kubectl get ns/linuxcon -o yaml
$ kubectl delete ns/linuxcon
```

The above commands show how to view, create and delete namespaces. Note that the **`describe`** subcommand shows several settings, such as *Labels*, *Annotations*, *resource quotas*, and *resource limits*, which we will discus later in the course.

Once a namespace has been created, you can reference it via YAML when creating a resource:

```
$ cat redis.yaml
apiVersion: V1
kind: Pod
metadata:
    name: redis
    namespace: linuxcon
...
```

Managing State With Deployments

Overview

The default controller for a container deployed via **`kubectl run`** is a *Deployment*. While we have been working with them already, we will take a closer look at configuration options.

As with other objects, a *deployment* can be made from a YAML or JSON **`spec`** file. When added to the cluster, the controller will create a *ReplicaSet* and a *Pod* automatically. The containers, their settings and applications can be modified via an update, which generates a new *ReplicaSet*, which, in turn, generates new *Pods*.

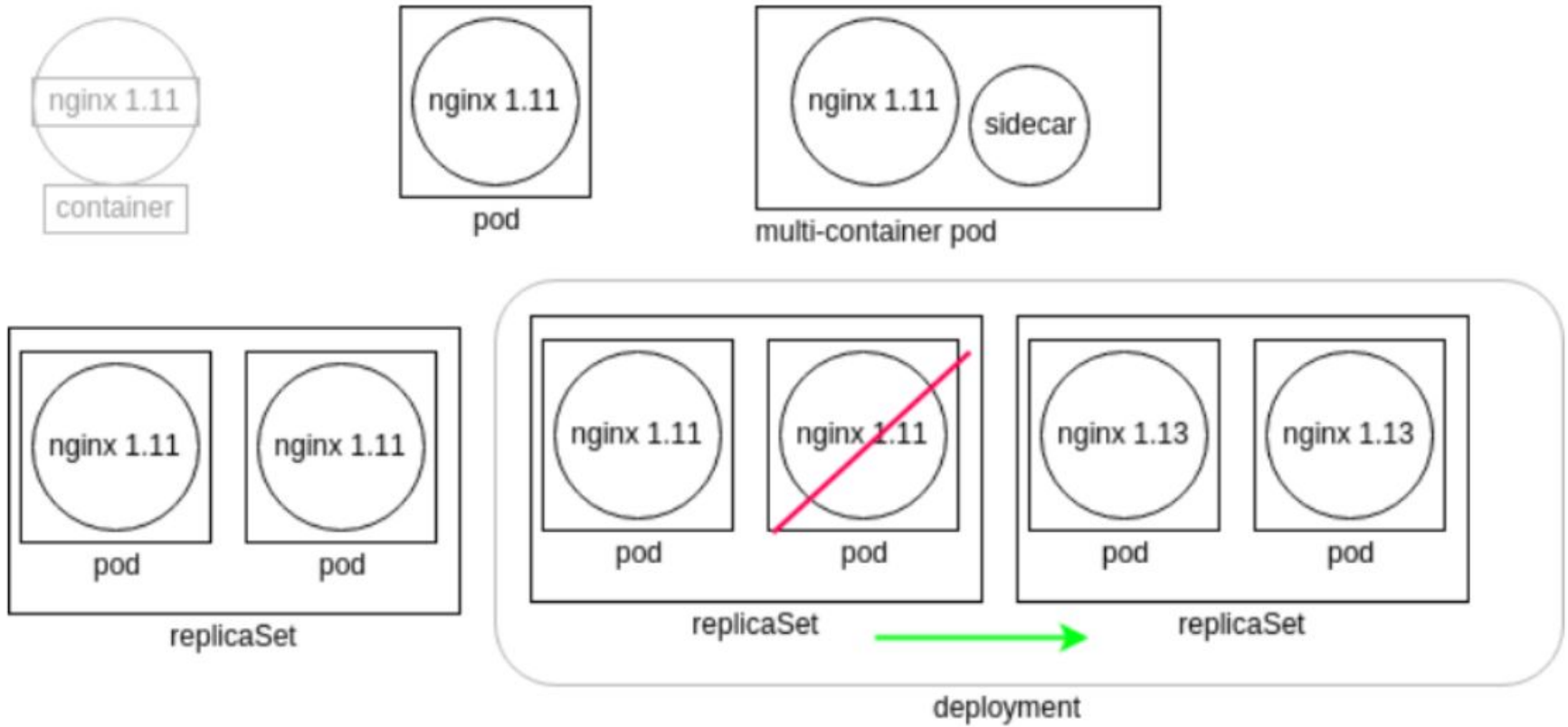
The updated objects can be staged to replace previous objects as a block or as a rolling update, which is determined as part of the deployment specification. Most updates can be configured by editing a YAML file and running **`kubectl apply`**. You can also use **`kubectl edit`** to modify the in-use configuration. Previous versions of the *ReplicaSets* are kept, allowing a rollback to return to a previous configuration.

We will also talk more about ***labels***. Labels are essential to administration in Kubernetes, but are not an API resource. They are user-defined key-value pairs which can be attached to any resource, and are stored in the metadata. Labels are used to query or select resources in your cluster, allowing for flexible and complex management of the cluster.

As a label is arbitrary, you could select all resources used by developers, or belonging to a user, or any attached string, without having to figure out what kind or how many of such resources exist.

Objects Relationships

Here we can see the relationship of objects from the container which Kubernetes does not directly manage, up to the deployment.



Objects Relationships

The boxes and shapes are logical, in that they represent the controllers, or watch loops, running as a thread of **kube-controller-manager**. Each controller queries the **kube-apiserver** for the current state of the object they track. The state of each object on a worker node is sent back from the local **kubelet**.

The graphic in the upper left represents a container running nginx 1.11. Kubernetes does not directly manage the container. Instead, the kubelet daemon checks the pod specifications by asking the container engine, which could be Docker or cri-o, for the current status. The graphic to the right of the container shows a pod which represents a watch loop checking the container status. kublet compares the current pod spec against what the container engine replies and will terminate and restart the pod if necessary.

A multi-container pod is shown next. While there are several names used, such as `sidecar` or `ambassador`, these are all multi-container pods. The names are used to indicate the particular reason to have a second container in the pod, instead of denoting a new kind of pod.

On the lower left we see a *replicaSet*. This controller will ensure you have a certain number of pods running. The pods are all deployed with the same *podspec*, which is why they are called replicas. Should a pod terminate or a new pod be found, the *replicaSet* will create or terminate pods until the current number of running pods matches the specifications. Any of the current pods could be terminated should the spec demand fewer pods running.

The graphic in the lower right shows a *deployment*. This controller allows us to manage the versions of images deployed in the pods. Should an edit be made to the *deployment*, a new *replicaSet* is created, which will deploy pods using the new *podSpec*. The deployment will then direct the old *replicaSet* to shut down pods as the new *replicaSet* pods become available. Once the old pods are all terminated, the deployment terminates the old *replicaSet* and the deployment returns to having only one *replicaSet* running.

Deploments Details

In the previous page, we created a new deployment running a particular version of the **nginx** web server.

To generate the YAML file of the newly created objects, do:

```
$ kubectl get deployments,rs,pods -o yaml
```

Sometimes, a JSON output can make it more clear:

```
$ kubectl get deployments,rs,pods -o json
```

Now, we will look at the YAML output, which also shows default values, not passed to the object when created.

```
apiVersion: v1
items:
- apiVersion: extensions/v1beta1
  kind: Deployment
```

- **apiVersion**
  A value of **v1** shows that this object is considered to be a stable resource. In this case, it is not the *deployment*. It is a reference to the **List** type.
- **items**
  As the previous line is a **List**, this declares the list of items the command is showing.
- **- apiVersion**
  The dash is a YAML indication of the first item, which declares the **apiVersion** of the object as **extensions/v1beta1**. While this would indicate that the object is not considered stable and will be dynamic, deployments are the suggested object to use.
- **kind**
  This is where the type of object to create is declared, in this case, a **deployment**.

Deployment Configuration Metada

Continuing with the YAML output, we see the next general block of output concerns the metadata of the deployment. This is where we would find labels, annotations, and other non-configuration information. Note that this output will not show all possible configuration. Many settings which are set to false by default are not shown, like `podAffinity` or `nodeAffinity`.

```
metadata:
  annotations:
    deployment.kubernetes.io/revision: "1"
  creationTimestamp: 2017-12-21T13:57:07Z
  generation: 1
  labels:
    app: dev-web
  name: dev-web
  namespace: default
  resourceVersion: "774003"
  selfLink: /apis/extensions/v1beta1/namespaces/default/deployments/dev-web
  uid: d52d3a63-e656-11e7-9319-42010a800003
```

Deployment Configuration Metada

Next, you can see an explanation of the information present in the deployment metadata (the file provided on the previous page):

- `annotations:`
  These values do not configure the object, but provide further information that could be helpful to third-party applications or administrative tracking. Unlike labels, they cannot be used to select an object with **kubectl**.
- `creationTimestamp :`
  Shows when the object was originally created. Does not update if the object is edited.
- `generation` :
  How many times this object has been edited, such as changing the number of replicas, for example.
- `labels` :
  Arbitrary strings used to select or exclude objects for use with **kubectl**, or other API calls. Helpful for administrators to select objects outside of typical object boundaries.
- `name` :
  This is a **required** string, which we passed from the command line. The name must be unique to the namespace.
- `resourceVersion` :
  A value tied to the **etcd** database to help with concurrency of objects. Any changes to the database will cause this number to change.
- `selfLink` :
  References how the `kube-apiserver` will ingest this information into the API.
- `uid` :
  Remains a unique ID for the life of the object.

Deployment Configuration Spec

There are two `spec` declarations for the deployment. The first will modify the *ReplicaSet* created, while the second will pass along the Pod configuration.

```
spec:
  progressDeadlineSeconds: 600
  replicas: 1
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: dev-web
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
```

- `spec` :
  A declaration that the following items will configure the object being created.
- `progressDeadlineSeconds:`
  Time in seconds until a progress error is reported during a change. Reasons could be quotas, image issues, or limit ranges.
- `replicas` :
  As the object being created is a *ReplicaSet*, this parameter determines how many Pods should be created. If you were to use `kubectl edit` and change this value to two, a second Pod would be generated.
- `revisionHistoryLimit:`
  How many old ReplicaSet specifications to retain for rollback.

Deployment Configuration Spec

The elements present in the example we provided on the previous page are explained below (continued):

- `selector` :
  A collection of values ANDed together. All must be satisfied for the replica to match. Do not create Pods which match these selectors, as the deployment controller may try to control the resource, leading to issues.
- `matchLabels` :
  Set-based requirements of the Pod selector. Often found with the `matchExpressions` statement, to further designate where the resource should be scheduled.
- `strategy` :
  A header for values having to do with updating Pods. Works with the later listed type. Could also be set to `Recreate`, which would delete all existing pods before new pods are created. With `RollingUpdate`, you can control how many Pods are deleted at a time with the following parameters.
- `maxsurge` :
  Maximum number of Pods over desired number of Pods to create. Can be a percentage, default of 25%, or an absolute number. This creates a certain number of new Pods before deleting old ones, for continued access.
- `maxUnavailable`:
  A number or percentage of Pods which can be in a state other than `Ready` during the update process.
- `type` :
  Even though listed last in the section, due to the level of white space indentation, it is read as the type of object being configured. (e.g. `RollingUpdate`).

Deployment Configuration Pod Template

Next, we will take a look at a configuration template for the pods to be deployed. We will see some similar values.

