k8s-deep-dive

Contents

    * Packaging applications
    * Declarative model
    * Pods
    * Deployments
    * Services
- Clusters
- Deployments
    * Theory
    * `ReplicaSets`
    * Pods
    * Rollouts
    * Rollbacks
    * Labels
    * Skeleton
- Services
    * Theory
    * Labels
    * Endpoints
    * Types
    * Registration
    * Discovery
    * Network magic
    * Network Traffic
    * Namespaces
    * Skeleton
- Ingress
    * Theory
    * Network traffic
    * Cluster Traffic
    * Skeleton
- Storage
    * The big picture
    * The Storage Providers
    * The CSI - Container Storage Interface
    * The Persistent Volume Subsystem
    * `StorageClass`
    * Application
- `ConfigMaps` & Secrets
    * The big picture
    * `ConfigMaps`
    * Secrets
- `StatefulSet`
    * Theory
    * Naming
    * Creation
    * Deleting
    * Volumes
    * Handling Failures
    * Services
    * Network traffic
    * Skeleton

- Security
  - Theory
  - Authentication
  - Authentication
  - Authorization
  - Admission control
  - General Summary
- Kubernetes API
  - Theory
  - Serialization
  - The API server
  - The API
  - The core
  - Named Groups
- Thread modeling
  - Spoofing
  - Tampering
  - Repudiation
  - Information Disclosure
  - Denial of Service
  - Elevation of privilege
- Real world security
  - CI/CD pipeline

Introduction

`Kubernetes` is an application `orchestrator`, for the most part it orchestrates containerized cloud native micro services. An orchestrator is a system that deploys and manages apps, it can deploy your app and dynamically respond to changes, for example k8s, can:

1. Deploy you app
2. Scale it up and down dynamically based on demand
3. Self heal when things break
4. Perform zero downtime rolling updates and rollbacks
5. Many, many more things

`What really is containerised app` - it is an app that runs in a container, before we had containers, apps ran on physical servers or virtual machines, containers are just the next iteration of how we package and run apps, as such they are faster more lightweight and more suited to modern business requirements than servers and virtual machines

`What is a cloud native app` - it is one that is designed to meet the cloud like demands of auto scaling self healing rolling updates, rollbacks and more, it is important to be clear that cloud native apps are not apps that will only run in the public cloud, yes they absolutely can run on a public clouds, but they can also run anywhere that you have k8s even your on premise datacenter.

`What are microservice apps` - is built from lots of independent small specialised parts that work together to form a meaningful app. For example you might have an e-commerce app that comprises all of the following small components

1. Web front end
2. Catalog service
3. Shopping cart

4. Authentication service
5. Logging service
6. Persistent store

Each of these individual services is called a micro service, typically each is coded and owned by a different team, each can have its own release cycle and can be scaled independently, for example you can patch and scale the logging micro service without affecting any of the others. Building apps this way is vital for cloud native features, For the most part, each microservice runs as a container, assuming that e-commerce app with the 6 microservice there would be one or more web front end containers one or more catalog containers one or more shipping cart containers etc. With all of this in mind

`Kubernetes deploys and manages (orchestrates)apps that are packaged and run as containers (containerized)and that are built in ways (cloud native microservice)that allows them to scale, self heal and be updated in line with modern cloud like requirements.`

History

Since amazon brought the Amazon Web Services, the world changed, since then everyone is playing catch-up. One of the companies trying to catch up was Google. It has its own very good cloud and needs a way to abstract the value of AWS and make it easier for potential customers to get off AWS and into their cloud. Google also has a lot of experience working with containers at scale, for example huge google apps such as Search and Gmail have been running at extreme scale on containers for a lot of years, since way before Docker brought us easy to use containers. To orchestrate and manage these containerised apps, Google had a couple of in-house proprietary systems called Borg and Omega. Well Google took the lessons learned from these systems, and created a new platform called Kubernetes, and donated it to the newly formed `Cloud Native Computing Foundation (CNCF)` in 2014, as an open source project. Kubernetes enables two things Google and the rest of the industry needs

1. It abstracts underlying infrastructure such as `AWS`
2. It makes it easy to move apps on and off clouds

Since its introduction in 2014, Kubernetes has become the most important cloud native technology on the planet. Like many of the modern cloud native projects, it's written in Go, it is built in the open on GitHub it is actively discussed on the IRC channels, you can follow it everywhere on social media, there are also regular conferences and regular meetups

Meaning

The name Kubernetes comes from the Greek word meaning Helmsman - the person steers the seafaring ship. This theme is reflected in the logo which is the wheel (helm control) of a sea faring ship. You will often see it shortened to k8s - pronounced `kate`. The number 8 replaces the 8 characters between the K and the S in the name, and that is people sometimes joke that Kubernetes has girlfriend named Kate

Kubernetes and Docker

Kubernetes and Docker are two complementary technologies, Docker has tools that build and package apps as container images. It can also run containers, Kubernetes can't do either of those things, Instead, Kubernetes operates at a higher level providing orchestration services such as self-healing, scaling and updating. It is common practice to use Docker for build time tasks such as packaging apps as containers, but then use a combination of Kubernetes and Docker to run them. In this model, Kubernetes preforms the high level orchestration tasks, while Docker performs the low level tasks such as starting and stopping containers.

Assume you have a Kubernetes cluster with 10 nodes, to run your production app. Behind the scenes each

cluster node is running Docker as its container runtime. This means Docker is the low-level technology that starts and stops the containerised apps. Kubernetes is the higher level technology that looks after the bigger picture, it tells docker how to do that, also it is deciding which nodes to run containers on, deciding when to scale up or down, and execute updates. Docker is not the only container runtime that Kubernetes supports, it also does support `gVisor`, `containerd` and `kata`. Kubernetes has features which abstract the container runtime and make it interchangeable

1. The container runtime interface (CRI) - is an abstraction layer that standardizes the way 3rd party container runtimes work with Kubernetes
2. Runtime Classes allows you to create different classes of runtimes. For example the `gVisor` or `Kata` Containers runtimes might provide better workload isolation than the Docker and `containerd` runtimes

Kubernetes and Swarm

In 2016 and 2017 we had the orchestrator wars, where `Docker Swarm`, `Mesosphere DCOS`, and `Kubernetes` completed to become the de-facto container orchestrator. To cut a long story short, Kubernetes WON.

There is a good chance you will hear people talk about how Kubernetes relates to Google's Borg and Omega systems, as previously mentioned, Google has been running containers at scale for a long time - apparently crunching through billions of containers a week. So yes, Google has been running things like `Search`, `Gmail`, and `GFS` on lots of containers for a very longs time. Orchestrating these containerised apps was the job of a couple of in-house technologies called Borg and Omega. So it is not a huge stretch to make the connection with Kubernetes - all three are in the game or orchestration or containers at scale, and they are all related to Google.

```
Kubernetes is not an opened source version of Borg and Omega, it shares common traits and
technologies, common DNA, if you wish, but the Borg and Omega are still proprietary closed
 source projects
```

They are all separate but all three are related, in fact, some of the people who built Borg and Omega were and still are involved with Kubernetes. So although Kubernetes was built from scratch, it leverages much of what was learned at Google with Omega and Borg

Theory

At the highest level, Kubernetes is two things - A cluster to run apps on & an orchestrator of cloud native microservice apps.

Kubernetes as an OS Kubernetes has emerged as the de-facto platform for deploying and managing cloud native apps, in many ways it is like an operating system for the cloud. In the same way that Linux abstracts the hardware differences between server platforms, Kubernetes abstracts the differences between the different private and public clouds. Net result is that as long as you are running Kubernetes, it does not matter if the underlying systems are on premises, in your own datacenter, edge devices or in the public cloud or domain.

Kubernetes as a cluster Kubernetes is like any other cluster - a bunch of machines to host apps on. We call these machines nodes, and they can be physical servers, virtual machines, cloud instances, Raspberry Pis, and more. A Kubernetes cluster is made of a `control plane` and `nodes`. This control plane exposes the API, has scheduler for assigning work, and records the state of of the cluster and apps in a persistent store. Nodes are where user apps run. It can be useful to think of the `control plane` as the brains of the cluster and the nodes as the muscle. In this analogy the `control plane` is the brains because it implements the clever features such as scheduling, auto-scaling and zero-downtime rolling updates.

Kubernetes as orchestrator Orchestrator is just a fancy word for a system that takes care of deploying and managing apps. If we take a quick analogy from the real world, a football team is made of individuals. Every individual is different and each has a different role to play in the team - some defend some attack some are great at passing some tackle some shoot… Along comes the coach and she or he gives everyone a position and organizes them into a team with a purpose. The coach also makes sure that the team keeps its formation sticks to the game-plan and deals with any injuries and other changes in the circumstances. Well microservices apps on Kubernetes are the same.

You start out with lots of individual specialised microservices. Some serve web pages, some do authentication some perform searches, other persist data. Kubernetes comes along - like the coach, organizes everything into a useful app and keeps things running smoothly. It even responds to events and other changes in the circumstances - auto-scaling, updating, rolling release etc.

When we start out with an app, package it as a container then give it to the cluster - Kubernetes. The cluster is made up of one or more control plane nodes and a bunch of worker nodes. As already stated, control plane nodes implement the cluster intelligence, worker nodes are where user apps run.

Control plane

As previously mentioned a Kubernetes cluster is made of control plane nodes and worker nodes. These are Linux hosts that can be virtual machines, bare metal servers in your datacenter or basement, instances in a private or public cloud. You can even run Kubernetes on ARM and IoT devices

A Kubernetes control plane node is a server running collection on system services that make up the control plane of the cluster. Sometimes we call those Masters, Heads or Head nodes. The simplest setups run a single control plane node. However this is only suitable for labs and test environments, for production environments multiple control plane nodes configured for high availability is vital. Also considered a good practice not to run user apps on control plane nodes. This frees them up to concentrate entirely on managing the cluster. These are some of the core components

1. **etcd**: A distributed key-value store that stores all cluster data, i.e called the cluster store.
2. **kube-apiserver**: The front-end for the Kubernetes control plane, exposing the Kubernetes API.
3. **kube-scheduler**: Assigns workloads (Pods) to nodes based on resource availability and constraints.
4. **kube-controller-manager**: Runs controllers that regulate the state of the cluster (e.g., Node Controller, Replication Controller).

```
Kubernetes is self bootstrapping - meaning that components such as the API server, just
like other system components part of kubernetes, are backed by kubernetes native objets -
 like pods, deployments and services, located in the kube-system namespace, just the way
 an actual user would deploy their own images and containers, in services, deployments,
 and so on, pretty much the entire set of system kubernetes components, are represented
with the same types of objects like services, deployments, endpoints, replicasets and so
on. The kubernetes enviornment can be self inspected, meaning that we can easily see the
different components, part of the contorl plane, which are deployed as native kubernetes
objects, with the kubectl command, we just have to look in the kube-system namespace for
them, not in the default namespace. Other system components like ingress controllers might
 be deployed in different than the kube-system namespace, but the general rule mentioned
above still holds, kubernetes is in a way self-bootstrapping
```

```
$ kubectl get pods -n kube-system
NAME                       READY STATUS  RESTARTS AGE
coredns-668d6bf9bc-mtpmm 1/1   Running 1        (4h12m ago) 4h21m
etcd                       1/1   Running 1        (4h12m ago) 4h21m
kube-apiserver             1/1   Running 1        (4h12m ago) 4h21m
```

```
kube-controller-manager  1/1   Running 1        (4h12m ago) 4h21m
kube-proxy-z9gf9         1/1   Running 1        (4h12m ago) 4h21m
kube-scheduler           1/1   Running 1        (4h12m ago) 4h21m
storage-provisioner      1/1   Running 3        (3h7m  ago) 4h21m
```

Above we can see a very simple example of what the control plane consists of, as we can see these are the actual pods, backed by an image, which are responsible for the internal mechanisms and workings of the kubernetes orchestration, we can say `kubernetes is run on itself, it is a self-bootstraping system`. Each of those components, will be investigated in depth, further down in this document, but for now this is the overarching way we can look at the kubernetes architecture.

The API Server The API server is the Grand Central of Kubernetes. All communication, between all components must go through the API server. It is important to understand that internal system components as well as external user components all communicate via the API server - all roads lead to the API server

It exposes a RESTful API that you POST YAML configuration files to over HTTPS. These YAML files which we sometimes call manifests describe the desired state of an app. This desired state includes things like which container image to use, which ports to expose and how many Pod replicas to run. All requests to API server are subject to authentication and authorization checks. Once these are done, the configuration in the YAML file is validated, persisted to the cluster store, and work is scheduled to the server

As mentioned already the api server is nothing more than a pod backed by an image, and a service objects. Meaning that we can simply see those by doing - `kubectl get pods -n kube-system`. These will show you all pods part of the system namespace, and in there one will immediately notice, a pod called - `kube-apiserver`.

The Cluster Store The cluster store is the only stateful part of the control plane and persistently stores the entire configuration and state of the cluster. As such it is vital components of every Kubernetes cluster - no cluster store, no cluster. The cluster store is currently based on `etcd`, a popular distributed database. As it is the single source of truth for a cluster, you should run between 3-5 replicas of the `etcd` service for high-availability and you should provide adequate ways to recover when things go bad. A default installation of Kubernetes installs a replica of the cluster store on every control plane node and automatically configures the high availability (HA)

On the topic of availability, `etcd` prefers consistency over availability. This means it does not tolerate split brains and will halt updates to the cluster in order to maintain consistency. However, if this happens user apps should continue to work, you just won't be able to update the cluster configuration.

As with all distributed databases, consistency of writes to the database is vital. For example multiple writes to the same value originating from different places needs to be handled. `etcd` uses the popular RAFT consensus algorithm to accomplish this.

The Controller Manager The controller manager implements all the background controllers that monitor cluster components and respond to events. Architecturally, it is a controller of controllers, meaning it spawns all the independent controllers and monitors them. Some of the controllers include the `Deployment` controller, the `StatefulSet` controller and the `ReplicaSet` controller. Each one is responsible for a small subset of cluster intelligence and runs as a background watch loop constantly watching the API Server for changes.