```
template:
  metadata:
  creationTimestamp: null
    labels:
      app: dev-web
  spec:
    containers:
    - image: nginx:1.13.7-alpine
      imagePullPolicy: IfNotPresent
      name: dev-web
      resources: {}
      terminationMessagePath: /dev/termination-log
      terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    securityContext: {}
    terminationGracePeriodSeconds: 30
```

Note: If the meaning is basically the same as what was defined before, we will not repeat the definition:

- `template` :
  Data being passed to the *ReplicaSet* to determine how to deploy an object (in this case, containers).
- `containers` :
  Key word indicating that the following items of this indentation are for a container.

Deployment Configuration Pod Template

Next, we will continue to take a look at a configuration template for the pods to be deployed. If the meaning is basically the same as what was defined before, we will not repeat the definition:

- `image` :
  This is the image name passed to the container engine, typically **Docker**. The engine will pull the image and create the Pod.
- `imagePullPolicy` :
  Policy settings passed along to the container engine, about when and if an image should be downloaded or used from a local cache.
- `name` :
  The leading stub of the Pod names. A unique string will be appended.
- `resources` :
  By default, empty. This is where you would set resource restrictions and settings, such as a limit on CPU or memory for the containers.
- `terminationMessagePath` :
  A customizable location of where to output success or failure information of a container.
- `terminationMessagePolicy` :
  The default value is `File`, which holds the termination method. It could also be set to `FallbackToLogsOnError` which will use the last chunk of container log if the message file is empty and the container shows an error.
- `dnsPolicy` :
  Determines if DNS queries should go to **kube-dns** or, if set to `Default`, use the node's DNS resolution configuration.
- `restartPolicy` :
  Used should the container be restarted if killed. Automatic restarts are part of the typical strength of Kubernetes.

Deployment Configuration Status

The `Status` output is generated when the information is requested:

```
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: 2017-12-21T13:57:07Z
    lastUpdateTime: 2017-12-21T13:57:07Z
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  observedGeneration: 2
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
```

The output above shows what the same deployment were to look like if the number of replicas were increased to two. The times are different than when the deployment was first generated.

- **`availableReplicas`** :
  Indicates how many were configured by the *ReplicaSet*. This would be compared to the later value of **`readyReplicas`**, which would be used to determine if all replicas have been fully generated and without error.
- **`observedGeneration`** :
  Shows how often the deployment has been updated. This information can be used to understand the rollout and rollback situation of the deployment.

Scaling and Rolling Updates

The API server allows for the configurations settings to be updated for most values. There are some immutable values, which may be different depending on the version of Kubernetes you have deployed.

A common update is to change the number of replicas running. If this number is set to zero, there would be no containers, but there would still be a *ReplicaSet* and *Deployment*. This is the backend process when a *Deployment* is deleted.

```
$ kubectl scale deploy/dev-web --replicas=4
deployment "dev-web" scaled

$ kubectl get deployments
NAME      READY   UP-TO-DATE  AVAILABLE  AGE
dev-web   4/4     4           4          20s
```

Non-immutable values can be edited via a text editor, as well. Use `edit` to trigger an update. For example, to change the deployed version of the `nginx` web server to an older version:

```
$ kubectl edit deployment nginx
....
      containers:
      - image: nginx:1.8 #<<---Set to an older version
        imagePullPolicy: IfNotPresent
        name: dev-web
....
```

This would trigger a rolling update of the deployment. While the deployment would show an older age, a review of the Pods would show a recent update and older version of the web server application deployed.

Deployment Rollbacks

With some of the previous *ReplicaSets* of a Deployment being kept, you can also roll back to a previous revision by scaling up and down. The number of previous configurations kept is configurable, and has changed from version to version. Next, we will have a closer look at rollbacks, using the `--record` option of the `kubectl create` command, which allows annotation in the resource definition.

```
$ kubectl create deploy ghost --image=ghost --record

$ kubectl get deployments ghost -o yaml

metadata:
    annotations:
        deployment.kubernetes.io/revision: "1"
        kubernetes.io/change-cause: kubectl create deploy ghost --image=ghost --record
```

Should an update fail, due to an improper image version, for example, you can roll back the change to a working version with `kubectl rollout undo`:

```
$ kubectl set image deployment/ghost ghost=ghost:09 --all
$ kubectl rollout history deployment/ghost deployments "ghost":
REVISION     CHANGE-CAUSE
1            kubectl create deploy ghost --image=ghost --record
2            kubectl set image deployment/ghost ghost=ghost:09 --all

$ kubectl get pods
NAME                        READY     STATUS              RESTARTS    AGE
ghost-2141819201-tcths      0/1       ImagePullBackOff    0           1m

$ kubectl rollout undo deployment/ghost ; kubectl get pods
NAME                     READY    STATUS    RESTARTS    AGE
ghost-3378155678-eq5i6   1/1      Running   0           7s
```

Deployment Rollbacks

You can roll back to a specific revision with the `--to-revision=2` option.

You can also edit a Deployment using the `kubectl edit` command.

You can also pause a Deployment, and then resume.

```
$ kubectl rollout pause deployment/ghost
```

```
$ kubectl rollout resume deployment/ghost
```

Please note that you can still do a rolling update on *ReplicationControllers* with the `kubectl rolling-update` command, but this is done on the client side. Hence, if you close your client, the rolling update will stop.

Labels

Part of the metadata of an object is a *label*. Though labels are not API objects, they are an important tool for cluster administration. They can be used to select an object based on an arbitrary string, regardless of the object type. Labels are immutable as of API version `apps/v1`.

Every resource can contain labels in its metadata. By default, creating a *Deployment* with `kubectl create` adds a label, as we saw in:

```
....
    labels:
        pod-template-hash: "3378155678"
        run: ghost
....
```

You could then view labels in new columns:

```
$ kubectl get pods -l run=ghost
NAME                    READY STATUS  RESTARTS AGE
ghost-3378155678-eq5i6 1/1   Running 0        10m

$ kubectl get pods -L run
NAME                    READY STATUS  RESTARTS AGE RUN
ghost-3378155678-eq5i6 1/1   Running 0        10m ghost
nginx-3771699605-4v27e 1/1   Running 1         1h nginx
```

Labels

While you typically define labels in pod templates and in the specifications of Deployments, you can also add labels on the fly:

```
$ kubectl label pods ghost-3378155678-eq5i6 foo=bar
```

```
$ kubectl get pods --show-labels
NAME                     READY STATUS  RESTARTS AGE LABELS
ghost-3378155678-eq5i6 1/1   Running 0        11m foo=bar, pod-template-hash=3378155678,run=ghost
```

For example, if you want to force the scheduling of a pod on a specific node, you can use a *nodeSelector* in a pod definition, add specific labels to certain nodes in your cluster and use those labels in the pod.

```
....
spec:
    containers:
    - image: nginx
    nodeSelector:
        disktype: ssd
```

Services

Services Overview

As touched on previously, the Kubernetes architecture is built on the concept of transient, decoupled objects connected together. *Services* are the agents which connect Pods together, or provide access outside of the cluster, with the idea that any particular Pod could be terminated and rebuilt. Typically using *Labels*, the refreshed Pod is connected and the microservice continues to provide the expected resource via an *Endpoint* object. Google has been working on **Extensible Service Proxy** (**ESP**), based off the **nginx** HTTP reverse proxy server, to provide a more flexible and powerful object than *Endpoints*, but **ESP** has not been adopted much outside of the **Google App Engine** or **GKE** environments.

There are several different service types, with the flexibility to add more, as necessary. Each service can be exposed internally or externally to the cluster. A service can also connect internal resources to an external resource, such as a third-party database.

The `kube-proxy` agent watches the Kubernetes API for new services and endpoints being created on each node. It opens random ports and listens for traffic to the `ClusterIP:Port`, and redirects the traffic to the randomly generated service endpoints.

Services provide automatic load-balancing, matching a label query. While there is no configuration of this option, there is the possibility of session affinity via IP. Also, a headless service, one without a fixed IP nor load-balancing, can be configured.

Unique IP addresses are assigned, and configured via the **etcd** database, so that *Services* implement **iptables** to route traffic, but could leverage other technologies to provide access to resources in the future.

Service Update Pattern

Labels are used to determine which Pods should receive traffic from a service. As we have learned, *labels* can be dynamically updated for an object, which may affect which Pods continue to connect to a service.

The default update pattern is for a rolling deployment, where new Pods are added, with different versions of an application, and due to automatic load balancing, receive traffic along with previous versions of the application.

Should there be a difference in applications deployed, such that clients would have issues communicating with different versions, you may consider a more specific label for the deployment, which includes a version number. When the deployment creates a new replication controller for the update, the label would not match. Once the new Pods have been created, and perhaps allowed to fully initialize, we would edit the labels for which the Service connects. Traffic would shift to the new and ready version, minimizing client version confusion.

Accessing An Application With A Service

The basic step to access a new service is to use **kubectl**.

```
$ kubectl expose deployment/nginx --port=80 --type=NodePort
```

```
$ kubectl get svc
```

```
NAME         CLUSTER-IP   EXTERNAL-IP   PORT(S)  AGE
kubernetes   10.0.0.1     <none>        443/TCP  18h
nginx        10.0.0.112   <nodes>        80/TCP   5s
```

```
$ kubectl get svc nginx -o yaml
```

```
apiVersion: v1
kind: Service
...
spec:
    clusterIP: 10.0.0.112
    ports:
    - nodePort: 31230
...
```

Open browser `http://Public-IP:31230`

Accessing An Application With A Service

The `kubectl expose` command created a service for the `nginx` deployment. This service used port 80 and generated a random port on all the nodes. A particular `port` and `targetPort` can also be passed during object creation to avoid random values. The `targetPort` defaults to the `port`, but could be set to any value, including a string referring to a port on a backend Pod. Each Pod could have a different port, but traffic is still passed via the name. Switching traffic to a different port would maintain a client connection, while changing versions of software, for example.

The `kubectl get svc` command gave you a list of all the existing services, and we saw the `nginx` service, which was created with an internal cluster IP.

The range of cluster IPs and the range of ports used for the random `NodePort` are configurable in the API server startup options.

Services can also be used to point to a service in a different namespace, or even a resource outside the cluster, such as a legacy application not yet in Kubernetes.

Services Types

Services can be of the following types:

- **ClusterIP**
- **NodePort**
- **LoadBalancer**
- **ExternalName**.

The **ClusterIP** service type is the default, and only provides access internally (except if manually creating an external endpoint). The range of ClusterIP used is defined via an API server startup option.

The **NodePort** type is great for debugging, or when a static IP address is necessary, such as opening a particular address through a firewall. The NodePort range is defined in the cluster configuration.

The **LoadBalancer** service was created to pass requests to a cloud provider like GKE or AWS. Private cloud solutions also may implement this service type if there is a cloud provider plugin, such as with **CloudStack** and **OpenStack**. Even without a cloud provider, the address is made available to public traffic, and packets are spread among the Pods in the deployment automatically.

A newer service is **ExternalName**, which is a bit different. It has no selectors, nor does it define ports or endpoints. It allows the return of an alias to an external service. The redirection happens at the DNS level, not via a proxy or forward. This object can be useful for services not yet brought into the Kubernetes cluster. A simple change of the type in the future would redirect traffic to the internal objects.

The `kubectl proxy` command creates a local service to access a ClusterIP. This can be useful for troubleshooting or development work.

Services Diagram

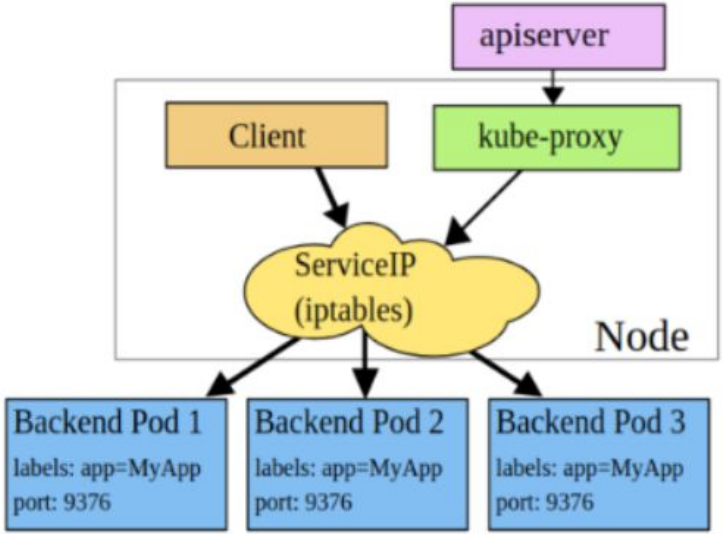
The `kube-proxy` running on cluster nodes watches the API server service resources. It presents a type of virtual IP address for services other than `ExternalName`. The mode for this process has changed over versions of Kubernetes.

In v1.0, services ran in `userspace` mode as TCP/UDP over IP or Layer 4. In the v1.1 release, the **iptables** proxy was added and became the default mode starting with v1.2.

In the **iptables** proxy mode, `kube-proxy` continues to monitor the API server for changes in *Service* and *Endpoint* objects, and updates rules for each object when created or removed. One limitation to the new mode is an inability to connect to a Pod should the original request fail, so it uses a `Readiness Probe` to ensure all containers are functional prior to connection. This mode allows for up to approximately 5000 nodes. Assuming multiple Services and Pods per node, this leads to a bottleneck in the kernel.

Another mode beginning in v1.9 is **ipvs**. While in *beta*, and expected to change, it works in the kernel space for greater speed, and allows for a configurable load-balancing algorithm, such as round-robin, shortest expected delay, least connection and several others. This can be helpful for large clusters, much past the previous 5000 node limitation. This mode assumes IPVS kernel modules are installed and running prior to `kube-proxy`.

The `kube-proxy` mode is configured via a flag sent during initialization, such as `mode=iptables` and could also be `IPVS` or `userspace`.



Traffic from ClusterIP to Pod

Local Proxies For Development

When developing an application or service, one quick way to check your service is to run a local proxy with **`kubectl`**. It will capture the shell, unless you place it in the background. When running, you can make calls to the Kubernetes API on `localhost` and also reach the ClusterIP services on their API URL. The IP and port where the proxy listens can be configured with command arguments.

Run a proxy:

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

Next, to access a `ghost` service using the local proxy, we could use the following URL, for example, at `http://localhost:8001/api/v1/namespaces/default/services/ghost`.

If the service port has a name, the path will be `http://localhost:8001/api/v1/namespaces/default/services/ghost:<port_name>`.

DNS

DNS has been provided as CoreDNS by default as of v1.13. The use of CoreDNS allows for a great amount of flexibility. Once the container starts, it will run a Server for the zones it has been configured to serve. Then, each server can load one or more plugin chains to provide other functionality.

The 30 or so in-tree plugins provide most common functionality, with an easy process to write and enable other plugins as necessary.

Common plugins can provide metrics for consumption by Prometheus, error logging, health reporting, and TLS to configure certificates for TLS and gRPC servers.

To make sure that your DNS setup works well and that services get registered, the easiest way to do it is to run a pod in the cluster and `exec` in it to do a DNS lookup.

Create this sleeping `busybox` pod with the **`kubectl create`** command :

```
apiVersion: v1
kind: Pod
metadata:
    name: busybox
    namespace: default
spec:
    containers:
    - image: busybox
      name: busy
      command:
        - sleep
        - "3600"
```

Then, use **`kubectl exec`** to do your **nslookup** like so:

```
$ kubectl exec -ti busybox -- nslookup nginx
```

```
Server: 10.0.0.10
Address 1: 10.0.0.10
Name: nginx
Address 1: 10.0.0.112
```

You can see that the DNS name `nginx` (corresponding to the *nginx* service) is registered with the ClusterIP of the service.

Volumes And Data

Volumes Overview

Container engines have traditionally not offered storage that outlives the container. As containers are considered transient, this could lead to a loss of data, or complex exterior storage options. A Kubernetes *volume* shares the Pod lifetime, not the containers within. Should a container terminate, the data would continue to be available to the new container.

A *volume* is a directory, possibly pre-populated, made available to containers in a Pod. The creation of the directory, the backend storage of the data and the contents depend on the volume type. As of v1.13, there were 27 different volume types ranging from `rbd` to gain access to Ceph, to **NFS**, to dynamic volumes from a cloud provider like Google's `gcePersistentDisk`. Each has particular configuration options and dependencies.

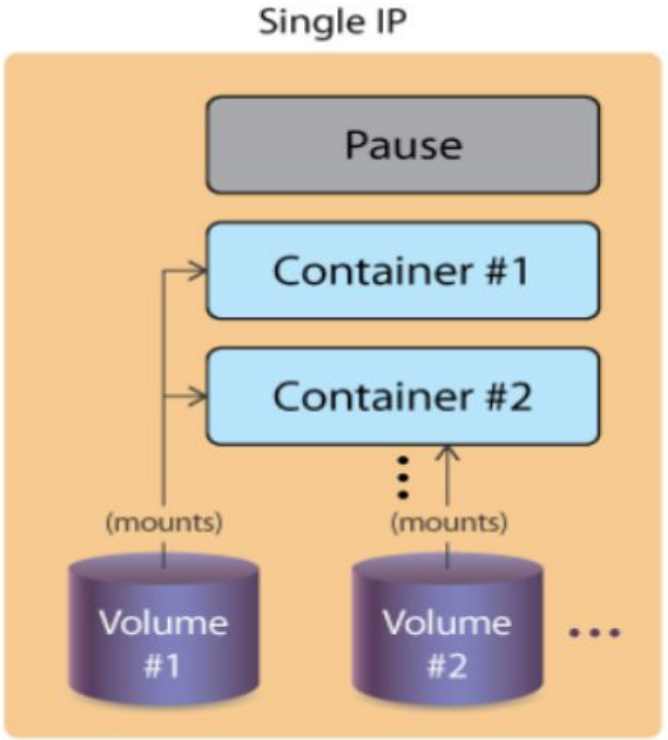
The *Container Storage Interface* (*CSI*) adoption enables the goal of an industry standard interface for container orchestration to allow access to arbitrary storage systems. Currently, volume plugins are "in-tree", meaning they are compiled and built with the core Kubernetes binaries. This "out-of-tree" object will allow storage vendors to develop a single driver and allow the plugin to be containerized. This will replace the existing **Flex** plugin which requires elevated access to the host node, a large security concern.

Should you want your storage lifetime to be distinct from a Pod, you can use *Persistent Volumes*. These allow for empty or pre-populated volumes to be claimed by a Pod using a *Persistent Volume Claim*, then outlive the Pod. Data inside the volume could then be used by another Pod, or as a means of retrieving data.

There are two API Objects which exist to provide data to a Pod already. Encoded data can be passed using a *Secret* and non-encoded data can be passed with a *ConfigMap*. These can be used to pass important data like SSH keys, passwords, or even a configuration file like `/etc/hosts`.

Introducing Volumes

A Pod specification can declare one or more volumes and where they are made available. Each requires a name, a type, and a mount point. The same volume can be made available to multiple containers within a Pod, which can be a method of container-to-container communication. A volume can be made available to multiple Pods, with each given an `access mode` to write. There is no concurrency checking, which means data corruption is probable, unless outside locking takes place.



K8s Pod Volumes

Introducing Volumes

A particular `access` mode is part of a Pod request. As a request, the user may be granted more, but not less access, though a direct match is attempted first. The cluster groups volumes with the same mode together, then sorts volumes by size, from smallest to largest. The claim is checked against each in that access mode group, until a volume of sufficient size matches. The three access modes are `RWO` (`ReadWriteOnce`), which allows read-write by a single node, `ROX` (`ReadOnlyMany`), which allows read-only by multiple nodes, and `RWX` (`ReadWriteMany`), which allows read-write by many nodes.

When a volume is requested, the local **kubelet** uses the `kubelet_pods.go` script to map the raw devices, determine and make the mount point for the container, then create the symbolic link on the host node filesystem to associate the storage to the container. The API server makes a request for the storage to the `StorageClass` plugin, but the specifics of the requests to the backend storage depend on the plugin in use.

If a request for a particular `StorageClass` was not made, then the only parameters used will be access mode and size. The volume could come from any of the storage types available, and there is no configuration to determine which of the available ones will be used.

Volume Spec

One of the many types of storage available is an `emptyDir`. The kubelet will create the directory in the container, but not mount any storage. Any data created is written to the shared container space. As a result, it would not be persistent storage. When the Pod is destroyed, the directory would be deleted along with the container.

```
apiVersion: v1
kind: Pod
metadata:
    name: busybox
    namespace: default
spec:
    containers:
    - image: busybox
      name: busy
      command:
        - sleep
        - "3600"
      volumeMounts:
    - mountPath: /scratch
      name: scratch-volume
    volumes:
    - name: scratch-volume
            emptyDir: {}
```

The YAML file above would create a Pod with a single container with a volume named `scratch-volume` created, which would create the `/scratch` directory inside the container.

Volume Types

There are several types that you can use to define volumes, each with their pros and cons. Some are local, and many make use of network-based resources.

In **GCE** or **AWS**, you can use volumes of type `GCEpersistentDisk` or `awsElasticBlockStore`, which allows you to mount **GCE** and **EBS** disks in your Pods, assuming you have already set up accounts and privileges.

`emptyDir` and `hostPath` volumes are easy to use. As mentioned, `emptyDir` is an empty directory that gets erased when the Pod dies, but is recreated when the container restarts. The `hostPath` volume mounts a resource from the host node filesystem. The resource could be a directory, file socket, character, or block device. These resources must already exist on the host to be used. There are two types, `DirectoryOrCreate` and `FileOrCreate`, which create the resources on the host, and use them if they don't already exist.

NFS (Network File System) and **iSCSI** (Internet Small Computer System Interface) are straightforward choices for multiple readers scenarios.

rbd for block storage or **CephFS** and **GlusterFS**, if available in your **Kubernetes** cluster, can be a good choice for multiple writer needs.

Besides the volume types we just mentioned, there are many other possible, with more being added: `azureDisk`, `azureFile`, `csi`, `downwardAPI`, `fc` (fibre channel), `flocker`, `gitRepo`, `local`, `projected`, `portworxVolume`, `quobyte`, `scaleIO`, `secret`, `storageos`, `vsphereVolume`, `persistentVolumeClaim`, etc.

Shared Volumes Examples

The following YAML file creates a pod with two containers, both with access to a shared volume:

```
containers:
       - image: busybox
     volumeMounts:
       - mountPath: /busy
     name: test
     name: busy
       - image: busybox
     volumeMounts:
       - mountPath: /box
     name: test
     name: box
     volumes:
       - name: test
       emptyDir: {}
```

```
$ kubectl exec -ti busybox -c box -- touch /box/foobar
```

```
$ kubectl exec -ti busybox -c busy -- ls -l /busy total 0
-rw-r--r-- 1 root root 0 Nov 19 16:26 foobar
```

You could use `emptyDir` or `hostPath` easily, since those types do not require any additional setup, and will work in your Kubernetes cluster.

Note that one container wrote, and the other container had immediate access to the data. There is nothing to keep the containers from overwriting the other's data. Locking or versioning considerations must be part of the application to avoid corruption.

Persistent Volumes And Claims

A *persistent volume* (*pv*) is a storage abstraction used to retain data longer then the Pod using it. Pods define a volume of type `persistentVolumeClaim` (`pvc`)with various parameters for size and possibly the type of backend storage known as its `StorageClass`. The cluster then attaches the `persistentVolume`.

Kubernetes will dynamically use volumes that are available, irrespective of its storage type, allowing claims to any backend storage.

There are several phases to persistent storage:

- **Provisioning** can be from PVs created in advance by the cluster administrator, or requested from a dynamic source, such as the cloud provider.
- **Binding** occurs when a control loop on the master notices the PVC, containing an amount of storage, access request, and optionally, a particular `StorageClass`. The watcher locates a matching PV or waits for the `StorageClass` provisioner to create one. The PV must match at least the storage amount requested, but may provide more.
- The **use** phase begins when the bound volume is mounted for the Pod to use, which continues as long as the Pod requires.
- **Releasing** happens when the Pod is done with the volume and an API request is sent, deleting the PVC. The volume remains in the state from when the claim is deleted until available to a new claim. The resident data remains depending on the `persistentVolumeReclaimPolicy`.
- The **reclaim** phase has three options:
  - **Retain**, which keeps the data intact, allowing for an administrator to handle the storage and data.
  - **Delete** tells the volume plugin to delete the API object, as well as the storage behind it.
  - The **Recycle** option runs an `rm -rf /mountpoint` and then makes it available to a new claim. With the stability of dynamic provisioning, the **Recycle** option is planned to be deprecated.

```
$ kubectl get pv
$ kubectl get pvc
```

Persistent Volumes

The following example shows a basic declaration of a `PersistentVolume` using the `hostPath` type.