`The aim of the game is to ensure the observed state of the cluster matches the desired state. The logic implemented by each controller is as follows, and is at the heart of Kubernetes and declarative design patterns`

1. Obtain desired state

2. Observe current state
3. Determine differences
4. Reconcile differences

Each controller is also extremely specialized and only interested in its own little corner of the Kubernetes cluster. No attempts is made to over complicate design by implementing awareness of other parts of the system, each controller takes care of its own business and leaves everything else alone. This is key to the distributed design of Kubernetes and adheres to the Unix philosophy. Controllers are control loops that watch the state of the cluster and make changes to bring the current state closer to the desired state. Some core controller components

`Controllers run on a watch-loop meaning that these are pods or processes in the end of the day, which watch the api server, for new changes, in the desired state, and then make sure to match that in the actual state in the kubernetes environment`

- **Node Controller**: Manages node lifecycle.
- **Service Controller**: Ensures that traffic is routed to the correct pods on the fly.
- **Deployment Controller**: Manages updates to Pods and `ReplicaSets` objects
- **Ingress Controller** is a special type of controller that handles Ingress resources. It is not part of the control plane itself but is instead a user-deployed component that runs as a Pod in the cluster. It watches for Ingress resources and configures external load balancers or proxies (e.g., `NGINX`, `Traefik`) to route traffic to the appropriate services.

`K8s objects allow the definition of something called annotations, these are key value pairs/maps which are a way to store metadata or configuration information, which is a general purpose way that is used by different k8s controllers, to configure their own behavior, each object in k8s is controlled and managed by an accompanying controller, this controller is responsible for managing the objects in the control plane, and each controller has specific features which can be enabled using the annotation metadata when defining the object manifest itself, the annotation values are tied to the object, but are parsed and enforced by the controller which manages the specific type of object, this is important to understand, controllers are stateless processors of the stateful k8s environment and objects`

Here is an example of a manifest snippet which shows how annotations are defined in a manifest, each is specific to the target object in this case the target objects are ingress and deployment, do not concern yourselves with these types of objects, the important part to take a note of is that these annotation values are bound to the specific object "instance" and its manifest. The object instances are in this case defined by the `metadata.name`, the name is a special property in k8s which gives the created object a unique name that k8s can use to identify the object uniquely in the k8s environment, it is not only meant to be read by users, but the k8s control plane and controllers as well

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: my-ingress # defines a new name for the ingress object to create
    annotations:
        nginx.ingress.kubernetes.io/rewrite-target: / # Rewrite URL paths
        nginx.ingress.kubernetes.io/ssl-redirect: "true" # Redirect HTTP to
           HTTPS
---
apiVersion: apps/v1
```

```
kind: Deployment
metadata:
    name: my-deployment # defines a new name for the deployment object to
       create
    annotations:
        my-controller/scale-to: "5" # Custom annotation for scaling
        sidecar.istio.io/inject: "true" # Enable istio sidecar injection
        autoscaling.alpha.kubernetes.io/metrics: '{"type":"Resource","
           resource":{"name":"cpu","targetAverageUtilization":50}}' # metrics
            configuration
```

The Scheduler At a high level, the scheduler watches the API server for new work tasks and assigns them to appropriate healthy worker nodes. Behind the scenes, it implements complex logic that filters out nodes incapable of running tasks and the ranks the nodes that are capable. The ranking system is complex but the node with the highest ranking score is selected to run the task.

When identifying nodes capable of running a task, the scheduler performs various predicate checks. These include is the node tainted, are there any affinity or anti affinity rules, is the required network port available on the node does it have sufficient available resources etc. Any node incapable of running the task is ignored and those remaining are ranked according to things such as does it already have the required image how much free resources does it have, how many tasks is it currently running. Each is worth points and the node with the most points is selected to run the task. If the scheduler does not find a suitable node, the task is not schedule and gets marked as pending. The scheduler is not responsible for running tasks just picking the nodes to run them. A task is normally a Pod/container.

The Node The kubelet is main Kubelet agent and runs on every cluster node. In fact, it is common to use the terms node and kubelet interchangeably. When you join a node to a cluster the process installs the kubelet which is then responsible for registering it with the cluster. This process registers the node's CPU, memory, and storage into the wider cluster pool.

One of the main jobs of the kubelet is to watch the API server for new work tasks. Any time it sees one, it executed the task and maintains a reporting channel back to the control plane. If a kubelet can't run a task, it reports back to the control plane and lets the control plane decide what actions to take. For example if a kubelet can not execute a task, it is not responsible for finding another node to run it on. It simply reports back to the control plane and the control plane decides what to do.

The Proxy The last piece of the node puzzle is the `kube-proxy`. This runs on every node and is responsible for local cluster networking. It ensures each node gets its own unique IP address, and it implements local `iptables` or `IPVS` rules to handle routing and load balancing of traffic on the Pod network. More on all of this later on in other chapters down below.

```
The kube-proxy is a crucial component in the node to node internal cluster network
communication stack, which allows nodes to communicate to other nodes, through the use
of iptables and IPVS, rules, providing load balancing traffic between Nodes and by proxy
, Pods on different Nodes. The kube proxy is not actively participating in the actual
traffic, rather it monitors for new Services and Endpoints and helps setup the state of
 the IPVS or iptable rules on the node, the active part happens in the actual kernel
reading the ipvs rules and doing the IP translation / mapping to the other nodes
```

The Runtime The kubelet needs a container runtime to perform a container related task - things like pulling images and starting and stopping containers. In the early days, Kubernetes had native support for

Docker, More recently it has moved to a plugin model called the `Container Runtime Interface (CRI)`. At a high level, the `CRI` masks the internal machinery of Kubernetes and exposes a clean documented interface for 3rd party container runtimes to plug into. Kubernetes is dropping support for Docker as a container runtime, this is because Docker is bloated and does not support the `CRI` (requires a shim instead). `containerd` is replacing it as the most common container runtime on Kubernetes

Besides the kubelet and proxy, Kubernetes nodes also run other essential management components, one key components if the container runtime interface (CRI) which is responsible for running and managing containers. Popular CRI implementations include containerd (used in OpenShift). Another component is the container network interface (CNI) which handles networking for pods, including IP address allocation and routing, different CNI plugins such as calico, flannel, cilium, and weave-net, enable networking capabilities based on the cluster needs

```
containerd is the container supervisor and runtime logic stripped out from docker engine.
 It was donated to the CNCF by Docker Inc, and has a lot of community support. Other CRI
container runtimes also exist.
```

The DNS As well as the various control plane and node components, every Kubernetes cluster has an internal `DNS` service, that is vital to service discovery. The cluster's `DNS` service has a static IP address that is hard coded into every Pod on the cluster. This ensures every container and Pod can locate it and use it for discovery. Service registration is also automatic. This means apps do not need to be coded with the intelligence to register with Kubernetes service discovery. Cluster `DNS` is based on open source `CoreCNS` project.

Control plane summary `Kubernetes control plane nodes are servers that run the cluster's control plane services`. These services are the brains of the cluster where all the control and scheduling decisions happen. Behind the scenes, these services include the API server, the cluster store, scheduler and the specialised controllers.

```
The API server is the front end into the control plane and all instructions and communication
 pass through it. By default it exposes a RESTful endpoint on port 443.
```

Packaging applications

An app needs to tick a few boxes to run on a Kubernetes cluster. These include:

1. Packages as a container image
2. Wrapped the image as a container instance in a pod
3. Deployed via a declarative config manifest file

It goes like this. You write an application microservice in a language of your choice. Then you build it into a container image and store it in a registry. At this point the app service is containerized. Next you define a Kubernetes Pod to run the containerized app. At the kind of high lever we are at, a Pod is just a wrapper that allows a container to run on a Kubernetes cluster. Once you have defined the pod, you are ready to deploy the app to Kubernetes.

While it is possible to run static Pods like this on a Kubernetes cluster, the preferred model is to deploy all Pods via a higher level controllers. The most common controller is the Deployment. It offers scalability, self healing and rolling updates for stateless apps. You define Deployments in YAML manifest files that specify things how many replicas to deploy and how to perform updates. Once everything is defined in the Deployment YAML file, you can use the Kubernetes command line tools to post it to the API server as the desired state of the app, and Kubernetes will implement it

Declarative model

The declarative model and the concept of desired state are at the very heart of Kubernetes. So it is vital you understand them. In Kubernetes the declarative model works like this.

1. Declare the desired state of an app, microservice in a manifest file
2. Post the desired state to the API server
3. Kubernetes stores it in the cluster store as the app's desired state
4. Kubernetes implements the target desired state in the cluster
5. Controller makes sure the observed state of the app does not vary from the desired state

Manifest files are written in simple YAML and tell Kubernetes what an app should look like. This is called desired state. It includes things such as which image to use, how many replicas to run, which network ports to listen on, and how to perform updates.

Once you have created the manifest you post it to the API server. The easiest way to do this is with the kubectl command line utility. This sends the manifest to the control plane as an HTTPS POST request (on port 443)

Once the request is authenticated and authorized. Kubernetes inspects the manifest, identifies which controller to send it to (i.e. Deployments controller) and records the configuration in the cluster store as part of the overall desired state. Once this is done any required work tasks get scheduled to the cluster nodes where the kubelet co-ordinates the hard work of pulling images starting containers attaching to networks, and starting app processes.

Finally controllers run as background reconciliation loops that constantly monitor the state of things, if the observed state deviates from the desired state, Kubernetes performs the tasks which are necessary to reconcile the differences and bring the observed state back in sync with the desired state

```
It is important to understand that what we have described is the opposite of the traditional
 imperative model. The imperative model is where you write long scripts of platform
specific commands to build and monitor things, not only is the declarative model a lot
simpler than long scripts with lots of imperative commands, it also enables self-healing,
 scaling and lends itself to version control and self-documentation. It does all of this
by telling the cluster how thing should look like. If they start to look different, the
appropriate controller notices the discrepancy and does all the hard work to reconcile the
 situation.
```

Assume you have an app with a desired state that includes 10 replicas of a web front end Pod. If a node running two replicas fails, the observed state will be reduced to 8 replicas but desired state will still be 10. This will be observed by a controller and Kubernetes will schedule two new replicas to bring the total back up to 10. The same thing will happen if you intentionally scale the desired number of replicas up or down. You could even change the image you want to use (this is called a `rollout`). For example if the app is currently using `v2.00` of an image and you update the desired state to specify `v2.01` the relevant controller will notice the difference and go through the process of updating the cluster so all 10 replicas are running the new version.

To be clear. Instead of writing a complex script to step through the entire process of updating every replica to the new version, you simply tell Kubernetes you want the new version and Kubernetes does the hard work for you.

Pods

In the VMware world the atomic unit of scheduling is the virtual machine. In the docker world it is the container, in the Kubernetes world it is the Pod. It is true that Kubernetes runs containerized apps. However Kubernetes demands that every container runs inside a pod.

Pods are objects in the Kubernetes API, so we capitalize the first letter. This adds clarity and the official Kubernetes docs are moving towards this standard. You can think of Pods as specs and rules for running containers, they define all kind of container specific rules and boundaries, which are used when the containers are managed in their lifecycle - started, running, terminated by the container runtime (like containerd)

Pods & Containers The very first thing to understand is that the term Pod comes from a pod of whales - in the English language we call a group of whales a pod of whales. As the Docker logo is a whale, Kubernetes ran with the whale concept and that is why we have Pods. The simplest model is to run a single container in every Pod. This is why we often use the term Pod and container interchangeably. However there are advanced use cases that run multiple containers in a single Pod, Powerful examples of multi container Pods include:

- Service meshes
- Containers with a tightly coupled log scraper
- Web containers supported by a helper container pulling updated content

The point is that a Kubernetes Pod is a construct for running one or more containers . A pod is an object, defined declaratively in the k8s state, they are not physical entities that run on the Nodes, they are used by the kubelet service or daemon to control containers

Pod anatomy At the highest level a Pod is ring fenced environment to run containers. Pods themselves do not actually run apps, apps always run in containers, the Pod is just a sandbox to run one or more containers. Keeping it high level, Pods ring fence an area of the host OS, build a network stack, create a bunch of kernel namespaces and run one or more containers

If you are running multiple containers in a Pod they all share the same Pod environment. This includes the network stack, volumes `IPC` namespace, shared memory and more. As an example this means all containers in the same Pod will share the same IP address. If two containers in the same Pod want to talk to each other, they can use the Pod's `localhost` interface.

Multi container Pods are ideal when you have requirements for tightly coupled containers that may need to share memory and storage. However if you do not need to tightly couple containers, you should put them in their own Pods, and loosely couple them over the network. This keeps things clean by having each Pod dedicated to a single task. However it creates a lot of potentially `un-encrypted` network traffic. One must seriously consider using a service mesh to secure traffic between Pods and app services

Now here is an interesting part of how the pods work internally, even though as already mentioned pods are configuration entities, managed by the kubelet, the kubelet itself, spawns things called pause containers, each pod is associated with these pause containers, which are more like a namespace holders, they are responsible for holding the network namespace and other resources a Pod environment has "promised" to its running containers, this is done `in case all containers managed by a pod die, somehow, or currently a pod has no running containers, for what ever reason`. The statement that a pod is not a container `still holds`, it is just that the kubelet uses auxiliary structures to retain the resources and overall state associated with a pod configuration object. This is mostly an implementation detail, and end users can not interact with the so called `pause containers`

Here is an example: imagine a pod with two containers - web server and logging sidecar container

1. The kubelet starts the pause container first
2. The pause container sets up the shared network and IPC namespaces.
3. The kubelet starts the web server container and joins it to the pause container namespace
4. The kubelet starts the logging container and joins it to the same namespace

Now both containers share the same networking namespaces, they communicate over localhost, and the same IPC namespace (they can use shared memory)

```
Pause containers simply reserve the linux kernel namespaces, which are then used to be
shared between the actual running containers configured for the Pod, all the resources
that are shared between containers inside a pod, are actually bound to the namespace of
the pause container, for that pod, which the kubelet has started
```

Pods as unit of scaling Pods are also the minimum unit of scheduling in Kubernetes If you need to scale an app, you add or remove Pods. You do not scale by adding more containers to existing Pods. Multi-container Pods are only for situations where to different but complimentary containers need to share resources.