```
kind: PersistentVolume
apiVersion: v1
metadata:
    name: 10Gpv01
    labels:
         type: local
spec:
    capacity:
        storage: 10Gi
    accessModes:
        - ReadWriteOnce
    hostPath:
        path: "/somepath/data01"
```

Each type will have its own configuration settings. For example, an already created Ceph or GCE Persistent Disk would not need to be configured, but could be claimed from the provider.

Persistent volumes are not a namespaces object, but persistent volume claims are. A beta feature of v1.13 allows for static provisioning of Raw Block Volumes, which currently support the Fibre Channel plugin, AWS EBS, Azure Disk and RBD plugins among others.

The use of locally attached storage has been graduated to a stable feature. This feature is often used as part of distributed filesystems and databases.

Persistent Volumes Claim

With a persistent volume created in your cluster, you can then write a manifest for a claim and use that claim in your pod definition. In the Pod, the volume uses the `persistentVolumeClaim`.

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
    name: myclaim
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
                storage: 8GI
```

In the Pod:

```
spec:
    containers:
....
    volumes:
        - name: test-volume
          persistentVolumeClaim:
                claimName: myclaim
```

Persistent Volumes Claim

The Pod configuration could also be as complex as this:

```
volumeMounts:
      - name: Cephpd
        mountPath: /data/rbd
  volumes:
    - name: rbdpd
      rbd:
        monitors:
        - '10.19.14.22:6789'
        - '10.19.14.23:6789'
        - '10.19.14.24:6789'
        pool: k8s
        image: client
        fsType: ext4
        readOnly: true
        user: admin
        keyring: /etc/ceph/keyring
        imageformat: "2"
        imagefeatures: "layering"
```

Dynamic Provisioning

While handling volumes with a persistent volume definition and abstracting the storage provider using a claim is powerful, a cluster administrator still needs to create those volumes in the first place. Starting with Kubernetes v1.4, **Dynamic Provisioning** allowed for the cluster to request storage from an exterior, pre-configured source. API calls made by the appropriate plugin allow for a wide range of dynamic storage use.

The `StorageClass` API resource allows an administrator to define a persistent volume provisioner of a certain type, passing storage-specific parameters.

With a `StorageClass` created, a user can request a claim, which the API Server fills via auto-provisioning. The resource will also be reclaimed as configured by the provider. **AWS** and **GCE** are common choices for dynamic storage, but other options exist, such as a **Ceph** cluster or iSCSI. Single, default class is possible via annotation.

Here is an example of a `StorageClass` using GCE:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast                          # Could be any name
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Secrets

Pods can access local data using volumes, but there is some data you don't want readable to the naked eye. Passwords may be an example. Using the *Secret* API resource, the same password could be encoded or encrypted.

You can create, get, or delete secrets:

```
$ kubectl get secrets
```

Secrets can be encoded manually or via `kubectl create secret`:

```
$ kubectl create secret generic --help
```

```
$ kubectl create secret generic mysql --from-literal=password=root
```

A secret is not encrypted, only **base64**-encoded, by default. You must create an *EncryptionConfiguration* with a key and proper identity. Then, the **kube-apiserver** needs the `--encryption-provider-config` flag set to a previously configured provider, such as *aescbc* or *ksm*. Once this is enabled, you need to recreate every secret, as they are encrypted upon write.

Multiple keys are possible. Each key for a provider is tried during decryption. The first key of the first provider is used for encryption. To rotate keys, first create a new key, restart (all) **kube-apiserver** processes, then recreate every secret.

You can see the encoded string inside the secret with **kubectl**. The secret will be decoded and be presented as a string saved to a file. The file can be used as an environmental variable or in a new directory, similar to the presentation of a volume.

Secrets

A secret can be made manually as well, then inserted into a YAML file:

```
$ echo LFTr@1n | base64
TEZUckAxbgo=

$ vim secret.yaml
apiVersion: v1
kind: Secret
metadata:
  name: LF-secret
data:
  password: TEZUckAxbgo=
```

Secrets As Environment Variables

A secret can be used as an environmental variable in a Pod. You can see one being configured in the following example:

```
...
spec:
    containers:
    - image: mysql:5.5
      env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
            secretKeyRef:
                name: mysql
                key: password
          name: mysql
```

There is no limit to the number of Secrets used, but there is a 1MB limit to their size. Each secret occupies memory, along with other API objects, so very large numbers of secrets could deplete memory on a host.

They are stored in the **tmpfs** storage on the host node, and are only sent to the host running Pod. All volumes requested by a Pod must be mounted before the containers within the Pod are started. So, a secret must exist prior to being requested.

Mounting Secrets As Volumes

You can also mount secrets as files using a volume definition in a pod manifest. The mount path will contain a file whose name will be the key of the secret created with the **kubectl create secret** step earlier.

```
...
spec:
    containers:
    - image: busybox
      command:
        - sleep
        - "3600"
      volumeMounts:
      - mountPath: /mysqlpassword
        name: mysql
      name: busy
    volumes:
    - name: mysql
        secret:
            secretName: mysql
```

Once the pod is running, you can verify that the secret is indeed accessible in the container:

```
$ kubectl exec -ti busybox -- cat /mysqlpassword/password
LFTr@1n
```

Portable Data With ConfigMaps

A similar API resource to Secrets is the *ConfigMap*, except the data is not encoded. In keeping with the concept of decoupling in Kubernetes, using a *ConfigMap* decouples a container image from configuration artifacts.

They store data as sets of key-value pairs or plain configuration files in any format. The data can come from a collection of files or all files in a directory. It can also be populated from a literal value.

A *ConfigMap* can be used in several different ways. A Pod can use the data as environmental variables from one or more sources. The values contained inside can be passed to commands inside the pod. A Volume or a file in a Volume can be created, including different names and particular access modes. In addition, cluster components like controllers can use the data.

Let's say you have a file on your local filesystem called `config.js`. You can create a *ConfigMap* that contains this file. The `configmap` object will have a `data` section containing the content of the file:

```
$ kubectl get configmap foobar -o yaml
```

```
kind: ConfigMap
apiVersion: v1
metadata:
    name: foobar
data:
    config.js: |
        {
...
```

Portable Data With ConfigMaps

ConfigMaps can be consumed in various ways:

- Pod environmental variables from single or multiple *ConfigMaps*
- Use *ConfigMap* values in Pod commands
- Populate Volume from *ConfigMap*
- Add *ConfigMap* data to specific path in Volume
- Set file names and access mode in Volume from *ConfigMap* data
- Can be used by system components and controllers.

Using ConfigMaps

Like secrets, you can use *ConfigMaps* as environment variables or using a volume mount. They must exist prior to being used by a Pod, unless marked as `optional`. They also reside in a specific namespace.

In the case of environment variables, your pod manifest will use the **`valueFrom`** key and the **`configMapKeyRef`** value to read the values. For instance:

```
env:
- name: SPECIAL_LEVEL_KEY
  valueFrom:
    configMapKeyRef:
      name: special-config
      key: special.how
```

With volumes, you define a volume with the `configMap` type in your pod and mount it where it needs to be used.

```
volumes:
    - name: config-volume
      configMap:
        name: special-config
```

INGRESS

INGRESS Overview

In an earlier chapter, we learned about using a *Service* to expose a containerized application outside of the cluster. We use *Ingress Controllers* and *Rules* to do the same function. The difference is efficiency. Instead of using lots of services, such as `LoadBalancer`, you can route traffic based on the request host or path. This allows for centralization of many services to a single point.

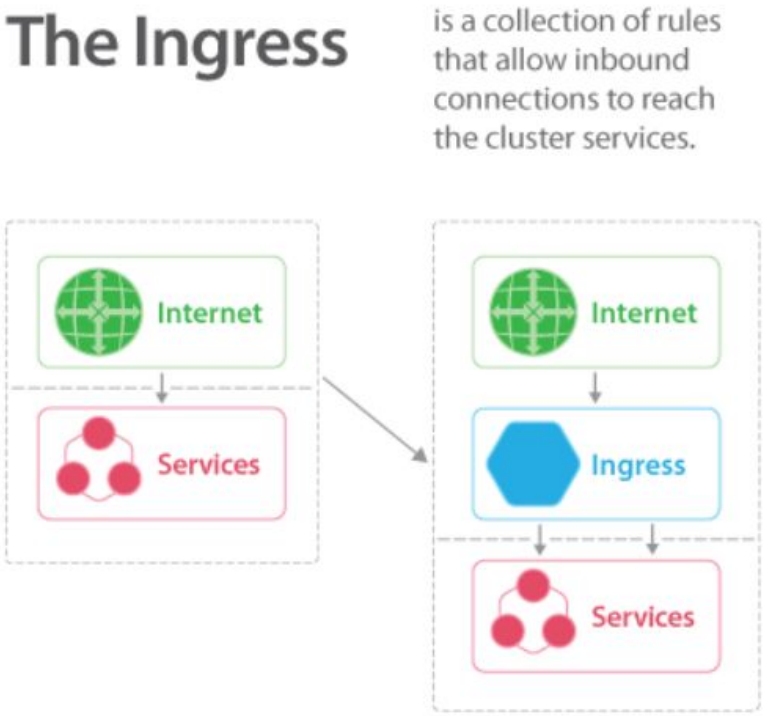
An *Ingress Controller* is different than most controllers, as it does not run as part of the **kube-controller-manager** binary. You can deploy multiple controllers, each with unique configurations. A controller uses *Ingress Rules* to handle traffic to and from outside the cluster.

There are two supported controllers, `GCE` and `nginx`. `HAProxy` and `Traefix.io` are also in common use. Any tool capable of reverse proxy should work. These agents consume rules and listen for associated traffic. An *Ingress Rule* is an API resource that you can create with **kubectl**. When you create that resource, it reprograms and reconfigures your *Ingress Controller* to allow traffic to flow from the outside to an internal service. You can leave a service as a `ClusterIP` type and define how the traffic gets routed to that internal service using an *Ingress Rule*.

INGRESS Controller

An *Ingress Controller* is a daemon running in a Pod which watches the `/ingresses` endpoint on the API server, which is found under the `extensions/v1beta1` group for new objects. When a new endpoint is created, the daemon uses the configured set of rules to allow inbound connection to a service, most often HTTP traffic. This allows easy access to a service through an edge router to Pods, regardless of where the Pod is deployed.

Multiple *Ingress Controllers* can be deployed. Traffic should use annotations to select the proper controller. The lack of a matching annotation will cause every controller to attempt to satisfy the ingress traffic.



Ingress Controller for inbound connections

Nginx

Deploying an `nginx` controller has been made easy through the use of provided YAML files, which can be found in the ingress-nginx GitHub repository.

This page has configuration files to configure `nginx` on several platforms, such as AWS, GKE, Azure, and bare-metal, among others.

As with any *Ingress Controller*, there are some configuration requirements for proper deployment. Customization can be done via a *ConfigMap*, *Annotations*, or, for detailed configuration, a `Custom template`:

- Easy integration with RBAC
- Uses the annotation `kubernetes.io/ingress.class: "nginx"`
- L7 traffic requires the `proxy-real-ip-cidr` setting
- Bypasses `kube-proxy` to allow session affinity
- Does not use `conntrack` entries for iptables DNAT
- TLS requires the `host` field to be defined.

Google Load Balancer Controller

There are several objects which need to be created to deploy the GCE Ingress Controller. YAML files are available to make the process easy. Be aware that several objects would be created for each service, and currently, quotas are not evaluated prior to creation.

The GLBC Controller must be created and started first. Also, you must create a ReplicationController with a single replica, three services for the application Pod, and an Ingress with two hostnames and three endpoints for each service. The backend is a group of virtual machine instances, Instance Group.

Each path for traffic uses a group of like objects referred to as a *pool*. Each pool regularly checks the next hop up to ensure connectivity.

The multi-pool path is:

`Global Forwarding Rule -> Target HTTP Proxy -> URL map -> Backend Service -> Instance Group`.

Currently, the TLS Ingress only supports port 443 and assumes TLS termination. It does not support SNI, only using the first certificate. The TLS secret must contain keys named `tls.crt` and `tls.key` .

Ingress Api Resources

Ingress objects are still an extension API, like Deployments and ReplicaSets. A typical Ingress object that you can `POST` to the API server is:

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
name: ghost
spec:
rules:
    - host: ghost.192.168.99.100.nip.io
http:
paths:
    - backend:
            serviceName: ghost
            servicePort: 2368
```

You can manage ingress resources like you do pods, deployments, services etc:

```
$ kubectl get ingress
$ kubectl delete ingress <ingress_name>
$ kubectl edit ingress <ingress_name>
```

Deploying The Ingress Controller

To deploy an Ingress Controller, it can be as simple as creating it with **kubectl**. The source for a sample controller deployment is available on GitHub.

```
$ kubectl create -f backend.yaml
```

The result will be a set of pods managed by a replication controller and some internal services. You will notice a default HTTP backend which serves 404 pages.

```
$ kubectl get pods,rc,svc

NAME                              READY  STATUS    RESTARTS  AGE
po/default-http-backend-xvep8     1/1    Running   0         4m
po/nginx-ingress-controller-fkshm 1/1    Running   0         4m

NAME                    DESIRED CURRENT READY AGE
rc/default-http-backend 1       1       0     4m

NAME                      CLUSTER-IP  EXTERNAL-IP  PORT(S)  AGE
svc/default-http-backend  10.0.0.212  <none>       80/TCP   4m
svc/kubernetes            10.0.0.1    <none>       443/TCP  77d
```

Creating an Ingress Rule

To get exposed with ingress quickly, you can go ahead and try to create a similar rule as mentioned on the previous page. First, start a `Ghost` deployment and expose it with an internal `ClusterIP` service:

```
$ kubectl run ghost --image=ghost
$ kubectl expose deployments ghost --port=2368
```

With the deployment exposed and the Ingress rules in place, you should be able to access the application from outside the cluster.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
    name: ghost
spec:
    rules:
    - host: ghost.192.168.99.100.nip.io
      http:
      paths:
      - backend:
            serviceName: ghost
            servicePort: 2368
```

Multiple Rules

On the previous page, we defined a single rule. If you have multiple services, you can define multiple rules in the same Ingress, each rule forwarding traffic to a specific service.

```
rules:
- host: ghost.192.168.99.100.nip.io
  http:
    paths:
    - backend:
        serviceName: ghost
        servicePort: 2368
- host: nginx.192.168.99.100.nip.io
  http:
    paths:
    - backend:
        serviceName: nginx
        servicePort: 80
```

SCHEDULING

kube-scheduler

The larger and more diverse a Kubernetes deployment becomes, the more administration of scheduling can be important. The **kube-scheduler** determines which nodes will run a Pod, using a topology-aware algorithm.

Users can set the priority of a pod, which will allow preemption of lower priority pods. The eviction of lower priority pods would then allow the higher priority pod to be scheduled.

The scheduler tracks the set of nodes in your cluster, filters them based on a set of predicates, then uses priority functions to determine on which node each Pod should be scheduled. The Pod specification as part of a request is sent to the **kubelet** on the node for creation.

The default scheduling decision can be affected through the use of *Labels* on nodes or Pods. Labels of **podAffinity**, **taints**, and **pod bindings** allow for configuration from the Pod or the node perspective. Some, like **tolerations**, allow a Pod to work with a node, even when the node has a **taint** that would otherwise preclude a Pod being scheduled.

Not all labels are drastic. Affinity settings may encourage a Pod to be deployed on a node, but would deploy the Pod elsewhere if the node was not available. Sometimes, documentation may use the term *require*, but practice shows the setting to be more of a *request*. As beta features, expect the specifics to change. Some settings will evict Pods from a node should the required condition no longer be true, such as `requiredDuringScheduling, RequiredDuringExecution`.

Other options, like a custom scheduler, need to be programmed and deployed into your Kubernetes cluster.

Predicates

The scheduler goes through a set of filters, or **predicates**, to find available nodes, then ranks each node using *priority functions*. The node with the highest rank is selected to run the Pod.

```
predicatesOrdering = []string{CheckNodeConditionPred,
GeneralPred, HostNamePred, PodFitsHostPortsPred,
MatchNodeSelectorPred, PodFitsResourcesPred, NoDiskConflictPred,
PodToleratesNodeTaintsPred, PodToleratesNodeNoExecuteTaintsPred,
CheckNodeLabelPresencePred, checkServiceAffinityPred,
MaxEBSVolumeCountPred, MaxGCEPDVolumeCountPred,
MaxAzureDiskVolumeCountPred, CheckVolumeBindingPred,
NoVolumeZoneConflictPred, CheckNodeMemoryPressurePred,
CheckNodeDiskPressurePred, MatchInterPodAffinityPred}
```

The **predicates**, such as `PodFitsHost` or `NoDiskConflict`, are evaluated in a particular and configurable order. In this way, a node has the least amount of checks for new Pod deployment, which can be useful to exclude a node from unnecessary checks if the node is not in the proper condition.

For example, there is a filter called `HostNamePred`, which is also known as `HostName`, which filters out nodes that do not match the node name specified in the pod specification. Another predicate is `PodFitsResources` to make sure that the available CPU and memory can fit the resources required by the Pod.

The scheduler can be updated by passing a configuration of `kind: Policy` which can order predicates, give special weights to priorities and even `hardPodAffinitySymmetricWeight` which deploys Pods such that if we set Pod A to run with Pod B, then Pod B should automatically be run with Pod A.

Scheduling Policies

The default scheduler contains a number of predicates and priorities; however, these can be changed via a scheduler policy file.

A short version is shown below:

```
"kind" : "Policy",
"apiVersion" : "v1",
"predicates" : [
        {"name" : "MatchNodeSelector", "order": 6},
        {"name" : "PodFitsHostPorts", "order": 2},
        {"name" : "PodFitsResources", "order": 3},
        {"name" : "NoDiskConflict", "order": 4},
        {"name" : "PodToleratesNodeTaints", "order": 5},
        {"name" : "PodFitsHost", "order": 1}
        ],
"priorities" : [
        {"name" : "LeastRequestedPriority", "weight" : 1},
        {"name" : "BalancedResourceAllocation", "weight" : 1},
        {"name" : "ServiceSpreadingPriority", "weight" : 2},
        {"name" : "EqualPriority", "weight" : 1}
        ],
"hardPodAffinitySymmetricWeight" : 10
}
```

Typically, you will configure a scheduler with this policy using the `--policy-config-file` parameter and define a name for this scheduler using the `--scheduler-name` parameter. You will then have two schedulers running and will be able to specify which scheduler to use in the pod specification.

Pod Specification

Most scheduling decisions can be made as part of the Pod specification. A pod specification contains several fields that inform scheduling, namely:

- **`nodeName`**
- **`nodeSelector`**
- **`affinity`**
- **`schedulerName`**
- **`tolerations`**

The `nodeName` and `nodeSelector` options allow a Pod to be assigned to a single node or a group of nodes with particular labels.

`Affinity` and `anti-affinity` can be used to `require` or `prefer` which node is used by the scheduler. If using a preference instead, a matching node is chosen first, but other nodes would be used if no match is present.

The use of `taints` allows a node to be labeled such that Pods would not be scheduled for some reason, such as the master node after initialization. A `toleration` allows a Pod to ignore the taint and be scheduled assuming other requirements are met.

Should none of these options meet the needs of the cluster, there is also the ability to deploy a custom scheduler. Each Pod could then include a `schedulerName` to choose which schedule to use.

Specifying The Node Label

The `nodeSelector` field in a pod specification provides a straightforward way to target a node or a set of nodes, using one or more key-value pairs.

```
spec:
  containers:
  - name: redis
    image: redis
  nodeSelector:
    net: fast
```

Setting the `nodeSelector` tells the scheduler to place the pod on a node that matches the labels. All listed selectors must be met, but the node could have more labels. In the example above, any node with a key of `net` set to `fast` would be a candidate for scheduling. Remember that labels are administrator-created tags, with no tie to actual resources. This node could have a slow network.

The pod would remain `Pending` until a node is found with the matching labels.

The use of affinity/anti-affinity should be able to express every feature as `nodeSelector`. When affinity becomes stable, the plan is to deprecate `nodeSelector`.

Pod Affinity Rules

Pods which may communicate a lot or share data may operate best if co-located, which would be a form of affinity. For greater fault tolerance, you may want Pods to be as separate as possible, which would be anti-affinity. These settings are used by the scheduler based on the labels of Pods that are already running. As a result, the scheduler must interrogate each node and track the labels of running Pods. Clusters larger than several hundred nodes may see significant performance loss. Pod affinity rules use `In`, `NotIn`, `Exists`, and `DoesNotExist` operators.

The use of `requiredDuringSchedulingIgnoredDuringExecution` means that the Pod will not be scheduled on a node unless the following operator is true. If the operator changes to become false in the future, the Pod will continue to run. This could be seen as a `hard` rule.

Similarly, `preferredDuringSchedulingIgnoredDuringExecution` will choose a node with the desired setting before those without. If no properly-labeled nodes are available, the Pod will execute anyway. This is more of a soft setting, which declares a preference instead of a requirement.

With the use of `podAffinity`, the scheduler will try to schedule Pods together. The use of `podAntiAffinity` would cause the scheduler to keep Pods on different nodes.

The `topologyKey` allows a general grouping of Pod deployments. Affinity (or the inverse anti-affinity) will try to run on nodes with the declared topology key and running Pods with a particular label. The `topologyKey` could be any legal key, with some important considerations. If using `requiredDuringScheduling` and the admission controller `LimitPodHardAntiAffinityTopology` setting, the `topologyKey` must be set to `kubernetes.io/hostname`. If using `PreferredDuringScheduling`, an empty `topologyKey` is assumed to be all, or the combination of `kubernetes.io/hostname`, `failure-domain.beta.kubernetes.io/zone` and `failure-domain.beta.kubernetes.io/region`.

PodAffinity Example

An example of `affinity` and `podAffinity` settings can be seen below. This also requires a particular label to be matched when the Pod starts, but not required if the label is later removed.

```
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
      - labelSelector:
          matchExpressions:
          - key: security
            operator: In
            values: - S1
        topologyKey: failure-domain.beta.kubernetes.io/zone
```

Inside the declared topology zone, the Pod can be scheduled on a node running a Pod with a key label of `security` and a value of `S1`. If this requirement is not met, the Pod will remain in a `Pending` state.

PodAntiAffinity Example

Similarly to **podAffinity**, we can try to avoid a node with a particular label. In this case, the scheduler will prefer to avoid a node with a key set to **security** and value of **s2**.

```
podAntiAffinity:
  preferredDuringSchedulingIgnoredDuringExecution:
  - weight: 100
    podAffinityTerm:
      labelSelector:
        matchExpressions:
        - key: security
          operator: In
          values:
          - s2
      topologyKey: kubernetes.io/hostname
```

In a large, varied environment, there may be multiple situations to be avoided. As a preference, this setting tries to avoid certain labels, but will still schedule the Pod on some node. As the Pod will still run, we can provide a **weight** to a particular rule. The weights can be declared as a value from 1 to 100. The scheduler then tries to choose, or avoid the node with the greatest combined value.

Node Affinity Rules

Where Pod affinity/anti-affinity has to do with other Pods, the use of *nodeAffinity* allows Pod scheduling based of node labels. This is similar, and will some day replace the use of the *nodeSelector* setting. The scheduler will not look at other Pods on the system, but the labels of the nodes. This should have much less performance impact on the cluster, even with a large number of nodes.

- Uses `In`, `NotIn`, `Exists`, `DoesNotExist` operators
- `requiredDuringSchedulingIgnoredDuringExecution`
- `preferredDuringSchedulingIgnoredDuringExecution`
- Planned for future: `requiredDuringSchedulingRequiredDuringExecution`

Until *nodeSelector* has been fully deprecated, both the selector and required labels must be met for a Pod to be scheduled.

Node Affinity Example

```
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/colo-tx-name
            operator: In
            values:
            - tx-aus
            - tx-dal
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: disk-speed
            operator: In
            values:
            - fast
            - quick
```

The first **nodeAffinity** rule requires a node with a key of **kubernetes.io/colo-tx-name** which has one of two possible values: **tx-aus** or **tx-dal**.

The second rule gives extra weight to nodes with a key of **disk-speed** with a value of **fast** or **quick**. The Pod will be scheduled on some node - in any case, this just prefers a particular label.

Taints

A node with a particular `taint` will repel Pods without `tolerations` for that `taint`. A taint is expressed as `key=value:effect`. The `key` and the value `value` are created by the administrator.