```
You never scale an app by adding more of the same app containers to a Pod, multi container
 pods are not a way to scale an app, they are only for co scheduling and co locating
containers that need tight coupling, like a web service and a logging service, or a in
memory data store etc. If you need to scale the app you add more pods, or remove pods,
this is called horizontal scaling
```

Pods atomic operations The deployment of a Pod is an atomic operation. This mean a Pod is only ready for service when all its containers are up and running. The entire Pod either comes up and is put into service or it does not and it fails. A single Pod can only be schedules to a single node - you can not schedule a single Pod across multiple nodes. This is also true of multi container Pods - all containers in the same Pod run on the same node.

Pod lifecycle Pods are mortal. The are created they live and they die. If they die unexpectedly you do not have to bring them back to life. Instead Kubernetes starts a new one in its place. However even though the new Pod looks, smells and feels like the old one, it is not. It is a shiny new Pod with a shiny new ID and IP address.

This has implications on how you design you app. Do not design them to be tightly coupled to particular instance of a Pod. Instead design them so that when Pods fail a totally new one can pop up somewhere else in the cluster and seamlessly take its place

Pod immutability Pods are also immutable this means you do not change them once they are running. Once a Pod is running you never change its configuration. If you need to change or update it, you replace it with a new Pod instance running the new configuration. When we have talked about updating Pods, we have really meant delete the old one and replace it with a new one having the new configuration The immutable nature of Pods is a key aspect of cloud native microservices, design and patterns and forces the following:

- When updates are needed replace all old pods with new ones that have the updates
- When failures occur replace failed Pods with new identical ones

To be clear you never update the running pod, you always replace it with a new pod containing the updates, you also never log onto failed pods and attempt fixes you build fixes into an updated pod and replace failed ones with the update ones.

Pods vs Nodes It is vital to understand the difference between those two, while both are part of the k8s infrastructure their purpose is vastly different, and while usually a pod corresponds to a single container instance, that is not always the case, as mentioned above, a Pod can in theory run multiple containers of the same image, or even of different images (more common) those can share a single state and make integration and integration between these services more robust and easier in some situations that is desirable

```
Pods are not Nodes, Nodes are the computing environments that run the containers,the
container runtimes, the kubelet and any other component of the k8s infrastructure, those
 could be many things, (virtual machines, physical machines, embedded devices and so on
)as long as they support running the k8s runtime, the pods, are a logical collection of
containers that run on a k8s Node
```

Pod strategies The atomic unit of a scheduling on K8s is the pod, this is just a fancy way of saying that apps deployed to the k8s are always managed by Pods.

Why need Pods ? Why not just run the container on the k8s node directly, the short answer is that you can not, k8s does not allow containers to run directly on a cluster or a node, they always have to be wrapped in a Pod object, there are three main reasons why Pods exist

1. Pods augment containers
2. Pods assist in scheduling
3. Pods enable resource sharing

On the augmentation front, Pods augment container in all of the following ways.

1. Labels and annotations
2. Restart policies
3. Probes, startup, readiness, liveness, and more
4. Affinity and anti affinity rules
5. Termination control
6. Security policies
7. Resource requests and limits

```
Note that containers still technically run directly on the node, the container runtime
 to be more precise, which is running on the node, but the container runtime and the
containers themselves are managed and run by the Pods, Pods are not physically running
the containers, Pods are not like containers, Pods are k8s objects/manifests and through
their pod manifest spec they instruct the kubelet, and the kubelet instructs the container
 runtime (containerd)what to do and how to run the set of containers the Pods 'manage',
 what resources to allocate for them, and all the nitty gritty fine grained control a
container might require. This is needed because there is NO equivalent 'container' spec
 which can do this, so an intermediate control object like the Pod is needed to do just
THAT
```

Labels let you group pods and associate them with other objects in powerful ways, annotations let you add experimental features and integrations with 3rd party tools and services, Probes let you test the health and status of Pods, enabling advanced scheduling, updates and more. Affinity and anti affinity rules give you control over where Pods run, Termination control lets you to gracefully terminate Pods and the app s they run, Security policies let you enforce security features, Resource requests and limits let you specify minimum and maximum values for things like CPU and memory and disk IO. Despite bringing so many features to the party, pods are super lightweight and add very little overhead. Pods also enable resource sharing, they provide execution environment for one or more containers, this shared environment includes things such as shared filesystem network stack, memory and fs-volumes.

Pods deployed directly fro the Pod manifest are called static Pods and have no super powers such as self healing scaling and rolling updates, This is because they are only monitored and managed by the local kubelet process which is limited to attempting container and Pod restarts, on the local node, if the node they are running on fails there is no control plan process watching and capable of starting a new one on a different node,

Pods deployed via controllers have all the benefits of being monitored and managed by a highly available

controller running on the control plane, the local kubelet on the node they are running on can still attempt local restarts but if restart attempts fail or the node itself fails the observing controller can start a replacement pod on a different node.

Just to be clear it is vital to understand that Pods as mortal, When they die, they are gone, there is no fixing them and bringing them back from the dead, this firmly places them in the cattle category in the pets vs cattle paradigm, pods are cattle and when they die they get replaced by another. This is why apps should always store state and data outside the pod, it is also why you should not rely on individual pods, they are ephemeral - lasting for a very short time

Pods are objects in the Nodes/Workers, and one can think of it as the Pod being room in a house, the House itself, is the Node, the Room is the pod, and the people living in that room are the containers, the Room or Pod, does not run anything, it does only manage the containers providing them with the shared state, it is like a bridge or adapter between multiple containers, as already mentioned one pod object can manage multiple containers, and can provide them with shared state which is isolated from other pods and containers

1. Each pod has its own IP address, allowing it to communicate with other Pods, between several Nodes. The container interface plugin is responsible for assigning IP addresses and setting up networking, pods on the same node communicate through a bridge, pods on a different nodes communicate through routing rules set in the CNI plugin
2. Storage between containers in a given Pod is shared using volumes, meaning that a given Pod, mounts volumes from the Host/Node to all containers that it is responsible for, these volumes are then used and shared only by the containers that this Pod governs.
3. Resources, the governing pod makes sure that each configured container does not exceed the resources allocated for it, and if it does it will restart or kill the container

`REMEMBER! Pods are not physical services running on the host, they are merely objects defined in the k8s deployment config, these pod objects are picked up by the kubelet, which is the service running on the Node, it is actually the active service that manages pods and by proxy containers defined for these pods, the pods are merely configuration objects, which tell the kubelet how to manage a common set of containers and what to do with them in the event of abnormal occurences or if the desired state diverges from the actual state`

Pods deployment The process is simple, the pods are defined in files, as already mentioned pods are mere objects, part of the k8s environment,

1. Define it in an YAML manifest file
2. Post the YAML to the REST API server
3. The server authenticates the request
4. The configuration file is validated
5. The scheduler deploys the pod to a healthy node
6. The local kubelet monitors it

The pod is deployed via a controller the configuration will be added to the cluster store as part of overall desired state that has to be maintained and a controller will monitor it. The pod deployment process is an atomic one, this means it is all or nothing deployment either succeeds or it does not, you will never have a scenario where a partially deployed pod is servicing requests, only after all the pod's resources are running and ready will it start servicing requests

Pod lifecycle The pods' lifecycle starts with the YAML object, and is served down to the API server, then it enters the `pending phase`, it is schedule to a healthy node, with enough resources, and the local

kubelet instance running on tat node instructs the container runtime to pull all required images and start all containers, once all containers are pulled and running, the pod enters the `running phase`, if it is a short lived pod, as soon as all containers terminate successfully the Pod itself terminated and enters the succeeded state. If it is a long running pod, it remains indefinitely in the `running phase`

Short lived pods can run all different types of apps, some such as web servers are intended to be long lived and should remain in the running phase indefinitely, if any containers in a long lived Pod fail the local kubelet may attempt to restart them. We say the kubelet may attempt to restart them, this is based on the container's restart policy, which is defined in the Pod object itself, Options include - Always, OnFailure and Never, always is the default restart policy appropriate for most long lived pods, Other workloads such as batch jobs, are designed to be short lived and only run until a task is complete, Once all containers in a short lived pod terminate, the pods terminate and its status is set to successful, these container restart policies - Never, OnFailure, are appropriate for short lived pods

Pod multi-container control Multi container pods are powerful pattern and heavily used in the real world, at a very high level every container should have a single clearly defined responsibility, for example an app that pulls content from a repository and serves it as a web page, has two clear functions - pull the content, serve the content

In this example one should design two containers one responsible for pulling the content and the other to serve the web page, we call this separation of concerns. This design approach keeps each container small and simple, and it encourages re-use, and makes troubleshooting simpler. However these are scenarios where it is a good idea to tightly couple two or more functions, consider the same example app that pulls content and serves it via web page, a simple design would have the sync container the one pulling content, put content updates in a volume shared with the web container, for this to work both containers need to run in the same Pod, so they share the same volume / storage from the Pods execution environment. Co-locating multiple containers in the same pod allows containers to be designed with a single responsibility but work closely with others, Kubernetes offers several well defined multi container Pod patterns.

1. Sidecar pattern
2. Adapter pattern
3. Ambassador pattern
4. Init pattern

Sidecar The sidecar pattern is probably the most popular and most generic multi container pattern it has a main app container and a sidecar container, it is the job of the sidecar to augment or perform a secondary task for the main app container, the previous example of a main app web container plus a helper pulling up to date content is a classic example of the sidecar pattern - the sync container pulling the content from the external repository is the sidecar. An increasingly important user of the sidecar model is the service mesh, at a high level service meshes inject sidecar containers into app pods and the sidecar do things like encrypt traffic and expose telemetry and metrics

Adapter The adapter pattern is a specific variation of the generic sidecar pattern where the helper container takes non standardized output from the main container and rejigs it into a format required by an external system, a simple example is NGINX logs being sent to Prometheus, Out of the box Prometheus does not understand NGINX logs, so a common approach is to put an adapter sidecar into the NGINX pod, that converts the NGINX logs into a format accepted by Prometheus

Ambassador The ambassador pattern is another variation of the sidecar pattern. This time, the helper container brokers connectivity to an external system, for example the main app container can just dump its output to a port the ambassador container is listening on and sit back while the ambassador container does the hard work of getting it to the external system.

Init That pattern is not a form of the sidecar it runs a special init container that is guaranteed to start and complete before your main app container. As the name suggests its jobs is to run tasks and initialize the environment for the main app container. For example a main app container may need permissions setting an external API to be up and accepting connections or a remote repository, cloning to a local volume, in cases like these an init container can do that prep work and will only exit when the environment is ready for the main app container, The main app container will not start until the init container completes

Takeaways Note that while the ambassador and adapter patterns might seem similar, they are meant for different tasks, while the adapter is meant as mostly translator or normalization level, for data, from one form to another. The ambassador pattern is meant for strictly handling communication between to containers or services, it abstracts away the communication details, for example Envoy, is a sidecar mesh, which serves to abstract away the database connection and communication details between a service and a database, of any type, it provides a common communications protocol that the container can use to communicate, without caring what is on the other side, as long as the other side also understands that protocol, but in all actuality the other side might be using the same pattern to receive the communication.

Deployments

Most of the time you will deploy Pods, indirectly via a higher level controllers. Examples of higher level controllers include `Deployments`, `DeamonSets` and `StatefulSets`. As an example a Deployment is a higher level Kubernetes object that wraps around a Pod and adds features such as self-healing, scaling, zero-downtime rollouts, and versioned rollbacks.

Behind the scenes, `Deployments, DeamonSets and StatefulSets` are implemented as controllers that run as watch loops constantly observing the cluster and the k8s API Server making sure observed state matches desired state.

Services

Since we have already mentioned that Pods can die, they are also managed via a higher level controllers and get replaced when they die or fail. But replacements come with a totally different IP addresses. This also happens with rollouts and scaling operations. Rollouts replace old Pods with new ones with new IPs. Scaling up adds new Pods with new IP addresses, whereas calling down takes existing Pods away. Events like these cause a lot of IP churn. The point we are making is that Pods are unreliable and this poses a challenge. Assume you have got a microservice app with a bunch of Pods performing video rendering. How will this work if other parts of the app that use the rendering service can not rely on rendering Pods being there when needed. This is where Services come in to play. They provide reliable networking for a set of Pods.

Services are fully fledged objects in the Kubernetes API - just like Pods and Deployments. They have a front end consisting of a DNS name, IP address and port. On the back end they load balance traffic across a dynamic set of Pods. As pods come and go, the Service observes this, automatically updates itself, and continues to provide that stable networking endpoint. The same applies if you scale the number of Pods up or down. New Pods are seamlessly added to the Service and receive traffic. Terminated Pods are seamlessly removed from the Service and will not receive traffic. That is the job of a Service - it is a stable network abstraction point that provides TCP and UDP load balancing across a dynamic set or number (replicas) of Pods/containers

As they operate at the TCP and UDP layer, they do not posses application intelligence, this means they can not provide app layer host and path routing. For that you need an Ingress which understand HTTP and provides host and path based routing.

```
Services bring stable IP addresses and DNS names to the unstable world of Pods, they are
 the abstraction layer, that allows other Services, Pods or Containers to communicate
```

`without having to worry about the fact that a target Pod can die`

Clusters

Namespaces are the native way to divide a single k8s cluster into multiple virtual clusters, these are not the standard Linux kernel namespaces, that we have already looked at, the ones responsible for namespacing processes on the kernel level. K8s namespaces divide the Kubernetes clusters into virtual clusters called - Namespaces

Namespaces partition a Kubernetes cluster and are designed as an easy way to apply quotas and policies to groups of objects, they are not designed for strong workload isolation. Most k8s objects are deployed into a Namespace, these objects are said to be namespaced, and include common objects like Pods, Services and Deployments. If you do not explicitly define a target namespace when deployment a namespaced object, it will be deployed to the default namespace, you can run the following command to

Namespaces are a good way of sharing a single cluster among different departments and environments for example a single cluster might have the following Namespace, Dev, Test, QA. Each one can have its own set of users and permissions as well as unique resource quotas, What they are not good for isolating hostile workloads, this is because a compromised container or pod in one namespace can wreak havoc in other namespaces, putting into context, you should not competitive workloads together. Every k8s cluster has set of pre created namespaces, virtual clusters

```
# show the list of all namespaces in the current cluster
$ kubectl get namespaces
NAME              STATUS AGE
kube-system       Active 3d
default           Active 3d
kube-public       Active 3d
kube-node-lease Active 3d
```

The default namespace is where newly created objects go unless you explicitly specify otherwise, Kube-system is where DNS the metrics server and other control plane components run, Kube-public is for objects that need to be readable by anyone, and last but not least kube-node-lease is used for node heartbeat, and managing node leases.

Namespaces are first class resources in the core v1 API group, This means that they are stable well understood and have been around for a long time, it also means you can create and manage them imperatively with `kubectl` and decoratively with YAML manifests.

```
# sample namespace that is not the default one
kind: Namespace
apiVersion: v1
metadata:
    name: shield
    labels:
    env: marvel
```

```
# apply the config to the cluster
$ kubectl apply -f shield-ns.yml
```

When you start using Namespaces you will quickly realize it is painful remembering to dd the -n or –namespace flag on all `kubectl` commands. A better way might be to set your `kubeconfig` to automatically work with a particular namespace, the following command configures `kubectl` to run all future commands against the shield Namespace

```
# this will make sure that all following commands against kubectl run in the
   context of the namespace `shield`
$ kubectl config set-context --current --namespace shield
```

To deploy to a given Namespace, as already mentioned most all objects are always tied to a namespace, and if you do not specify otherwise the default namespace will be used when deploying objects, there are two different ways to deploy objects to a specific namespace - imperatively and declaratively.

The imperative method requires you to add the -n flag to the command, the declarative method specifies the namespace in the YAML manifest file. We will declaratively deploy a simple app to the shield namespace, and test it.

```
apiVersion: v1
kind: ServiceAccount
metadata:
    namespace: shield # Namespace
    name: default << ServiceAccount name
---
apiVersion: v1
kind: Service
metadata:
    namespace: shield # Namespace
    name: the-bus # Service name
spec:
    ports:
    - nodePort: 31112
      port: 8080
      targetPort: 8080
    selector:
        env: marvel
---
apiVersion: v1
kind: Pod
metadata:
    namespace: shield # Namespace
    name: triskelion # Pod name
<snip>
```

Note the use of metadata, this is a common pattern in k8s config manifests, the metadata field is not just for humans to read, it is often used to also provide control flow to the k8s cluster itself, based on the metadata, the k8s environment knows what to do with the object, the metadata provides context for the object it is defined for in this case we define that this particular object is created for this namespace, but other metadata keys also exist and are used to e.g the name of the object is defined in the metadata field, and that same name is what can be used to reference that object in other objects. The namespace itself, we have create above, the shield name was defined in the metadata section

To deploy these resources, save the YAML manifest as file, and then simply run `kubectl apply -f shield-app.yml`, to clean up the same resources one can use the `kubectl delete -f shield-app.yml`. The nice part here is that having all of this deployed in a declarative manner, allows us to clean up the resources using the same declaration and file, no need of manual steps to delete each object, or having to know in what order

they need to be deleted, worrying about stopping some of the resources which have been allocated by these k8s objects them and so on.

Deployments

Kubernetes offers several controllers that augment Pods with important capabilities, the deployment controller is specifically designed for stateless app, we will cover some other controllers later on as well.

Theory

There are two major pieces to deployments, the spec and the controller, the deployment spec is a declarative. The deployment spec is a declarative YAML object where you describe the desired state of a stateless app, you give that to kubernetes where the deployment controller implement and manages it, the controller aspect is highly available and operates as a background loop reconciling observed state with desired state. Deployment objects, and all of their features and attributes, are defined in the apps/v1 workloads API.

`Note that the kubernetes api is architecturally divided into smaller sub groups to make it easier to manage and navigate, the apps sub group is where Deployment, DeamonSets and StatefulSet and other workload related objects are defined, we sometimes call it the workloads API`

You start with a stateless app package it as a container then define it in a Pod template, at this point you could run it on the kubernetes, however static pods like this do not self heal they do not scale and they do not allow for easy updates and rollbacks. For these reasons you will almost always wrap them in a deployment object.

`ReplicaSets`

Behind the scenes deployments rely heavily on another object called replica set. While it is usually recommended not to manage replica sets directly, deployment controller manage them, it is important to understand the role they play, at a high level `Containers` provide way to package apps and dependencies Pods allow containers to run kubernetes and enable co scheduling and a bunch of other good stuff, `ReplicaSets` manage pods and bring self healing and scaling, `Deployments` manage replica sets and add rollouts and rollbacks.

`ReplicaSets` are implemented as a controller running as a background reconciliation loop checking the right number of Pod replicas are present on the cluster, if there are not enough it adds more, if there too many it terminates some, assume a scenario where the desired state is 10 replicas but only 8 are present, it makes no difference if this is due to a failure or if it is because an `autoscaler` has increased desired state from 8 to 10, Either way, this is a red alert condition for Kubernetes, so it orders the control plane to bring two more replicas.

`Note that ReplicaSet are owned by the Deployment object, meaning that they are subordinated to them, and their lifecycle is tied to the Deployment object's lifecycle, when deployment configuration is updated create new ReplicaSets for the deployment to which the update was made, to begin the deployments of the new Pods, while the old ReplicaSet pods are being wound down.`

Pods

A deployment object only manages a single pod template, for example, an app with a front end web service and a back end catalog will have a different pod for each (two Pod templates). As a result it will need two deployment objects one managing front end pods, the other managing back end pods, however a deployment can manage multiple replicas of the same pod, for example the front end deployment might be managing 5 identical front end pod replicas.

Rollouts

Rolling updates with deployments zero downtime rolling updates of stateless apps are what Deployments are all about and they are amazing, however they require a couple of things from your microservice apps in order to work properly, - loose coupling via API and backward and forward compatibility. Both of these are hallmarks of modern cloud native microservice apps and work as follows. All microservices in an app should be decoupled and only communicate via a well defined API. This allows any microservice to be updated without having to think about clients and other microservices that interact with them everything talks to a formalized API that expose documented interface and hide specifics. Ensuring releases are backwards and forwards compatible means you ca perform independent, updated without having to factor in which versions of the clients are consuming the service. With those points in mind, zero downtime rollouts work like this:

Assume you are running 5 replicas of a stateless web front end. As long as all clients communicate via API and are backwards and forwards compatible it does not matter which of the 5 replicas a client connects to. To perform a rollout, Kubernetes creates a new replica running the new version and terminates an existing one running the old version. At this point you have got 4 replicas on the old version and 1 on the new. This process repeats until all 5 replicas are on the new version. As the app is stateless and there are always multiple replicas up and running clients experience no downtime or interruption of service, there is actually a lot that goes on behind the scenes so let us look at this

You design apps which each discrete microservice as its own Pod. For convenience self healing and scaling rolling update and more - you wrap the pod in their own higher level controller such a Deployment. Each Deployment describes all the following

- How many Pods replicas
- What image to use for the Pods container
- What network ports to expose
- Details about how to perform rolling updates

In the case of Deployments when you post the YAML file to the API server, the Pods get scheduled to healthy nodes and a deployment and `ReplicaSets` work together to make the magic happen. The `ReplicaSet` controller sits in a watch loop making sure our observed state and desired state are in agreement. A Deployment object sits above the `ReplicaSet` governing its configuration as well as how rollouts will be performed. Now assume you are exposed to a known vulnerability and need to rollout a newer image, with the fix, to do this you update the same Deployment YAML file with the new image version and re-post that to the API server. This updates the existing Deployment object with a new desired state requesting the same number of Pods but all running the newer image.

To make this happen, kubernetes creates a second `ReplicaSet` to create and manage the Pods with the new image, you now have two `ReplicaSets` - the original one for the Pods with the old image, and the new one, for the Pods with the new image. As Kubernetes increases the number of Pods in the new `ReplicaSet` it decreases the number of Pods in the old `ReplicaSet`. Net result you get a smooth incremental rollout with zero downtime.

You can rinse and repeat the process for future updates - just keep updating the same Deployment manifest file which should be stored in a version control system

The way Kubernetes knows how to correctly rollout a given pod is by using the list of labels, the Deployment controller looks for when finding Pods to update during rollouts operations in this example it is looking for Pods with the given label, `the label selector is immutable you can not change it once it is deployed`.

So imagine the following situation we have a deployment which has a container image with version 1.0, and we would like to deploy a set of new pods with a new version 2.0, this would imply we have to only change one thing, that is the version of the image defined in the YAML manifest file of the deployment so something

like changing the image tag from 1 to 2 here - `image: nigelpoulton/k8sbook:2.0`, this would trigger the internal process of creating new `ReplicaSet` for the new pods, which will start creating new pods, with the new version, scaling to the target number of replicas which we have defined in our deployment manifest file. We simply have to apply with - `kubectl apply -f deploy.yml`. To monitor the status one can use `kubectl rollout status deployment hello-deploy`.

The rollouts can also be paused, this can be done with the rollout pause command, for example `kubectl rollout pause deploy hello-deploy`, if one tries to run the `kubectl describe` provides some information on the state of the deployment and also the state of the `ReplicaSet`.

```
# pause the deployment, after we have run the apply -f deploy.yml,
   immediately just pause
$ kubectl rollout pause deploy hello-deploy
deployment.apps/hello-deploy paused
```

```
# print out the current state of the deployment object, note it is marked as
   DeploymentPaused
$ kubectl describe deploy hello-deploy
Name:                   hello-deploy
Annotations:            deployment.kubernetes.io/revision: 2
Selector:               app=hello-world
Replicas:               10 desired | 4 updated | 11 total | 11 available | 0
   unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        10
RollingUpdateStrategy: 1 max unavailable, 1 max surge

Conditions:
Type            Status                  Reason
----            ------                  ------
Available       True                    MinimumReplicasAvailable
Progressing     Unknown                 DeploymentPaused
OldReplicaSets: hello-deploy-85fd664fff (7/7 replicas created)
NewReplicaSet:  hello-deploy-5445f6dcbb (4/4 replicas created)
```

The `deployment.kubernetes.io` annotation shows the object is on revision 2, the revision 1, was the initial rollout and this update we have done is revision 2, Replicas shows the rollout is incomplete, the third line from the bottom shows the Deployment condition as progressing but paused, finally you can see that the `ReplicaSet` for the initial release is would up to 7 replicas and the one for the new release is up to 4, paused right before all new pods were actually finished deploying

```
# To resume the deployment process
$ kubectl rollout resume deploy hello-deploy
```

After we resume the process will continue from where it was paused, meaning that the remaining set of replicas and pods will be scaled up to the desired state, and the state of the deployment object will no longer show that it is `DeploymentPaused`, but the state will be completed, after all pods are up and running, in the meantime the old `ReplicaSet` will be gradually decommissioned and all of its pods with it, we will see in the next section how to actually rollback to this very `ReplicaSet` that will be getting decommissioned, which has the version 1.0 of the image.

Rollbacks

As you saw older `ReplicaSet` are wound down and no longer manage Pods. However their configuration still exists on the cluster, making them a great option for reverting to previous versions, The process of a rollback is the opposite of of a rollout you wind the one of the old `ReplicaSet` up while you wind the current one down.

Imagine the situation where the current image for the deployment object is updated to a new version, from 1 to 2, and that the deployment object was updated and everything went smoothly, the old version 1 Pods were decommissioned by the `ReplicaSet` and the new ones were now in place, however as we know the old `ReplicaSet` is still active, assuming our deployment configuration is configured to hold at least 2 versions of the deployment history with history - `revisionHistoryLimit`.

A rollout history can be obtained using the command `kubectl rollout history deployment hello-deploy`, Revision 1 was the initial deploy, that used the 1.0 image, tag, Revision 2.0 is the rolling update we just performed, The old `ReplicaSet` are still active, for the old image version, meaning that we can easily revert to those, which will in turn commission a new set of Pods with the original version of the image 1.0. So if we call `kubectl get rs`, we should at the very least see two `ReplicaSet` for bound to the hello-deploy object.

```
NAME                      DESIRED  CURRENT  READY  AGE
hello-deploy-65cbc9474c   0        0        0      42m
hello-deploy-6f8677b5b    10       10       10     5m
```

From this output we can see some useful info, like that the old `ReplicaSet` has no pods that are currently active, that one is the one which was deployed with the original image, version 1.0, and the new one has, this one is the new one with the new image version 2.0. Now what we need to do is simply commission the old one again, and decommission the new one. This will take care of creating the new pods and removing the old ones as necessary.

```
Note that rollback and update are in a way a similar thing, meaning that the process
being followed when a rollback is done is the same as rollout, the difference is only
meaningful for the person doing the process, the underlying Kubernetes infrastructure does
 not distingush between rollout and rollback, it just applies one set of ReplicaSet and
decommissions the other.
```

```
$ kubectl rollout undo deployment hello-deploy --to-revision=1
deployment.apps "hello-deploy" rolled back
```

This operation is not instant, remember that the rollback has to provision the new (technically old) set of pods with the original image 1.0, and remove the new ones, however as we have already seen this is not happening in an instant, as it is a gradual process of bringing up the pods with the old version 1.0 of the image and removing the ones with the new 2.0 version of the image.

Labels

As we have already seen that `Deployments` and `ReplicaSet` use labels and selectors to find Pods they own, it was possible in earlier versions of kubernetes for deployments to take over management of existing static pods if they had the same label, however recent versions use the system generated pod template hash label so only pods create by the `deployment/ReplicaSet` will be managed. Assume a quick example you already have 5 pods on a cluster with the label app=front-end. At a later date, you crate a deployment that requests 10 pods with the same app=front-end label. Older versions of Kubernetes would notice there were already 5 Pods with that label and only create 5 new ones, and the `Deployment/ReplicaSet` will manage all 10. However newer versions of Kubernetes tag all pods created by a `deployment/ReplicaSet` with the pod-template-hash label. This stops higher level controllers seizing ownership of existing static pods.