The `key` and `value` used can be any legal string, and this allows flexibility to prevent Pods from running on nodes based off of any need. If a Pod does not have an existing `toleration`, the scheduler will not consider the tainted node.

There are three effects, or ways to handle Pod scheduling:

- `NoSchedule`
  The scheduler will not schedule a Pod on this node, unless the Pod has this `toleration`. Existing Pods continue to run, regardless of `toleration`.
- `PreferNoSchedule`
  The scheduler will avoid using this node, unless there are no untainted nodes for the Pods `toleration`. Existing Pods are unaffected.
- `NoExecute`
  This `taint` will cause existing Pods to be evacuated and no future Pods scheduled. Should an existing Pod have a toleration, it will continue to run. If the Pod `tolerationSeconds` is set, they will remain for that many seconds, then be evicted. Certain node issues will cause the **kubelet** to add 300 second `tolerations` to avoid unnecessary evictions.

If a node has multiple `taints`, the scheduler ignores those with matching `tolerations`. The remaining unignored `taints` have their typical effect.

The use of `TaintBasedEvictions` is still an alpha feature. The **kubelet** uses `taints` to rate-limit evictions when the node has problems.

Tolerations

Setting `tolerations` on a node are used to schedule Pods on tainted nodes. This provides an easy way to avoid Pods using the node. Only those with a particular `toleration` would be scheduled.

An operator can be included in a Pod specification, defaulting to `Equal` if not declared. The use of the operator `Equal` requires a value to match. The `Exists` operator should not be specified. If an empty key uses the `Exists` operator, it will tolerate every taint. If there is no `effect`, but a key and operator are declared, all `effects` are matched with the declared key.

```
tolerations:
- key: "server"
  operator: "Equal"
  value: "ap-east"
  effect: "NoExecute"
  tolerationSeconds: 3600
```

In the above example, the Pod will remain on the server with a key of `server` and a value of `ap-east` for 3600 seconds after the node has been tainted with `NoExecute`. When the time runs out, the Pod will be evicted.

Logging And Troubleshooting

Overview

Kubernetes relies on API calls and is sensitive to network issues. Standard Linux tools and processes are the best method for troubleshooting your cluster. If a shell, such as bash, is not available in an affected Pod, consider deploying another similar pod with a shell, like `busybox`. DNS configuration files and tools like **dig** are a good place to start. For more difficult challenges, you may need to install other tools, like **tcpdump**.

Large and diverse workloads can be difficult to track, so monitoring of usage is essential. Monitoring is about collecting key metrics, such as CPU, memory, and disk usage, and network bandwidth on your nodes, as well as monitoring key metrics in your applications. These features are being ingested into Kubernetes with the Metric Server, which is a cut-down version of the now deprecated Heapster. Once installed, the Metrics Server exposes a standard API which can be consumed by other agents, such as autoscalers. Once installed, this endpoint can be found here on the master server: `/apis/metrics/k8s.io/`.

Logging activity across all the nodes is another feature not part of Kubernetes. Using **Fluentd** can be a useful data collector for a unified logging layer. Having aggregated logs can help visualize the issues, and provides the ability to search all logs. It is a good place to start when local network troubleshooting does not expose the root cause. It can be downloaded from the Fluentd website.

Another project from CNCF combines logging, monitoring, and alerting and is called **Prometheus** - you can learn more from the Prometheus website. It provides a time-series database, as well as integration with **Grafana** for visualization and dashboards.

We are going to review some of the basic `kubectl` commands that you can use to debug what is happening, and we will walk you through the basic steps to be able to debug your containers, your pending containers, and also the systems in Kubernetes.

Basic Troubleshooting Steps

The troubleshooting flow should start with the obvious. If there are errors from the command line, investigate them first. The symptoms of the issue will probably determine the next step to check. Working from the application running inside a container to the cluster as a whole may be a good idea. The application may have a shell you can use, for example:

```
$ kubectl exec -ti <busybox_pod> -- /bin/sh
```

If the Pod is running, use `kubectl logs pod-name` to view the standard out of the container. Without logs, you may consider deploying a `sidecar` container in the Pod to generate and handle logging. The next place to check is networking, including DNS, firewalls and general connectivity, using standard Linux commands and tools.

Security settings can also be a challenge. **RBAC**, covered in the security chapter, provides mandatory or discretionary access control in a granular manner. **SELinux** and **AppArmor** are also common issues, especially with network-centric applications.

A newer feature of Kubernetes is the ability to enable auditing for the kube-apiserver, which can allow a view into actions after the API call has been accepted.

The issues found with a decoupled system like Kubernetes are similar to those of a traditional datacenter, plus the added layers of Kubernetes controllers:

- Errors from the command line
- Pod logs and state of Pods
- Use shell to troubleshoot Pod DNS and network
- Check node logs for errors, make sure there are enough resources allocated
- RBAC, SELinux or AppArmor for security settings
- API calls to and from controllers to kube-apiserver
- Enable auditing

Cluster Start Sequence

The cluster startup sequence begins with systemd if you built the cluster using kubeadm. Other tools may leverage a different method. Use `systemctl status kubelet.service` to see the current state and configuration files used to run the kubelet binary.

- Uses `/etc/systemd/system/kubelet.service.d/10-kubeadm.conf`

Inside of the `config.yaml` file you will find several settings for the binary, including the `staticPodPath` which indicates the directory where kubelet will read every yaml file and start every pod. If you put a yaml file in this directory, it is a way to troubleshoot the scheduler, as the pod is created with any requests to the scheduler.

- Uses `/var/lib/kubelet/config.yaml` configuration file
- `staticPodPath` is set to `/etc/kubernetes/manifests/`

The four default yaml files will start the base pods necessary to run the cluster:

- kubelet creates all pods from *.yaml in directory: kube-apiserver, etcd, kube-controller-manager, kube-scheduler.

Once the watch loops and controllers from **kube-controller-manager** run using etcd data, the rest of the configured objects will be created.

Monitoring

Monitoring is about collecting metrics from the infrastructure, as well as applications.

The long used and now deprecated Heapster has been replaced with an integrated Metrics Server. Once installed and configured, the server exposes a standard API which other agents can use to determine usage. It can also be configured to expose custom metrics, which then could also be used by autoscalers to determine if an action should take place.

Prometheus is part of the Cloud Native Computing Foundation (CNCF). As a Kubernetes plugin, it allows one to scrape resource usage metrics from Kubernetes objects across the entire cluster. It also has several client libraries which allow you to instrument your application code in order to collect application level metrics.

Logging Tools

Logging, like monitoring, is a vast subject in IT. It has many tools that you can use as part of your arsenal.

Typically, logs are collected locally and aggregated before being ingested by a search engine and displayed via a dashboard which can use the search syntax. While there are many software stacks that you can use for logging, the **Elasticsearch**, **Logstash**, and **Kibana Stack** (ELK) has become quite common.

In **Kubernetes**, the **`kubelet`** writes container logs to local files (via the **Docker** logging driver). The **`kubectl logs`** command allows you to retrieve these logs.

Cluster-wide, you can use **Fluentd** to aggregate logs. Check the cluster administration logging concepts for a detailed description.

Fluentd is part of the **Cloud Native Computing Foundation** and, together with **Prometheus**, they make a nice combination for monitoring and logging. You can find a detailed walk-through of running Fluentd on Kubernetes in the Kubernetes documentation.

Setting up **Fluentd** for **Kubernetes** logging is a good exercise in understanding *DaemonSets*. **Fluentd** agents run on each node via a *DaemonSet*, they aggregate the logs, and feed them to an **Elasticsearch** instance prior to visualization in a **Kibana** dashboard.

Custom Resource Definitions

Custom Resources

We have been working with built-in resources, or API endpoints. The flexibility of Kubernetes allows for the dynamic addition of new resources as well. Once these *Custom Resources* have been added, the objects can be created and accessed using standard calls and commands, like **kubectl**. The creation of a new object stores new structured data in the **etcd** database and allows access via **kube-apiserver**.

To make a new custom resource part of a declarative API, there needs to be a controller to retrieve the structured data continually and act to meet and maintain the declared state. This controller, or `operator`, is an agent that creates and manages one or more instances of a specific stateful application. We have worked with built-in controllers such as *Deployments*, *DaemonSets* and other resources.

The functions encoded into a custom operator should be all the tasks a human would need to perform if deploying the application outside of Kubernetes. The details of building a custom controller are outside the scope of this course, and thus, not included.

There are two ways to add custom resources to your Kubernetes cluster. The easiest, but less flexible, way is by adding a *Custom Resource Definition* (*CRD*) to the cluster. The second way, which is more flexible, is the use of *Aggregated APIs (AA)*, which requires a new API server to be written and added to the cluster.

Either way of adding a new object to the cluster, as distinct from a built-in resource, is called a *Custom Resource*.

If you are using RBAC for authorization, you probably will need to grant access to the new CRD resource and controller. If using an *Aggregated API*, you can use the same or a different authentication process.

Custom Resource Definitions

As we have already learnt, the decoupled nature of Kubernetes depends on a collection of watcher loops, or controllers, interrogating the k**ube-apiserver** to determine if a particular configuration is true. If the current state does not match the declared state, the controller makes API calls to modify the state until they do match. If you add a new API object and controller, you can use the existing **kube-apiserver** to monitor and control the object. The addition of a *Custom Resource Definition* will be added to the cluster API path, currently under `apiextensions.k8s.io/v1beta1`.

While this is the easiest way to add a new object to the cluster, it may not be flexible enough for your needs. Only the existing API functionality can be used. Objects must respond to REST requests and have their configuration state validated and stored in the same manner as built-in objects. They would also need to exist with the protection rules of built-in objects.

A CRD allows the resource to be deployed in a namespace or be available in the entire cluster. The YAML file sets this with the `scope:` parameter, which can be set to `Namespaced` or `Cluster`.

Prior to v1.8, there was a resource type called *ThirdPartyResource* (*TPR*). This has been deprecated and is no longer available. All resources will need to be rebuilt as CRD. After upgrading, existing TPRs will need to be removed and replaced by CRDs such that the API URL points to functional objects.

Configuration Example

```
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  name: backups.stable.linux.com
spec:
  group: stable.linux.com
  version: v1
  scope: Namespaced
  names:
    plural: backups
    singular: backup
    shortNames:
    - bks
    kind: BackUp
```

apiVersion: Should match the current level of stability, currently `apiextensions.k8s.io/v1beta1.`

kind: CustomResourceDefinition The object type being inserted by the **kube-apiserver**.

name: backups.stable.linux.com The name must match the `spec` field declared later. The syntax must be `<plural name>.<group>`.

group: stable.linux.com The group name will become part of the REST API under `/apis/<group>/<version>` or `/apis/ stable/v1.` in this case with the version set to **v1**.

scope: Determines if the object exists in a single namespace or is cluster-wide.

plural: Defines the last part of the API URL, such as `apis/stable/v1/backups`.

singular and **shortNames** represent the name with displayed and make CLI usage easier.

New Object Configuration

```
apiVersion: "stable.linux.com/v1"
kind: BackUp
metadata:
  name: a-backup-object
spec:
  timeSpec: "* * * * */5"
  image: linux-backup-image
  replicas: 5
```

Note that the `apiVersion` and `kind` match the CRD we created in a previous step. The `spec` parameters depend on the controller.

The object will be evaluated by the controller. If the syntax, such as `timeSpec`, does not match the expected value, you will receive and error, should validation be configured. Without validation, only the existence of the variable is checked, not its details.

Optional Hooks

Just as with built-in objects, you can use an asynchronous pre-delete hook known as a `Finalizer`. If an API `delete` request is received, the object metadata field `metadata.deletionTimestamp` is updated. The controller then triggers whichever finalizer has been configured. When the finalizer completes, it is removed from the list. The controller continues to complete and remove finalizers until the string is empty. Then, the object itself is deleted.

Finalizer:

```
metadata:
  finalizers:
  - finalizer.stable.linux.com
```

Validation:

```
validation:
    openAPIV3Schema:
      properties:
        spec:
          properties:
            timeSpec:
              type: string
              pattern: '^(\d+|\*)(/\d+)?(\s+(\d+|\*)(/\d+)?){4}$'
            replicas:
              type: integer
              minimum: 1
              maximum: 10
```

Kubernetes Federations

Cluster Federation

Federation is the addition of a new, top-level API endpoint, or control plane, which accepts API calls and passes the calls to member API servers. The control plane communicates with each member control plane, allowing applications to be deployed and move to another member easily. The control plane runs inside of a Pod on a chosen host cluster, and stores cluster information via a PVC in a new 10Gi **etcd** database.