```
$ kubectl describe deploy hello-deploy
Name: hello-deploy
NewReplicaSet: hello-deploy-5445f6dcbb

$ kubectl describe rs hello-deploy-5445f6dcbb
Name: hello-deploy-5445f6dcbb
Selector: app=hello-world,pod-template-hash=5445f6dcbb

$ kubectl get pods --show-labels
NAME                       READY STATUS  LABELS
hello-deploy-5445f6dcbb.. 1/1   Running app=hello-world,pod-template-hash
   =5445f6dcbb
hello-deploy-5445f6dcbb.. 1/1   Running app=hello-world,pod-template-hash
   =5445f6dcbb
hello-deploy-5445f6dcbb.. 1/1   Running app=hello-world,pod-template-hash
   =5445f6dcbb
```

So you can see how the different levels of objects actually are linked together through the pod template hash, along with the label selector

Skeleton

The basic structure of the Deployment object is presented below, it is crucial to understand that the Deployment object technically controls many aspects of the underlying process of managing Pods, that includes creating and destroying `ReplicaSet` and other object. To be clear, the Deployment object is just that, an object, the actual management happens at the kubelet level, which reads these configurations and controls and manages the actual state of the Node, in the cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: hello-deploy
spec:
    replicas: 10
    selector:
        matchLabels:
            app: hello-world
    revisionHistoryLimit: 5
    progressDeadlineSeconds: 300
    minReadySeconds: 10
    strategy:
        type: RollingUpdate
        rollingUpdate:
            maxUnavailable: 1
            maxSurge: 1
    template:
        metadata:
            labels:
                app: hello-world
        spec:
            containers:
                - name: hello-pod
```

```
                image: nigelpoulton/k8sbook:1.0
                ports:
                    - containerPort: 8080
                resources:
                    limits:
                        memory: 128Mi
                        cpu: 0.1
```

- `apiVersion`: At the top the API version is specified that is to be used.
- `kind`: that is the type of the object that is being defined, in this case the Deployment
- `metadata`: gives the Deployment a name, this should be a valid DNS name, so, that means alphanumeric the dot and the dash are valid, avoid exotic characters.
- `spec`: this section is where most of the action is, anything directly below spec relates to the Deployment, anything nested below refers to the actual behavior of the deployment object
- `spec.template` is the Pod template the Deployment uses to stamp out the Pod replicas, in this example the Pod template defines a single container Pod.
- `spec.replicas` is how many pod replicas the deployment should create and manage.
- `spec.selector` is a list of labels that pods must have in order for the deployment to manage them, notice how the Deployment selector matches the labels assigned to the pod.
- `spec.revisionHistoryLimit` tells Kubernetes how may older version of `ReplicaSet` to keep, keeping more gives you more rollback options but keeping too many can bloat the object, this can be a problem on large cluster with lots of software releases.
- `spec.progressDeadlineSeconds` tells kubernetes how long to wait during a rollout for each new replica to come online, the example sets a 5 minute deadline, meaning that each new replica has 5 minutes to complete up before kubernetes considers the rollout stalled, to be clear the clock is reset after each new replica comes up meaning each step in the rollout gets its own 5 minute window.
- `spec.strategy` tells the deployment controller how to update the pods when a rollout occurs. There are some more details to take a look at here, first the `maxUnavailable` - which tells that no more than one Pod below the desired state should considered valid state, meaning that somehow two pods failed, getting us at 8, the kubelet will try to scale up to 10. The `maxSurge` - which means that we should not have more than one pod above the desired state, i.e if somehow the deployments overshoot 10, i.e become 12, the additional pods will be scaled down to match the desired state.

```
# to activate the deploy configuration
$ kubectl apply -f deploy.yml

# to get a brief description of it
$ kubectl get deploy hello-deploy

# to get full details of the object
$ kubectl describe deploy hell-deploy
```

Services

As we have already seen how pods are related containers, and then to Deployments, we have seen the core levels of abstraction, starting off from `Containers -> Pods -> -> ReplicaSet -> Deployments`, each of these provide different capabilities, the containers are what provide a meaningful way to run images, the Pods

are used to manage the resources and namespace the containers, the `ReplicaSet` are governing how to scale pods and containers, and the deployments are all about self healing and overall control over everything else below.

There is a higher level of abstraction in the kubernetes world, and these are called services. Services provide a reliable networking for a set of unreliable Pods managed by Deployments, since pods and containers effectively are immutable and ephemeral can be created and destroyed without any notice, we need a way to abstract away the gazillion number of Pods that might come into life or get destroyed, without having to think about that process at all.

When a Pods fail they get replaced by a new one with new IP. Scaling up introduces a new Pod with new IP addresses, scaling down removes Pods. Rolling updates also replace existing Pods with completely new ones with new IPs. This create a massive IP churn, and demonstrates why you should never connect directly to any particular pod. You also need to know 3 fundamental things about Kubernetes Services

- First when talking about Services, we are talking about Service object in the Kubernetes world that provides a stable networking for Pods. Just like a `Pod`, `ReplicaSet` and `Deployment`, `Services` are defined through a manifest YAML file, posted to the API server.
- Second every Service gets its own stable IP address, its own stable DNS name and its own stable port.
- Third, Services use labels and selectors to dynamically select the Pods to send traffic to.

Theory

With a service in place the Pods can scale up and down they can fail and they can be updated and rolled back, and clients will continue to access them without interruption. This is because the Service is observing the changes and updating its list of healthy Pods. But it never changes its stable IP, DNS and port

Think of services as having a static front end and a dynamic back end the front end consisting of the IP, DNS name and port never change, The back end comprising the list of healthy Pods can be constantly changing.

Labels

Services are loosely coupled with Pods via labels and selectors. This is the same technology that loosely couples Deployments to Pods and is key to the flexibility of Kubernetes. For the service to send traffic to a give Pod, the Pod needs every label the Service is selecting on. It can also have additional Labels the Service is not looking for. However the `Service might have multiple labels and the logic between those is AND`. Therefore extra care is needed when configuring selection labels for the pods

```
Services are Orthogonal to Deployments, they are not responsible for managing deployments
, they like the deployment object work with pods, therefore the Services and Deployment
objects are on the same "level" in the kubernetes hierarchy, just above the Pod, services
 do not control deployments, they work along side deployment objects to manage pods,
deployments are responsible for managing the deployment process, while services are meant
 to manage traffic and abstract away the communication between the pod and the outside
world
```

Take a look at these two definitions, one is for Service the other for Deployment

```
apiVersion: v1
kind: Service
metadata:
    name: hello-svc # <<- name of the service
    labels:
        app: hello-world
```

```
spec:
    type: NodePort
    ports:
        - port: 8080
          nodePort: 30001
          protocol: TCP
    selector:
        app: hello-world # <<- match and manage all pods with this label
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: hello-deploy # <<- name of the service
spec:
    replicas: 10 # <<- ReplicaSet properties and options start here
    selector:
        matchLabels:
            app: hello-world # <<- match and manage all pods with this label
    revisionHistoryLimit: 5
    progressDeadlineSeconds: 300
    minReadySeconds: 10
    strategy:
        type: RollingUpdate
        rollingUpdate:
            maxUnavailable: 1
            maxSurge: 1
    template: # <<- the pod definition starts here
        metadata:
            labels:
                app: hello-world # <<- this is the label of the pod that the
                   service and deployment will match
        spec:
            containers: # <<- tell the pod what image it has to use
                - name: hello-pod
                  image: nigelpoulton/k8sbook:1.0
                  ports:
                      - containerPort: 8080
                  resources:
                      limits:
                          memory: 128Mi
                          cpu: 0.1
```

Note that Deployments and Services both have to have correct selectors configured to match against the labels of the Pods. While deployments create the Pods through the use of the `ReplicaSets`, deployments are not managed by Services, services manage pods just like deployments do through the `ReplicaSets` objects. It might seem a bit odd, since the pod is defined only in the Deployment, and does not exist as a standalone "object" in the Kubernetes world

One might have noticed that the Service uses a different definition for the selector than the Deployments object, this is because the Services selector is simpler, the selector of services only provides a way for simple matching, there is no way to do advanced expression matching like the one of the Deployment object, which is the `selector.matchLabels` or `selector.matchExpression`

Here are two examples of how Deployment can use the advanced selector section, which is not available for Services, it can match either on simple labels, just like Services or on more advanced expression rules

```
# Deployment spec
spec:
  selector:
    matchLabels:
      app: my-app

spec:
  selector:
    matchExpressions:
      - key: app
        operator: In
        values: ["my-app", "test-app"]
```

The service on the other hand is rather simple, it just matches on one or more labels with AND condition

```
# Service spec
spec:
    selector:
        app: my-app
```

Endpoints

As pods come and go, the Service dynamically updates its list of healthy matching Pods. It does this through a combination of label selection and a construct called and Endpoint object. Every time you create a Service, Kubernetes automatically creates and associated Endpoint object. The endpoints object is used to store a dynamic list of healthy pods matching the service's label selector

Kubernetes constantly is evaluating the Service label selector against the healthy Pods on the cluster, as new pods that match the selector get added to the endpoints object whereas any pods that disappear get removed this means that the endpoints object is always up to date.

When sending traffic to pods via a service the cluster's internal DNS resolves to the service name to an IP address. It then sends the traffic to this stable IP address and the traffic gets routed to one of the Pods in the endpoints list, However a Kubernetes native application, can query the endpoint API directly bypassing the DNS lookup and use the service's IP address

Accessing Services from inside the cluster, there are several service types, the default one is called `ClusterIP`. A `ClusterIP` service has a stable virtual IP address that s only accessible from inside the cluster, we call this a `ClusterIP` it is programmed into the network fabric and guaranteed to be stable for the life of the service, programmed into the network fabric is a fancy way of saying that the network just knows about it and you do not need to bother with the details.

The `ClusterIP` is registered against the name of the service in the cluster internal DNS service, all pods in the cluster a are pre-configured to use the cluster DNS service meaning all pods can convert service names to `ClusterIP`

That means that if we create a new Service, called magic-sandbox will dynamically assign a stable `ClusterIP`. This name and the `ClusterIP` are automatically registered with the cluster's DNS service, These are all guaranteed to be long lived and stable. As all pods in the cluster send service discovery requests to the internal DNS they can all resolve magic-sandbox to the actual IP, based on the `ClusterIP`. `Iptables` or `IPVS` rules are distributed across the cluster to ensure traffic sent to the `ClusterIP` gets routed to matching

Pods. Net result if a Pods knows the name of a Service it can resolve that to a `ClusterIP` address and connect to the Pods behind it. This only works for Pods and other objects on the cluster as it requires access to the cluster's DNS service, It does not work outside the cluster

Lets have a simple example. Imagine a service named `one` and another one named `two`, active pods are running for both, we are located in a pod in service `one`, how would like to make a call to a Pod in service `two`. Here's how a `curl` request would look like from a Pod in Service `one` to a Pod in Service `two`:

```
# note that the HOSTNAME of the service two, includes the name of the service
   , as already established above, that is normal,
# then the namespace under which this service is deployed, by default that is
    the `default` namespace, and then the cluster
# name suffix, that is configured in the `CoreDNS` server that kubernetes is
   using as implementation of the DNS service
curl http://two.default.svc.cluster.local
```

What are the exact elements of this FQDN specified in the curl request:

- `two`: The name of the Service to call.
- `default`: The namespace where the Service `two` is deployed. If the Service is in a different namespace, replace default with that namespace name.
- `svc.cluster.local`: The default domain for Services in Kubernetes, The `svc.cluster.local` domain is the default DNS suffix for Services in Kubernetes. It is defined in the `CoreDNS` or `kube-dns` configuration. The configuration is typically stored in a **`ConfigMap`** named `coredns` (or `kube-dns` in older clusters) in the `kube-system` namespace.

`Services uses a special auxiliary EndpointSlices object internally, to manage the endpoints for the pods the service is responsible for and matches based on the selector labels`

Types

Accessing Services from outside the cluster, Kubernetes has two types of Services for requests originating from outside the cluster - `NodePort` and `LoadBalancer`

- `NodePort` Services build on top of the `ClusterIP` type and enable external access via a dedicated port on every cluster node, we call this port the `NodePort`. Since the default service type is `ClusterIP` and it registers a DNS name virtual IP and port with the cluster's DNS. `NodePort` Services build on this by adding a `NodePort` that can be used to reach the service from outside the cluster. Below is a type of `NodePort` service

`NodePort service types are not so special, since while they would allow you to access a service from the outside world, there is no other way but to know the exact IP address of a node, you call directly the node, meaning that you always hit the same node from the cluster.`

```
apiVersion: v1
kind: Service
metadata:
    name: hello-svc
    labels:
        app: hello-world
spec:
    type: NodePort
    ports:
```

```
        - port: 8080
          nodePort: 30001
          protocol: TCP
    selector:
        app: hello-world
```

Pods on the cluster can access this service by the name magic-sandbox, on port 8080. Clients connecting from outside the cluster can send traffic to any cluster node on port 30081.

- `LoadBalancer` service types make external access even easier by integrating with an internet facing load balancer, on your underlying cloud platform, You get a high performance highly available public IP or DNS name that you can access the service from, you can even register friendly DNS names to make access even simpler, you do not need to know the cluster node names or IP. `LoadBalancer` services are tightly coupled with cloud providers. They may not work in on-premises environments without additional configuration (e.g., using `MetalLB`).

`LoadBalancer has the benefit that there is a load balancer service/server infront of the cluster nodes, and unlike the NodePort type, we do not hit a cluster node IP directly, we hit the IP or domain name of the load balancer, which would then route the traffic to one of the underlying nodes on the cluster, this gives us the benefit of first not caring about node IP addresses and balancing traffic`

```
apiVersion: v1
kind: Service
metadata:
    name: my-loadbalancer-service
spec:
    type: LoadBalancer
    ports:
        - port: 80 # Port exposed by the load balancer
          targetPort: 8080 # Port on the Pods
    selector:
        app: my-app
```

What happens if we have a node inside of which we have deployed multiple pods that match the given service, would k8s load balance between the pods within the node itself ? Yes, most often the used algorithm is just simple round-robin, meaning that all pods on the given node for a given matching service/selector will be hit sequentially, one after the other, in a round robin style.

`Note these Types of services are strictly for accessing the service from outside the cluster, service based pod - pod communication, is handled differently, and is explored in depth below`

Registration

Service registration is the process of an app posting its connection details to a service registry so other apps can find it and consume it, a few important things to note bout service discovery in k8s - Kubernetes uses its internal DNS as a service registry, All k8s service automatically register their details with the DNS.

For this to work, k8s provides a well known internal DNS service that we usually call the cluster DNS. It is well known because every pod in the cluster knows where to find it, it is implemented in the kube-system namespace as a set of Pods managed by a Deployment called `coredns`. These pods are fronted by a Service called kube-dns. Behind the scenes it is based on a DNS technology called `CoreDNS`, and runs as a k8s native app.

The actual registration is divided in two parts - we can call them front and back end, briefly this is what is going on:

- `The front end` - that is the actual API server receiving the request to deploy the service on the cluster, there are certain steps (see below) that happen here, like registering the service IP in the cluster DNS creating the Service object, and other auxiliary objects
- `The back end` - this is all work that needs to be done on the actual Node that runs the Pods through the selector metadata. This is for example configuring `iptables or IPVS rules` on the actual nodes

```
# to list the actual pods which are running the coredns deployment
$ kubectl get pods -n kube-syste- -l k8s-app=kube-dns
NAME                       READY STATUS  RESTARTS AGE
coredns-5644d7b6d9-fk4c9 1/1   Running 0        28d
coredns-5644d7b6d9-s5zlr 1/1   Running 0        28d
```

```
# to list the actual deployment object that is managing these pods
$ kubectl get deploy -n kube-system -l k8s-app=kube-dns
NAME    READY UP-TO-DATE AVAILABLE AGE
coredns 2/2   2          2         28d
```

```
$ kubectl get svc -n kube-system -l k8s-app=kube-dns
NAME     TYPE       CLUSTER-IP     EXTERNAL-IP PORT(S)                  AGE
kube-dns ClusterIP 192.168.200.10 <none>      53/UDP,53/TCP,9153/TCP 28d
```

So how does the process of service registration work:

1. You post a new service manifest to the API server.
2. The request is authenticated authorized and subject to admission policies
3. The service is allocated a stable virtual IP address called `ClusterIP`
4. An endpoints object i.e `EndpointSlices` is created to hold a list of healthy pods matching the service label selector
5. The pod network is configured to handle traffic sent to the `ClusterIP`
6. The service name and IP are registered with the cluster DNS service

The step 6 is the secret sauce, we mentioned earlier that the cluster DNS is a kubernetes native app. This means it knows it is running on k8s and implements a controller that watches the API server for new Service objects, any time it observes one it automatically creates the DNS records mapping and links the service name to its `ClusterIP`. This means apps and even services do not need to perform their own service registration the cluster DNS does it for them it is important to understand that the name registered in the DNS for the service is the value stored in its `metdata.name` property, this is why it is important that service names are a valid DNS names and do not include exotic characters, The `ClusterIP` is dynamically assigned by K8s

```
apiVersion: v1
kind: Service
metadata:
    name: valid-dns-name-goes-here # this is the secret sauce and should be a
        valid dns name
```

Now that the service front end is registered and can be discovered by other apps, the back end needs building so there is something to send traffic to, this involves maintaining a list of healthy pod IPs the service will load balance traffic to. As explained earlier every service has a label selector that determines which pods it will load balance and manage traffic to. To help with backend operations such as knowing which pods to send traffic to and how traffic is routed k8s builds and endpoint object - `EndpointSlices` for every Service

The kubelet agent on every node is watching the API server for new `EndpointSlices` objects. When it sees one it creates local networking rules to redirect `ClusterIP` traffic to pod IP in modern k8s clusters the technology used to create these rules is the Linux IP virtual server (`IPVS`). Older version used `iptables`.

At this point the service is fully registered, and ready to be used. Next is the active phase which is the service discovery this happens when an actual application wants to actually connect to another Service running in the cluster.

Discovery

Service discovery is quite an important topic, but to be brief kubernetes uses DNS service, Kubernetes clusters run an internal DNS service that is the center of service discovery, service names are automatically registered with the DNS service on the cluster, every Pod and container is pre-configured to use the cluster DNS (e.g. /etc/resolv.conf). This means every Pod or container can resolve every Service name to a `ClusterIP` and connect to the Pods behind it.

The alternative form of service discovery is through environment variables every pod gets a set of environment variables that resolve Services currently on the cluster. This is extremely limited, they can not learn about new services added after the Pod they are in was created. This is a major reason DNS is the preferred method.

Let us assume there are two microservice apps on the same K8s cluster - called `enterprise` and `cerritos`. The Pods for enterprise sit behind a Service called `ent`, and the Pods for `cerritos` sit behind another Service called `cer`. They are being assigned `ClusterIP`, which are registered with the cluster DNS service, and things are as follows

App	Service name	ClusterIP
Enterprise	ent	192.168.201.240
Cerritos	cer	192.168.200.217

For service discovery to work apps need to know both of the following:

- The name of the other app they want to connect to - that is the name of service fronting the pods
- How to convert the name of the Service to an IP address that corresponds to a Pod managed by the Service

Apps developers are responsible for point 1, which is normal, They need to code apps with the names of other apps they want to consume, Actually they need to code the names of Services fronting the remote apps, or in other words the pods running the apps. K8s takes care of the second part.

Converting the names to IP addresses using the cluster DNS, happens by k8s, automatically configuring that in every container so it can find and use the cluster DNS to convert service names to IPs. It does this by populating every container's `/etc/resolv.conf` file with the IP address of the CLUSTER DNS service, as well any search domains that should be appended to unqualified names

```
$ cat /etc/resolv.conf
search svc.cluster.local cluster.local default.svc.cluster.local
nameserver 192.168.200.10
options ndots:5
```

Let us explore a little side tangent. What is the structure of the `resolv.conf` file, and here is some basic theory:

What is an unqualified name ? That is a short name such as `ent`. Appending a search domain converts it to a fully qualified domain name (FQDN) such as `ent.default.svc.cluster.local`. The following snippet

shows a container that is configured to send DNS queries to the cluster DNS at 192.168.200.10. It also lists three search domains to append to unqualified names.

Fully Qualified Domain Name (FQDN): A hostname that ends with a dot (e.g., `host.`) is considered an FQDN. This means the DNS resolver will treat it as a complete domain name and **will not append any search domains**.

Unqualified Hostname: A hostname without a trailing dot (e.g., `host`) is considered unqualified. In this case, the DNS resolver will append search domains (if configured) to attempt resolution.

- `nameserver` - well that is pretty self explanatory, this is pointing at the IP address of the cluster DNS service, this is a must have in order to resolve the Service names, otherwise there is no way for us to map the Service name to an actual `ClusterIP`, and eventually to an actual `EndpointSlices` and to a physical Pod IP address
- `search` - this one is a bit more complex, first we have to understand what an FQDN is, those are domain names that end with a dot `.`, usually the dot is omitted in most cases but according to the spec a fully qualified domain name is only the one that ends with a dot, if it does not it is not FQDN by omission, meaning that if we use the following hostname in our app configuration `ent` to refer to the enterprise service, this would be seen as non FQDN, therefore according to the `resolv.conf` it will try to resolve the host as follows `ent.svc.cluster.local, ent.cluster.local and ent.default.svc.cluster.local`, in that order it will try each and every one of those host names against the DNS cluster `nameserver`, if the DNS cluster returns a valid IP for the host we are good to go. Now what if our app configuration was actually providing the host as `ent.` note the `.` at the end, well in that case the search config will be ignored, as it will consider this a FQDN and directly try to resolve `ent.` ip address from the cluster DNS service, which will fail.
- `options` - directive allows you to configure additional resolver behavior. In our example, `options ndots:5` specifies a threshold for the number of dots (`.`) in a hostname before the resolver treats it as a fully qualified domain name (FQDN).

So what is the process, how and by whom is the `resolv.conf` file actually get interpreted, in order to obtain the actual IP address to establish TCP connection to

1. **Application:** Your app calls `getaddrinfo("example.com")` internally, take `curl` as simple example
2. **Resolver Library:** The resolver library (glibc) reads `/etc/resolv.conf` to determine the nameserver and search domains.
3. **DNS Query:** The resolver library sends a DNS query to the specified nameserver.
4. **Kernel:** The kernel handles the network communication (e.g., sending TCP packets to the DNS server).
5. **Response:** The resolver library processes the DNS response and returns the `ClusterIP` address of the service to your app.

This is however not the end, having the `ClusterIP`, does not help much since it is on a different network, if we want our Pod for service `ent` to talk to another Pod for service `cer`, we need to know the IP of at least one Pod from the `cer` service, how does that happen. Enter `Subnet masks, Gateways, IPVS and iptables…`

Network magic

`ClusterIP` are on a special network called the `service network` and there are no routes to it, this means containers send all `ClusterIP` traffic to their default gateway. In this case a default gateway is where devices send traffic when there is no known route, normally the default gateway forwards traffic to another device with a larger routing table in hope it will have a route to the destination network.

The way this works, is that as we know each device on a network has a local routing table, this routing table tells it how to route outgoing traffic, i.e to which gateway to send the traffic, when the destination network

is remote, when it is local the traffic sent directly. So here is a brief overview of the communication process

- **Your device**: `192.168.1.10`
- **Local network**: `192.168.0.0`
- **Default gateway**: `192.168.1.1`
- **Destination**: `www.google.com` (let's say its IP is `142.250.190.78`)

Destination Network	Subnet Mask	Gateway	Interface	Type
192.168.1.0	255.255.255.0	0.0.0.0	eth0	(Local Network)
172.16.0.0	255.240.0.0	192.168.1.253	eth0	(Remote Network)
10.0.0.0	255.0.0.0	192.168.1.254	eth0	(Remote Network)
0.0.0.0	0.0.0.0	192.168.1.1	eth0	(Default Gateway)

Now providing a bit more detail, so when your device wants to transmit traffic it needs to know where to send it to, this is done by using the subnet mask, the subnet mask is used to determine which network a given IP belongs to, using the routing table, your device gets the IP - `142.250.190.78` and applies the subnet masks from the routing table, top to bottom, first it checks against `255.255.255.0`, the resulting network from the masking the destination IP is `142.250.190.0`, it checks to see `if the destination network produced for this subnet mask in the table matches`, in this case it does not, the destination network for this mask `255.255.255.0` says `192.168.1.0`, no match, move to the next entry in the table. Now if we take a close look we will see that none of the destination networks, match, after the mask is applied the only one which does match, is the last one with a mask of `0.0.0.0`, it always matches, that is the default gateway, which is usually the last entry in the table, when no other subnet mask matches, that is our last hope, send the traffic to the default gateway, in this case `192.168.1.1`. So here is how the process usually goes:

1. Your device wants to send data to `142.250.190.78`.
2. It checks its local routing table and sees that `142.250.190.78` is not in the local network (`192.168.1.x`).
3. The subnet masking and routing table mapping says, "Send this to the default gateway, we did not find any other match in the routing table (`192.168.1.1`)."
4. The default gateway device) receives the data, checks its own routing table, and forwards it to the internet, or another gateway device
5. The response comes back to the router, which sends it to your device

`The key takeaway here is to realize that the subnet masking is a process that involves matching two pairs of values, from the table, when an IP is masked against a subnet mask, that produces a destination network address, that address has to match the destination network in the routing table, in order to consider the subnet masking a match, otherwise move to the next entry in the IP routing table`

The container's default gateway sends the traffic to the node it is running on, the node does not have a route to the service network either, so it sends it to its own default gateway, doing this causes the traffic to be processed by the node's kernel which is where the magic happens, Every kubernetes node runs a system service called `kube-proxy`, At a high level the proxy is responsible for capturing traffic destined for `ClusterIP` and redirecting it to the IP addresses of Pods matching the Service's label selector. `kube-proxy` is running on the node and is Pod based kubernetes native app, that implements a controller watching the API server for new Service and Endpoint objects - in other words this boils down to Pods being created and destroyed, on other Nodes. When it sees them it creates local `IPVS` rules telling the Node how to intercept traffic destined for these Service's `ClusterIP` and by "proxy" to the Pods being managed by these Service/Endpoint objects and forward it to actual Pod IPs.

This means that every time a Node's kernel processes traffic headed for an address on the Service Network a trap occurs and the traffic is redirected to the IP of a healthy pod matching the service's label selector. Meaning the Node itself becomes its own load balancer for traffic between other nodes, and pods on these nodes. `There is NO separate intermediate physical mediator service network which is responsible for the traffic,` instead the network model between the pods is FLAT, meaning nodes on a cluster communicate directly with each other, and the IP mapping happens thanks to kube-proxy running on the Pod which dynamically configures the `iptables` or in newer versions of k8s `IPVS` rules

Kubernetes originally used `iptables` to do this trapping and load-balancing. However it was replaced by `IPVS` in kubernetes 1.11. This is because `IPVS` is a high performance kernel based `L4 load balancer` that scales better than `iptables` and implements better load balancing. The previous implementation which was using `iptables`, was not well suited, since `iptables` are meant to be used first and fore most for firewall configurations not for load balancing traffic

`Key takeaway, from everything said above, you will notice that the actual LOAD BALANCING between Pod to Pod communication, that is internally between pods on different nodes, happens by the Nodes themselves or rather by the IPVS which is the kernel level load balancer implementation in Linux`

Network Traffic

So let us summarize the process, of how the entire network communication between Pods on different Nodes works

1. Query the DNS with the service name for the stable `ClusterIP` IP of the service
2. Receive the `ClusterIP` address
3. Send traffic to the `ClusterIP` address
4. No route, send to container's default gateway
5. Forward to node
6. No route, send to the node's default gateway
7. Processed by the Node's kernel,
8. Trap by `ipvs` rule
9. Rewrite IP destination to field to IP

Assume that using the example with the `enterprise` and `cerritos` from above, the enterprise app is sending traffic to `cerritos` app, first up it needs the host name of the service running the `cerritos` apps, that would be `cer` that is defined in the manifest of the Service as well as the actual `enterprise` app configuration, done by the developers themselves. An instance of the `enterprise` app, read a container running on a node, tries to send traffic to the `cer` service. But networks work with numbers not names or strings. So the container hosting the `enterprise` app sends the name `cer` to the pre configured DNS service on the container (the nameserver in the `/etc/resolv.conf` file). It is asking the DNS service to resolve this name `cer` to an actual stable IP address. The DNS service returns a stable `ClusterIP` address and the `enterprise` app sends the traffic to that address. However `ClusterIP` are on a special `service network`, they are not accessible from the Nodes, and the container does not have a route to them. So it sends it to the default gateway, which forwards it to the host Node. The Node does not have a route either, so it sends it to its own default gateway. However en-route, the request is processed and intercepted by the Node's Kernel. Now beforehand the `kube -proxy` service running on the Node, has already been watching for new Endpoints i.e Pods, on the cluster and already having configured the `IPVS` rules, it knows all the Pods and their IPs matching the target Service label, the kernel simply uses these rules to trigger a trap, and the request is redirected to an IP address of a Pod that matches the Service label selector, the `IPVS` implementation in the kernel as mentioned is a load balancer, meaning that subsequent requests to the same Service will hit other Pods, meaning the mapping will resolve to other IPs of Pods matching the Service label, maybe on a round robin principle, based on how the `IPVS` is configured, but that is not that important

Namespaces

Every cluster has an address space and Kubernetes Namespaces partition it. Cluster address spaces are based on a DNS domain that we call the cluster domain. The domain name is usually cluster.local and objects have unique names within it. For example a service called `ent` will have fully qualified name (FQDN) of `ent.default.svc.cluster.local`. The format is `<object-name>.<namespace>.svc.cluster.local`.