The federated control plane, aware of each of the cluster's resources, can redeploy them to a new cluster upon failure. This provides a higher level of availability, but still depends on the durability and HA of the underlying physical equipment and infrastructure.

Administrative movement, as distinct from failure-driven, can be done by changing the weight of an application. The ability to easily move resources allows Pods and their data to be located closer to the client to minimize latency. Moving data from one cluster, such as a public provider like GCE, to private hosting like OpenStack is also easy.

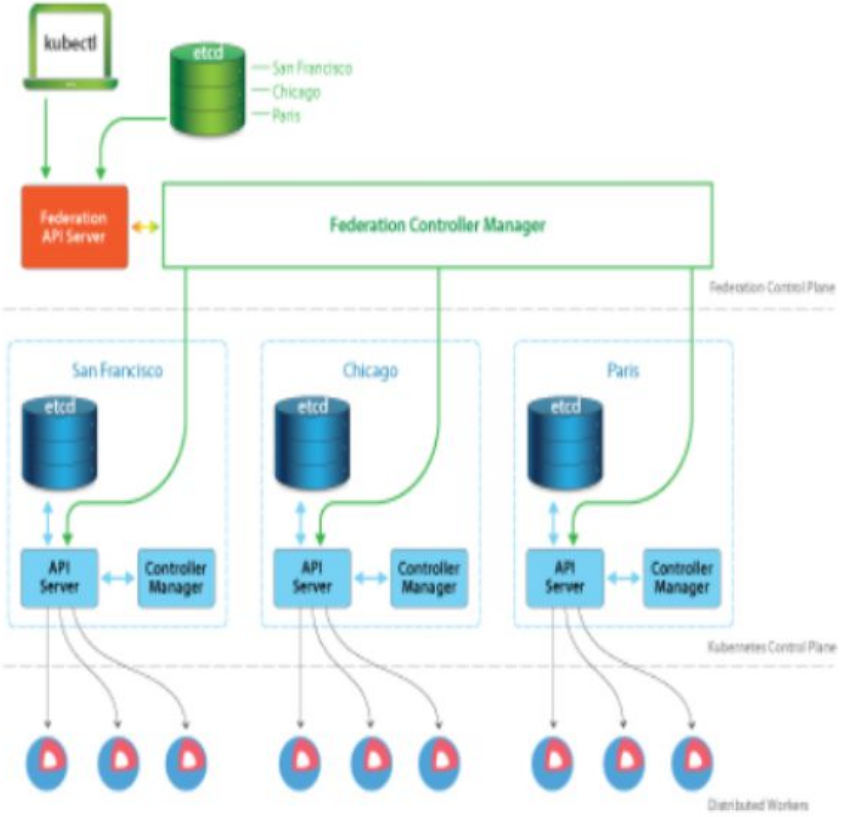
In addition to the scalability and higher availability use, you can deploy federated resources, such as *Federated Services* and *Federated Ingress*, among others. This would allow client use of a single IP, but enjoy traffic routed via the shortest (latency) path to a cluster with available resources automatically.

Using the federation is not without drawbacks. There will be an increase in inter-cluster communication and data, using more bandwidth. Administration of a federated cluster adds another level of complexity and consideration. Also, large federated clusters are new, so code maturity can be a consideration.

Version 1 Federated Control Plane

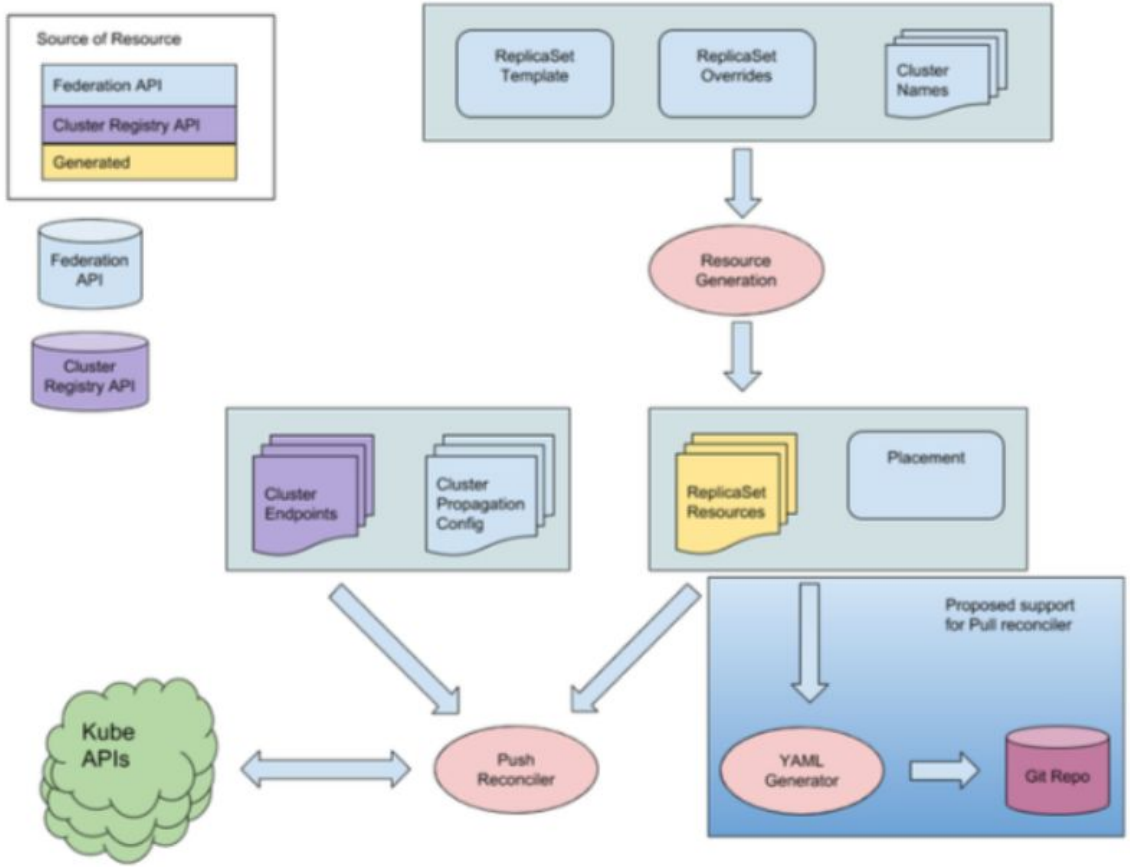
The graphic on this page gives you a high-level architectural view of a **Kubernetes** federation version 1. You can see three stand-alone clusters, each with their own **kube-apiserver**, scheduler, controller manager, and **etcd**-based storage. The `federation-controller-manager` connects to each local API server, both sending API calls, as well as regular health interrogation.

A new API server with its dedicated **etcd**-based storage presents a cluster endpoint for this federated cluster. Users creating resources on this federated endpoint will see them being replicated among the three individual clusters. Every 40 seconds, by default, the nodes `ClusterController` will sync `ClusterStatus` to ensure the health of the cluster. If a member node does not respond properly, the manager control-loop will redeploy resources until the expected deployment state has been met.



Federation V1 Layout

Version 2 Federated Control Plane



Switching Between Clusters

Without a federation, you can use multiple clusters in different zones. But you will need to manually switch endpoint targets for **kubectl**. This has none of the scalability or HA features, but allows for quickly changing between clusters. The target and other parameters can be passed to the shell for one-time use, as well.

```
$ kubectl config use-context sanfrancisco
$ kubectl get nodes
$ kubectl config use-context chicago
$ kubectl get nodes
$ kubectl --context=paris get nodes
```

Each context and authentication keys would be in your **kubectl** configuration file, usually located at `$HOME/.kube/config`. It contains a context for each cluster. A context is a cluster, a user, and the authentication information. There are several types of authentication possible, with SSL/TLS being the more secure.

Building a Federation With Kubefed

The **kubefed** tool entered beta stability since v1.6 and remains the current suggested method.

The `kubefed` command is included with the client tarball called `kubernetes-client-linux-amd64.tar.gz`, which is architecture and release-specific. Other releases can be found on GitHub. Currently, only stable releases include the **kubefed** binary. Once the tarball has been downloaded and expanded, you will need to move the binary and change the access mode.

A federation must begin with the context of a host cluster. These values can be found in the `~/.kube/config` file and viewed via the `kubectl config view` command.

The creation of the control plane interacts with the cluster, local storage, and DNS. As a result, the details passed to the `kubefed init fellowship` command will be different if the cluster is locally hosted or running on a cloud provider. While possible to run without persistent storage, it would be much better to ensure a 10Gi persistent volume is available for use. By default, a `pvc` will be made looking for a 10Gi `pv`. A command on the local equipment may look like this:

```
kubefed init fellowship \
    --host-cluster-context=rivendell \
    --dns-provider="google-clouddns" \
    --dns-zone-name="example.com." \
    --api-server-service-type="NodePort" \
    --api-server-advertise-address="10.0.10.20"
```

Building a Federation With Kubefed

The `kubefed join` command can then be used to add clusters to the federation. Particular context and credentials may need to be passed during the join:

```
kubefed join gondor \
    --host-cluster-context=rivendell
    --cluster-context=gondor_needs-no_king
    --secret-name=11kingdom
```

Using Federated Resources

Most of the resources that you can use in a typical cluster can be deployed in the same manner with a federation. While some resources are considered stable, others, like deployments, are still being developed. Others, like *Federated Ingress*, only run on GCE.

The cluster will try to equally spread replicas across clusters if the resource uses replicas. For example, if you have 10 replicas and five clusters, there would be two replicas per cluster. For other features, like rolling upgrades, the federated API server passes along the request for upgrade, but not the specifics. Each cluster would use local parameters, like `MaxUnavailable`, while updating the local replicas.

Each of the Kubernetes resources (cluster, ConfigMap, DaemonSet, Deployment, events, Horizontal Pod Autoscalers or HPA, Ingress, Jobs, Namespaces, ReplicaSets, Secrets) may have slight differences once used with a federation. The primary difference is the use of the federated server API as the target.

Some resources, like *namespaces*, when created with the federated server, will be created on each cluster. But, when removed, they are only removed from the federated API server. Each cluster retains the namespaces and all objects created within. An administrator would need to change the context to each cluster and delete the *namespace* manually to fully remove it, as well as the resources within.

As with much of Kubernetes, there is much development. You can look up the details of each resource and how they works with Federation for the release in use.

Balancing ReplicaSets

In addition to replicas being spread by default, you can re-balance the replicas, scheduling more pods on one cluster, and less on another. This is currently done via an annotation in the manifest of a *ReplicaSet*. At the moment, *Deployments* do not have a declared process for this, but it is expected to be similar.

By adjusting the weights, Kubernetes will re-balance the pods across the federated cluster. You can start with even weights, and then use `kubectl apply` to update the values.

```
annotations:
    federation.kubernetes.io/replica-set-preferences: |
      {
          "rebalance": true,
          "clusters": {
            "new-york": {
                 "minReplicas": 0,
                  "maxReplicas": 20,
                  "weight": 10
            },
            "berlin": {
                "minReplicas": 1,
                 "maxReplicas": 200,
                 "weight": 20
               }
           }
      }
```

In the example above, the `berlin` cluster would be more likely to get replicas, with the weights being relative values. If you set a value of zero weight, all the replicas would run on other nodes.

Security

Overview

Security is a big and complex topic, especially in a distributed system like Kubernetes. Thus, we are just going to cover some of the concepts that deal with security in the context of Kubernetes.

Then, we are going to focus on the authentication aspect of the API server and we will dive into authorization, looking at things like ABAC and RBAC, which is now the default configuration when you bootstrap a Kubernetes cluster with **kubeadm**.

We are going to look at the `admission control` system, which lets you look at and possibly modify the requests that are coming in, and do a final deny or accept on those requests.

Following that, we're going to look at a few other concepts, including how you can secure your Pods more tightly using security contexts and pod security policies, which are full-fledged API objects in Kubernetes.