Namespaces let you partition the address space below the cluster domain level. For example creating a couple of Namespaces called `dev` and `acc`, will give you two new address spaces.

- `dev: <object-name>.dev.svc.cluster.local`
- `acc:<object-name>.acc.svc.cluster.local`

Object names have to be unique within a Namespace, but not across Namespaces, For example you can not have two Services named the same in the same Namespace, but you can if they were to be in different namespaces. This is useful for parallel development and production configuration. Objects can connect to services, in the local Namespace using short names such as simply specifying the `<object-name>` (that is no magic, we already mentioned above this is thanks to service discovery, the pre-configured `/etc/resolv.conf` and all that network magic already discussed above)

Imagine we had two services both in different namespaces both with different names, `svc1 in namespace dev and svc2 and in namespace acc`, we want to hit them from a Pod called `svc3` in a third namespace, `default` one

Here is a cool trick that will prove that the namespaces are nothing really complex simply domain name context separators, imagine we have the following `resolv.conf` file in our container

```
search svc.cluster.local cluster.local default.svc.cluster.local dev.svc.
   cluster.local acc.svc.cluster.local
nameserver 192.168.200.10
```

We can still use the short names of these services, `svc1` and `svc2`, without providing the FQDN, Why ? Well the `resolv.conf` file will do the heavy lifting, it will take the search config, and start attempting to send the host names to the DNS cluster server, in order starting from i

- `svc1.svc.cluster.local` - no hit, there is no such service in the system/unnamed namespace
- `svc1.svc.default.svc.cluster.local` - no hit, there is no such service in the default namespace
- `svc1.dev.svc.cluster.local` - we have a hit, since it is deployed on the dev namespace that is its actual FQDN

The DNS will resolve for the - `svc1.dev.svc.cluster.local` - will return the IP address and the rest will be handled by the `IPVS` rules and the kernel, as we have already seen above.

Now, this is not something that one should or even can do, since the `/etc/resolv.conf` file is not mutable, it is maintained and managed by the k8s runtime and environment, even if we were to change it manually, that would be on a per container instance, and will not persist across Pod and container replication

The example above, was aiming to show, and link the different networking concepts we have examined, and how they interlink, from the moment the request is made by the app, using the unqualified domain name, down to the actual IP address and traffic routing to the target Pod.

Skeleton

Below is the general skeleton of a service object, presented as a YAML manifest file. There are a few interesting properties to take a note of which are important to understand how the service integrates with the running pods managed by a deployment object

```
apiVersion: v1
kind: Service
metadata:
    name: svc-test
    labels:
        chapter: services
spec:
    type: NodePort
    ports:
        - port: 8080
          nodePort: 30001
          targetPort: 9090
          protocol: TCP
    selector:
        chapter: services
```

- `apiVersion`: At the top the API version is specified that is to be used.
- `kind`: that is the type of the object that is being defined, in this case the Service
- `metadata`: gives the Service a name, this should be a valid DNS name, so, that means alphanumeric the dot and the dash are valid, avoid exotic characters.
- `spec`: this section is where most of the action is, anything directly below spec relates to the Service, anything nested below refers to the actual behavior of the service object
- `spec.type`: In this case it is configured as `NodePort` not a default `ClusterIP`, for the sake of this example
- `spec.port`: this is the port on which the service listens to
- `spec.targetPort`: this is the port on which the app inside the container listens to
- `spec.nodePort`: this is the cluster wide port on which the service can be accessed from the outside
- `spec.protocol`: by default, using TCP, but UDP for example is also a probable option, based on the type of app

```
# to just deploy the service manifest file
$ kubectl apply -f svc.yml

# to inspect the list of service
$ kubectl get svc

# to check on the details of specific service
$ kubectl get svc svc-test
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE
hello-svc NodePort 100.70.40.2 <none> 8080:30001/TCP 8s
```

To inspect some of the created resources, alongside the service, remember that endpoints are created per service, and they hold the information about the active / alive pods and their actual IP addresses, these will be used to route the traffic to the pods, when a service hostname is hit

```
# to get the endpoint objects, which are tied to a service
$ kubectl get endpointslices
NAME ADDRESSTYPE PORTS ENDPOINTS AGE
```

```
svc-test-sbhbj IPv4 8080 10.42.1.119,10.42.0.117,10.42.1.120... 6m38s

# to get the details for the given endpoint object for the svc-test
$ kubectl describe endpointslices svc-test-sbhbj
Name:         svc-test-sbhbj
Namespace:    default
Labels:       endpointslice.kubernetes.io/managed-by=endpointslice-controller.
   k8s.io kubernetes.io/service-name=svc-test
Annotations: endpoints.kubernetes.io/last-change-trigger-time: 2021-02-05T20
   :01:31Z
AddressType: IPv4
Ports:
Name         Port Protocol

Endpoints:
- Addresses: 10.42.1.119
    Conditions:
    Ready:     true
    Hostname:  <unset>
    TargetRef: Pod/svc-test-84db6ff656-wd5w7
    Topology:  kubernetes.io/hostname=k3d-gsk-book-server-0
- Addresses: 10.42.0.117
    <Snip>
```

Take a note at the `Endpoints:` section, which describes in detail all (output abbreviated) Pods and their IP addresses, they also provide status information about the Pod. Similarly to how `ReplicaSet` are helper objects to the `Deployments` the `Endpoints` are helper objects for the `Services`.

If we were to change the type of this service which in the manifest above is of type `NodePort`, to a `LoadBalancer`, one simply needs to change the configuration slightly to `type: LoadBalancer` and remove the config for `nodePort: 30001`, the rest will be `automagically` done by the cloud provider, internally Kubernetes will interface with the cloud provider's internal load balancer, and setup the required configurations to deploy and make the service accessible over the load balancers' host name (which is managed and owned by the cloud provider directly). This can also be done in an on premise location, but the load balancer setup is something that we would have to do manually, for example using something like `MetalLB`, which has a native integration with Kubernetes.

```
# to inspect the state of the service in a continuous loop, and see when the
   external-ip is assigned, use the --watch arg
# usually the external ip will take some time to get populated since there is
    some non trivial setup to be done when the
# new load balancer instance is setup by the cloud provider
$ kubectl get svc --watch
NAME     TYPE         CLUSTER-IP    EXTERNAL-IP PORT(S)        AGE
svc-test LoadBalancer 10.43.128.113 172.21.0.4  9000:32688/TCP 47s
```

The external IP columns shows the public address of the service assigned to by your cloud provider. On some cloud platforms this might be a DNS name, instead of an IP, and it may take a minute to let the setup complete.

Note that it is very much the case that the cloud provider would create a separate load balancer service instance per service, that is configured with `LoadBalancer`, this is to provide maximum isolation from the other services, which provides better encapsulation. Each load balancer server instance will be configured to

connect to the underlying Service. This however has some (actually a lot of issues and drawbacks) in the next section, these drawbacks will be addressed using another core Kubernetes object called `Ingress`

Ingress

Ingress is all about accessing multiple web applications through a single `LoadBalancer` service. A working knowledge of Kubernetes Services is recommended before reading forward. We have already seen how Service objects provide a stable networking for Pods. You also saw how to expose apps to external consumers via a `NodePort` services and `LoadBalancer` Services. However both of these have limitations. `NodePort` only work on high port numbers - 30000 - 32767 and require knowledge of node names or IPs `LoadBalancer` Services fix this but require a 1-to-1 mapping between an internal Service and a cloud load balancer. This means a cluster with 25 internet facing, apps will need 25 cloud load balancers, and cloud load balancers are not cheap. They may also be a finite resource you may be limited to how many cloud load balancer instances you can provision, regardless of how much money you are ready to pay them !

Ingress fixes this by exposing multiple Service through a single cloud load balancer, it creates a `LoadBalancer` Service, on port `80 or 443 and uses host based and path based routing to send traffic to the correct backend Service`.

Theory

Ingress is a stable resource in the Kubernetes API. It went `general availability in Kubernetes 1.19` after being in beta for over 15 releases. During the 3+ years it was in alpha and beta, service meshes increased in popularity and there is some overlap in functionality, as a result if you plan to run a service mesh you may not need ingress. Ingress is defined in the `networking.io` API sub group as a `v1` object and is based on the usual two constructs:

- A controller - running in a reconciliation loop, to handle the state
- An object spec - a well defined and versioned manifest specification

The object spec defined rules that govern traffic routing and the controller implements the rules. However a lot of Kubernetes clusters do not ship with a built in ingress controller you have to install your own. This is the opposite of other API resources, such as `Deployments` and `ReplicaSets`, which have a built in pre-configured controller. However some hosted Kubernetes clusters such as `GKE` (Google Kubernetes Engine) have installed one. Once you have an Ingress controller you deploy Ingress objects with rules that govern how traffic hitting the Ingress is routed

On the topic of routing, Ingress operates at a layer 7 of the OSI model, also known as the app layer. This means it has awareness of HTTPS headers, and can inspect them and forward traffic based on the hostnames and paths, The following table shows how hostnames and paths can route to backend `ClusterIP` Services.

Host-based	Path-based	Backend K8s Service
shield.mcu.com	mcu.com/shield	svc-shield
hydra.mcu.com	mcu.com/hydra	svc-hydra

This shows how two different hostnames, configured to hit the same load balancer, an ingress object is watching and uses the hostnames in the HTTPS headers to route traffic to the appropriate backend service. This is an example of the HTTP host based routing pattern, and it is almost identical for path based routing

`For this to wrok name resolution needs to point to the appropriate DNS names to the public endpoint of the Ingress load balancer`

A quick side note, The OSI model is the reference model for modern networking, it comprises seven layers, numbered from 1-7, with the lowest layer concerned with things like signaling and electronics - hardware, the middle layers dealing with reliability through things like `acks` and retries and the higher layers adding awareness of user apps such as HTTPS services, Ingress operates at a layer 7, also known as the app layer and implements HTTP intelligence

```
Ingress exposes multiple ClusterIP Services through a single cloud load balancer, you
create and deploy ingress objects, which are rules governing how traffic reaching the load
 balancer is routed to the backend services, the ingress controller which you usually have
 to install yourself uses hostnames and paths to make intelligent routing decisions.
```

So from 40k feet, what is going on on a high level is that the ingress controller is actually exposed as an actual Service object in the k8s environment, unlike other controllers, which are not, the Ingress controller is exposed through a load balancer type Service, to the public internet, then that service is actually backed by pods with the NGINX image under the hood, which actually does the active on demand routing which is defined in the Ingress object.

Common Ingress Controllers include different implementations, but amongst the most popular options are:

- **NGINX**
- **HAProxy**
- **AWS ALB**
- **Traefik**
- The Ingress Controller is typically deployed as a **Kubernetes Deployment** or **DaemonSet** and runs as a pod in the cluster.

  1. **Ingress Controller Deployment:**
     - The Ingress Controller is deployed as a pod (or multiple pods) in the cluster.
     - It listens for incoming traffic and routes it based on the Ingress rules.
  2. **Service for the Ingress Controller:**
     - A **Service** is created to expose the Ingress Controller to external traffic.
     - The type of this Service can be:
       * **LoadBalancer** (for cloud providers that support external load balancers).
       * **NodePort** (for exposing the Ingress Controller on specific ports of the cluster nodes).
       * **ClusterIP** (for internal-only access, though this is less common for Ingress Controllers).
  3. **Ingress Rules:**
     - The Ingress resource defines rules for routing traffic to backend services.
     - The Ingress Controller reads these rules and configures itself, its running Pods (e.g., NGINX, Traefik) to route traffic accordingly.

- What Makes the Ingress Controller Unique, from other controllers ?
- **Backed by an Image:** The Ingress Controller is implemented as a custom application (e.g., NGINX, Traefik) running in a container. This is different from most other Kubernetes controllers, which are part of the Kubernetes control plane and are not exposed to external traffic.
- **Exposed to the Internet:** The Ingress Controller is typically exposed via a **Service** (e.g., `LoadBalancer` or `NodePort`) to handle external HTTP/HTTPS traffic. This means it is directly accessible from outside the cluster.

- **Interfaces with External Traffic:** Unlike other k8s controllers, the Ingress Controller interacts directly with external clients (e.g., web browsers, APIs) to route traffic to backend services.

```
The Ingress controller is essentially a glorified service + pods that run a
   reverse proxy (like NGINX, Traefik, HAProxy,
etc.). Its primary responsibility is to:

1.  Watch for Ingress resources: The Ingress controller monitors the
   Kubernetes API for changes to Ingress objects.

2.  Configure the reverse proxy: Based on the rules defined in the Ingress
   objects, the Ingress controller dynamically
    configures the reverse proxy (e.g., NGINX) to route traffic to the
       appropriate backend services.

3.  Handle traffic routing: The reverse proxy (e.g., NGINX) then routes
   incoming traffic to the correct backend services
    based on the configured rules.
```

Network traffic

The network traffic from external parties or clients like browsers or any other client consumer into the cluster targeted at the ingress controller is first hitting the load balancer (assume that NGINX is the used implementation) container. As we have already established, the way the ingress controller is implemented is through k8s Services and Pods, which use the preferred load balancer implementation/image.

Now here is the interesting part, as we have already seen above, the usual Pod to Pod communication within a cluster happens with the help of `IPVS` or `iptables`, depending on the implementation/configuration. The `crucial part` here is to understand that the ingress controller pods are also working in the exact same way, one might think that for them the load balancing is performed by the proxy alone, but that is not the case, even in this case, the actual traffic is load balanced by the `IPVS` or `iptables` configuration, the load balancer (NGINX) in our case is only used to create the configurations and do the actual routing.