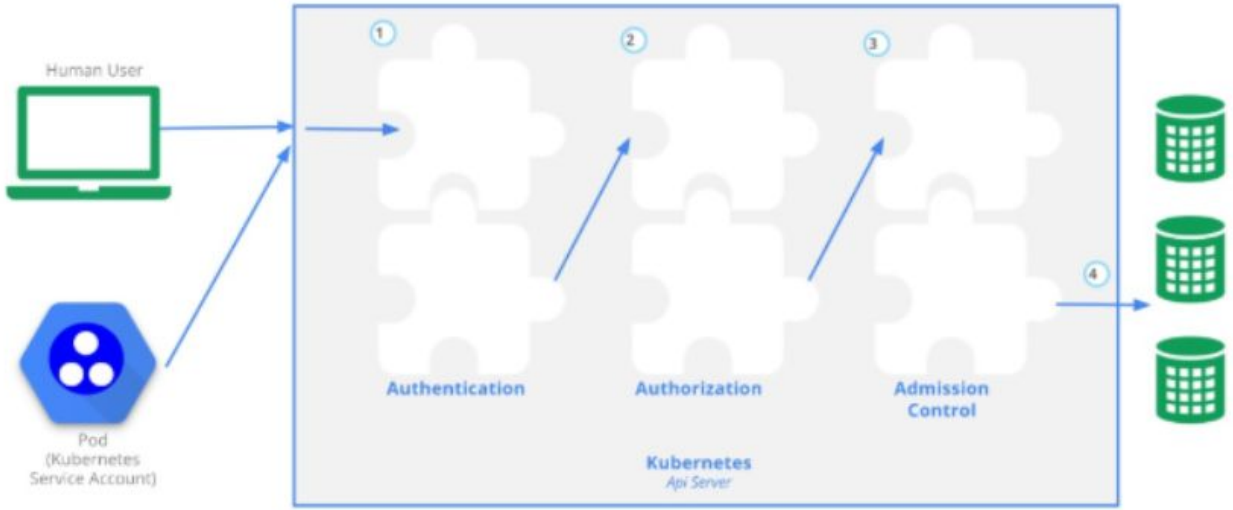
Finally, we will look at network policies. By default, we tend not to turn on network policies, which let any traffic flow through all of our pods, in all the different namespaces. Using network policies, we can actually define Ingress rules so that we can restrict the Ingress traffic between the different namespaces. The network tool in use, such as **Flannel** or **Calico** will determine if a network policy can be implemented. As Kubernetes becomes more mature, this will become a strongly suggested configuration.

Accessing The API

To perform any action in a **Kubernetes** cluster, you need to access the API and go through three main steps:

- Authentication
- Authorization (ABAC or RBAC)
- Admission Control.

These steps are described in more detail in the official documentation about controlling access to the API and illustrated by the picture below:



Accessing The API

Once a request reaches the API server securely, it will first go through any authentication module that has been configured. The request can be rejected if authentication fails or it gets authenticated and passed to the authorization step.

At the authorization step, the request will be checked against existing policies. It will be authorized if the user has the permissions to perform the requested actions. Then, the requests will go through the last step of admission. In general, admission controllers will check the actual content of the objects being created and validate them before admitting the request.

In addition to these steps, the requests reaching the API server over the network are encrypted using TLS. This needs to be properly configured using SSL certificates. If you use **kubeadm**, this configuration is done for you; otherwise, follow Kelsey Hightower's guide *Kubernetes The Hard Way*, or the API server configuration options.

Authentication

There are three main points to remember with authentication in **Kubernetes**:

- In its straightforward form, authentication is done with certificates, tokens or basic authentication (i.e. username and password).
- Users are not created by the API, but should be managed by an external system.
- System accounts are used by processes to access the API.

There are two more advanced authentication mechanisms: **Webhooks** which can be used to verify bearer tokens, and connection with an external **OpenID** provider.

The type of authentication used is defined in the **`kube-apiserver`** startup options. Below are four examples of a subset of configuration options that would need to be set depending on what choice of authentication mechanism you choose:

```
--basic-auth-file
```

```
--oidc-issuer-url
```

```
--token-auth-file
```

```
--authorization-webhook-config-file
```

One or more Authenticator Modules are used: x509 Client Certs; static token, bearer or bootstrap token; static password file; service account and OpenID connect tokens. Each is tried until successful, and the order is not guaranteed. Anonymous access can also be enabled, otherwise you will get a 401 response. Users are not created by the API, and should be managed by an external system.

Authorization

Once a request is authenticated, it needs to be authorized to be able to proceed through the **Kubernetes** system and perform its intended action.

There are three main authorization modes and two global Deny/Allow settings. The three main modes are:

- ABAC
- RBAC
- **WebHook**.

They can be configured as **`kube-apiserver`** startup options:

```
--authorization-mode=ABAC
--authorization-mode=RBAC
--authorization-mode=Webhook
--authorization-mode=AlwaysDeny
--authorization-mode=AlwaysAllow
```

The authorization modes implement policies to allow requests. Attributes of the requests are checked against the policies (e.g. user, group, namespace, verb).

ABAC

ABAC stands for **Attribute Based Access Control**. It was the first authorization model in **Kubernetes** that allowed administrators to implement the right policies. Today, **RBAC** is becoming the default authorization mode.

Policies are defined in a JSON file and referenced to by a **`kube-apiserver`** startup option:

```
--authorization-policy-file=my_policy.json
```

For example, the policy file shown below, authorizes user *Bob* to read pods in the namespace **`foobar`**:

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

RBAC

RBAC stands for **Role Based Access Control**.

All resources are modeled API objects in Kubernetes, from Pods to Namespaces. They also belong to `API Groups`, such as `core` and `apps`. These resources allow operations such as Create, Read, Update, and Delete (CRUD), which we have been working with so far. Operations are called `verbs` inside YAML files. Adding to these basic components, we will add more elements of the API, which can then be managed via RBAC.

Rules are operations which can act upon an API group. **Roles** are a group of rules which affect, or scope, a single namespace, whereas **ClusterRoles** have a scope of the entire cluster.

Each operation can act upon one of three subjects, which are `User Accounts` which don't exist as API objects, `Service Accounts`, and `Groups` which are known as `clusterrolebinding` when using **kubectl**.

RBAC is then writing rules to allow or deny operations by users, roles or groups upon resources.

RBAC Process Overview

While RBAC can be complex, the basic flow is to create a certificate for a user. As a user is not an API object of Kubernetes, we are requiring outside authentication, such as OpenSSL certificates. After generating the certificate against the cluster certificate authority, we can set that credential for the user using a **context**.

Roles can then be used to configure an association of `apiGroups`, `resources`, and the `verbs` allowed to them. The user can then be bound to a role limiting what and where they can work in the cluster.

Here is a summary of the RBAC process:

- Determine or create namespace
- Create certificate credentials for user
- Set the credentials for the user to the namespace using a context
- Create a role for the expected task set
- Bind the user to the role
- Verify the user has limited access.

Admission Controller

The last step in letting an API request into **Kubernetes** is admission control.

Admission controllers are pieces of software that can access the content of the objects being created by the requests. They can modify the content or validate it, and potentially deny the request.

Admission controllers are needed for certain features to work properly. Controllers have been added as Kubernetes matured. As of the v1.12 release, the **`kube-apiserver`** uses a compiled set of controllers. Instead of passing a list, we can enable or disable particular controllers. If you want to use a controller that is not available by default, you would need to download from source and compile.

The first controller is `Initializers` which will allow the dynamic modification of the API request, providing great flexibility. Each admission controller functionality is explained in the documentation. For example, the `ResourceQuota` controller will ensure that the object created does not violate any of the existing quotas.

Security Contexts

Pods and containers within pods can be given specific security constraints to limit what processes running in containers can do. For example, the UID of the process, the **Linux** capabilities, and the filesystem group can be limited.

This security limitation is called a *security context*. It can be defined for the entire pod or per container, and is represented as additional sections in the resources manifests. The notable difference is that **Linux** capabilities are set at the container level.

For example, if you want to enforce a policy that containers cannot run their process as the root user, you can add a pod security context like the one below:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  securityContext:
    runAsNonRoot: true
  containers:
  - image: nginx
    name: nginx
```

Then, when you create this pod, you will see a warning that the container is trying to run as root and that it is not allowed. Hence, the Pod will never run:

```
$ kubectl get pods
NAME    READY   STATUS                                                  RESTARTS   AGE
nginx   0/1     container has runAsNonRoot and image will run as root   0          10s
```

Pod Security Policies

To automate the enforcement of security contexts, you can define PodSecurityPolicies (PSP). A PSP is defined via a standard Kubernetes manifest following the PSP API schema. An example is presented on the next page.

These policies are cluster-level rules that govern what a pod can do, what they can access, what user they run as, etc.

For instance, if you do not want any of the containers in your cluster to run as the root user, you can define a PSP to that effect. You can also prevent containers from being privileged or use the host network namespace, or the host PID namespace.

Pod Security Policies

You can see an example of a PSP below:

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: restricted
spec:
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: MustRunAsNonRoot
  fsGroup:
    rule: RunAsAny
```

For Pod Security Policies to be enabled, you need to configure the admission controller of the **controller-manager** to contain **`PodSecurityPolicy`**. These policies make even more sense when coupled with the **RBAC** configuration in your cluster. This will allow you to finely tune what your users are allowed to run and what capabilities and low level privileges their containers will have.

Network Security Policies

By default, all pods can reach each other; all ingress and egress traffic is allowed. This has been a high-level networking requirement in Kubernetes. However, network isolation can be configured and traffic to pods can be blocked. In newer versions of Kubernetes, egress traffic can also be blocked. This is done by configuring a `NetworkPolicy`. As all traffic is allowed, you may want to implement a policy that drops all traffic, then, other policies which allow desired ingress and egress traffic.

The `spec` of the policy can narrow down the effect to a particular namespace, which can be handy. Further settings include a `podSelector`, or label, to narrow down which Pods are affected. Further ingress and egress settings declare traffic to and from IP addresses and ports.

Not all network providers support the `NetworkPolicies` kind. A non-exhaustive list of providers with support includes **Calico**, **Romana**, **Cilium**, **Kube-router**, and **WeaveNet**.

In previous versions of Kubernetes, there was a requirement to annotate a namespace as part of network isolation, specifically the `net.beta.kubernetes.io/network-policy= value`. Some network plugins may still require this setting.

Network Security Policy Example

The use of policies has become stable, noted with the `v1 apiVersion`. The example below narrows down the policy to affect the default namespace.

Only Pods with the label of `role: db` will be affected by this policy, and the policy has both Ingress and Egress settings.

The `ingress` setting includes a 172.17 network, with a smaller range of 172.17.1.0 IPs being excluded from this traffic.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: ingress-egress-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
<continued_on_next_page>
```

Network Security Policy Example

```
  - namespaceSelector:
      matchLabels:
        project: myproject
  - podSelector:
      matchLabels:
        role: frontend
  ports:
  - protocol: TCP
    port: 6379
egress:
- to:
  - ipBlock:
      cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

These rules change the namespace for the following settings to be labeled `project: myproject`. The affected Pods also would need to match the label `role: frontend`. Finally, TCP traffic on port 6379 would be allowed from these Pods.

The `egress` rules have the `to` settings, in this case the 10.0.0.0/24 range TCP traffic to port 5978.

The use of empty ingress or egress rules denies all type of traffic for the included Pods, though this is not suggested. Use another dedicated `NetworkPolicy` instead.

Note that there can also be complex `matchExpressions` statements in the spec, but this may change as `NetworkPolicy` matures.

```
podSelector:
  matchExpressions:
```

Default Policy Example

The empty braces will match all Pods not selected by other `NetworkPolicy` and will not allow ingress traffic. Egress traffic would be unaffected by this policy.

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

With the potential for complex ingress and egress rules, it may be helpful to create multiple objects which include simple isolation rules and use easy to understand names and labels.

Default Policy Example

Some network plugins, such as **WeaveNet**, may require annotation of the Namespace. The following shows the setting of a `DefaultDeny` for the `myns` namespace:

```
kind: Namespace
apiVersion: v1
metadata:
  name: myns
  annotations:
    net.beta.kubernetes.io/network-policy: |
     {
        "ingress": {
          "isolation": "DefaultDeny"
        }
     }
```

THANK YOU!