In the ingress controller k8s object, defines the routing and mapping that basically says which route should be mapped to which service, this configuration is directly transferred to the underlying to the underlying load balancer,in our example NGINX

So here is how it works, on a high level, the steps that a client traffic request will go through to hit an underlying cluster pod

1. Client sends and HTTP request to the k8s cluster, given the host name of the cluster
2. Load balancing ingress controller is deployed as a `LoadBalancer` type Service
3. Ingress controller (NGINX) receive the request, and it matched against its configured routing rules, from the Ingress object
4. Ingress controller in this case NGINX forwards, to the correct K8s service
5. `CoreDNS` resolves the name of that service and returns the virtual `ClusterIP` address of the service
6. Kube-proxy has already configured the `IPVS` or the `iptable` rules for that service, since it runs on a `watchloop`
7. Kernel seeing the request made from the ingress controller pods, matches the virtual `ClusterIP` of the service against the actual physical Pod IP addresses

8. Traffic is sent over directly to one of these real physical IP addresses, and a pod for the target service is hit

Now looking at the flow above, you will immediately notice that starting from step 5, the steps are exactly one to one the same steps that a Pod to Pod would take to actually communicate traffic to another Pod, this is because as we have already established the ingress controller, and the ingress object, are special types of objects, which in essence procure a normal native k8s service and k8s pod objects at the end of the day, they are just like any other service and pod.

Key point to take into account here is namespaces, if the ingress object manifest specifies a namespace, for the ingress controller that implies that the routing mapping will be created against that namespace, which means that only services from that namespace will be hit, in all actuality what is going on is that simply the `/etc/resolv.conf` file is configured with `search` rule to only look in that target namespace, so it will be capable of resolving the `/ClusterIP` addresses only of hosts which are part of that namespace

Cluster Traffic

So how does that tie into the actual cluster, and the public internet, how do we access the actual service from the public internet on a given cluster. As we have already seen the ingress controller is deployed as either a `NodePort` or `LoadBalancer` service, usually a `LoadBalancer`.

A k8s cluster itself is not directly exposed to the public internet, and certainly not through a single IP address. Instead individual services are exposed and each service can have its own external IP or hostname, Here is how the process works

When the NGINX controller is created and as we have already established, the NGINX controller is a glorified method of creating a `LoadBalancer` Service is created, the cloud provider provisions a load balancer (e.g. for AWS - ELB, GCP Load Balancer). The load balancer gets an external IP, or hostname, which you can use to access the service. If the load balancer's external IP is 203.0.133.10, you can hit http://203.0.113.0 which will directly hit the exposed load balancer service or in other words the ingress controller service and the actual ingress controller pods.

Lets use an example, say that Multiple Users are on the Cloud Platform, usually not each user will receive his own cluster, unless we are speaking about enterprise customers, or generally high value users, the cluster is shared between users, users however will have their resources namespaced.

In a shared Kubernetes cluster, namespaces are the primary mechanism for isolating resources between users or teams. Namespaces provide a logical boundary for resources, ensuring that objects created by one user or team do not interfere with those created by another. Let's dive into how namespaces work and how they are used to separate objects between users.

- User A:
- Creates a Kubernetes cluster and deploys an Ingress controller.
- The Ingress controller is exposed with an external IP 203.0.113.10.
- Configures DNS to point `myapp.com` to 203.0.113.10.
- User B:
- Creates a separate Kubernetes cluster and deploys an Ingress controller.
- The Ingress controller is exposed with an external IP 203.0.113.20.
- Configures DNS to point `myapi.com` to 203.0.113.20.
- Traffic Flow:

- When a client accesses `myapp.com`, DNS resolves it to 203.0.113.10, and the request is routed to User A's cluster.
- When a client accesses `myapi.com`, DNS resolves it to 203.0.113.20, and the request is routed to User B's cluster.

Skeleton

The skeleton of the ingress manifest file is relatively simple, however there are a few things to take a note of, first as all other k8s objects, these can be namespaces as well, the service mapping will be applied as already mentioned only to services in that namespace, therefore you have to make sure that the service rules and the namespace match and that namespace has a service with that name already created

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: example-ingress # the name of the ingress object
    namespace: app-namespace # the name of the specific namespace
spec:
    rules:
        - host: example.com
          http:
              paths:
                  - path: /
                    pathType: Prefix
                    backend:
                        service:
                            name: app-service # only services which are in
                               the app-namespace will be hit, so this service
                                better be there
                            port:
                                number: 80
```

Here is an example with the ingress controller configured to use TLS and on top of that in the regular `pass through` mode, which is basically making the ingress controller act as a dumb forwarder, meaning that there is no way for the ingress controller to read the request, therefore there is no way to configure sub path routing for a service, meaning we can only rely on `SNI`, to resolve the hostname

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: passthrough-ingress
    annotations:
        nginx.ingress.kubernetes.io/ssl-passthrough: "true"
spec:
    tls:
        - hosts:
              - example.com
              - api.example.com
    rules:
        - host: example.com
          http:
              paths:
```

```
                - path: /
                  backend:
                      service:
                          name: example-service
                          port:
                              number: 443
        - host: api.example.com
          http:
              paths:
                  - path: /
                    backend:
                        service:
                            name: api-service
                            port:
                                number: 443
```

A small side tangent, what is really `SNI` - server name indication is an extension of the TLS protocol that allows a client to specify the hostname it is trying to connect to during the initial handshake. This is crucial for servers hosting multiple websites or services on a single IP address, like our beloved k8s clusters and cloud platform providers. So that is how that works, is it rather simple, when you have encrypted traffic and no way to decrypt it, the `SNI` springs into action, and the ingress/NGINX service is able to determine at the very least the hostname, that is why route/path based routing is not possible, because it is part of the headers, and those are encrypted (so is the original host name information by the way, were it not for `SNI` extension, we would not be able to read the host name either), `SNI` does not help here

1. Client Hello: When a client initiates a TLS connection, it sends a "Client Hello" message that includes the `SNI` extension, specifying the hostname (e.g., example.com) it wants to access.
2. Server Selection: The server uses the `SNI` information to select the correct SSL/TLS certificate for the requested hostname.
3. Handshake Completion: The server responds with the appropriate certificate, and the TLS handshake proceeds as usual, securing the connection.

Here is another example which in this case uses a re-encrypt approach, meaning that it terminated the connection exactly at the ingress controller pods, and the traffic down to the target services and their pods is re-encrypted by a special certificate provided in the configuration. This process is often called TLS edge termination, the certificate to re-encrypt the traffic is called edge certificate

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
    name: example-ingress
    annotations:
        nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
    tls:
        - hosts:
              - example.com
          secretName: example-tls # this is the certificate to be used for re
             -encrypting the traffic to the pod
    rules:
        - host: example.com
```

```
    http:
      paths:
        - path: /app
          pathType: Prefix
          backend:
            service:
              name: app-service
              port:
                number: 80
        - path: /api
          pathType: Prefix
          backend:
            service:
              name: api-service
              port:
                number: 80
```

Storage

Storage is critical to most real world production apps, fortunately, k8s has a mature and feature rich storage subsystem, called the `persistent volume subsystem`

K8s supports lots of storage back ends and each requires slightly different configuration, The examples here are made to work with Google k8s engine, clusters and will not work on other cluster types. The principle and theory that you will learn is applicable to all types of Kubernetes though

The big picture

As mentioned k8s supports many storage providers, block, file and object storage from a variety of external systems that can be in the cloud or your on premise datacenters. However no matter what type of storage or where it comes from when its exposed on k8s it is called a volume. Azure File resources surfaced in k8s are called volumes as are block devices from HPE. Modern volumes are based on the Container Storage Interface (CSI) which is an open standard aimed at providing a clean storage interface for container orchestrators such as k8s. Prior to the CSI all storage plugins were implemented as part of the main k8s code tree. This meant that they had to be open source and all updates and bug fixes were tied to the main k8s release cycle. This was a nightmare for developers & maintainers. However now with the existence of CSI storage vendors no longer need to open source their code.

The Storage Providers

K8s uses storage from a wide range of external systems. These can native cloud services such as AW Elastic Block Store, Azure File, but can also be traditional on premise storage arrays providing NFS volumes and such. Other options exist but the take home point is that k8s gets its storage from a wide range of external systems including battle hardened enterprise grade systems, from all the major data management companies. Some obvious restrictions apply. For example you can not use AWS storage services if your k8s service is running on Microsoft Azure.

Each provider or provisioner needs CSI plugin to expose their storage assets to k8s, the plugin usually runs as a set of Pods in the kube-system Namespace.

The CSI - Container Storage Interface

The Container storage interface is a vital piece of the k8s storage jigsaw and has been instrumental in bringing enterprise grade storage from traditional vendors to k8s. However unless you are a developer writing storage plugins you are unlikely to interact with it very often. It is an open source project that defines a standards based interface so that storage can be leveraged in an uniform way across multiple container orchestrators. For example a storage vendor should be able to write a single CSI plugin that works across multiple orchestrators such as k8s and docker swarm. In practice k8s is the focus but docker is implementing support for the CSI as well.

In the k8s world the CSI is the preferred way to write plugins, drivers and means that plugin code no longer needs to exist in the main k8s code tree. It also exposes a clean interface and hides all the ugly volume machinery inside of the k8s code.

From a day to day perspective your main interaction with the CSI will be referencing the appropriate CSI plugin in your YAML manifest files and reading its documentation to find supported features and attributes. Sometimes we call these plugins - "provisioners" especially when we talk about storage classes later.

The Persistent Volume Subsystem

This is where we will spend most of our time configuring and interacting with storage. At a high level persistent volumes (PV) are how external storage assets are represented in k8s, Persistent volume claims (PVC) are like tickets that grant a Pod access to the persistent volume, the Storage Classes (SC) make it all possible and dynamic.

Here is an example - assume that you have an external storage system with two tiers of storage, - flash/ssd fast storage, and mechanical slow archive storage (hard drives). You would expect apps on your k8s cluster to use both, so you can crate two Storage Classes and map them as follows:

External tier	K8s Storage class name
SSD	sc-fast
MECHANICAL	sc-slow

With that Storage class in place apps can create volumes on the fly by creating persistent volume claims (PVC) referencing either of the storage classes. Each time this happens the CSI plugin referenced in the SC instructs the external storage system to create an appropriate storage asset. This is automatically mapped to a Persistent Volume (PV) on K8s and the app uses the persistent volume claims or PVC to claim it and mount it for use.

The section below tries to link all the components of the persistent volume subsystem, as best as it can, giving an example with 3 different persistent volume providers - `NFS, Ceph and AWS`, all of those are simply storage providers, which allow us to store data of any type, the underlying implementation is not important, what is important to understand is that we need the components of the persistent volume subsystem to be able to interact with them within our pods

PersistentVolume A PV is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using a `StorageClass`. PVs represent the actual storage resources available in the cluster, such as a disk on AWS EBS, a GCP Persistent Disk, or an NFS share. PVs are cluster-wide resources and are not tied to a specific namespace. They exist independently of Pods and PVCs. Think of a PV as a physical hard drive or a network-attached storage device.

`The PV of persistent volume actually represents the physical storage devices that the cloud platform is exposing to the user, that would be things like hard drives, solid state`

drives, NAS (Network attached storage)and more. These are stting at the lowest level in the the persistent volume system hierarchy in the k8s environment

StorageClass A `StorageClass`, this is an abstraction on top of the persistent volume, the `StorageClass` basically is responsible for creating the link between the physical device / persistent volume and the PVC and the Pods which would later on use them. This happens through the use of plugins or provisioners, these plugins are the custom vendor drivers which know how to interact with the underlying persistent volume - for example Network File System (NFS) or Ceph (a storage system that provides object, block and file storage). These have to be configured externally of the cluster, the `StorageClass` provides the means of providing an interface to interact with them, through the plugin (provisioner), the actual plugin has to be installed on a per use case basis, meaning based on which type of persistent volume provider one would like to use. Think of the `StorageClass` as the way to define the systems drivers for persistent volumes, to enable interaction with them, these system drivers are implemented by the plugin/provisioner which is defined in the `StorageClass` object

So here is the definition that a systems administrator, or a user would have to perform to setup the link between a persistent volume system, like NFS, Ceph and so on, to allow the cluster and the pods within to be configured to interact with those custom storage providers. The example below is usually what a systems administrator on the cluster would do, since most of the regular users would really rely on the default `StorageClass` provided by the cloud platform, there are cases where the users - large enterprise organizations might provide their own `StorageClass` for more exotic persistent volume vendors

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: nfs-storage
provisioner: cluster.local/nfs-client
parameters:
    archiveOnDelete: "false"
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: ceph-rbd
provisioner: rook-ceph.rbd.csi.ceph.com
parameters:
    clusterID: rook-ceph
    pool: replicapool
    imageFormat: "2"
    imageFeatures: layering
---
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: aws-ebs
provisioner: ebs.csi.aws.com
parameters:
    type: gp2 # General Purpose SSD
    encrypted: "true" # Optional: Enable encryption
reclaimPolicy: Delete
volumeBindingMode: WaitForFirstConsumer
```

PersistentVolumeClaim A PVC is a request for storage by a user. It is a way for users to ask for a specific amount of storage with certain characteristics (e.g., size, access mode). PVCs act as a "middleman" between Pods and PVs. They allow users to request storage without needing to know the details of the underlying storage infrastructure. PVCs are namespaced resources, meaning they belong to a specific namespace. When a PVC is created, Kubernetes binds it to a PV that matches the requested size and access mode. Think of a PVC as a "ticket" that a user creates to request storage. The ticket is then matched to an available physical storage device or in other words a persistent volume (PV) - like a "hard drive".

So here are the definitions of the PVCs for the storage classes mentioned above, as we can see the PVC reference the name of the storage class, in this case we have 3 different ones - NFS, Ceph and AWS. These provide the rules on how these storage classes and in particular the actual persistent volumes will be accessed by the pods which require access to them.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: nfs-pvc
spec:
    accessModes:
        - ReadWriteMany
    resources:
        requests:
            storage: 10Gi
    storageClassName: nfs-storage
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: ceph-pvc
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
            storage: 10Gi
    storageClassName: ceph-rbd
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: aws-ebs-pvc
spec:
    accessModes:
        - ReadWriteOnce
    resources:
        requests:
            storage: 10Gi
    storageClassName: aws-ebs
```

Pod A Pod is the smallest deployable unit in Kubernetes. They can request storage by referencing a PVC. Pods are ephemeral, meaning they can be created, deleted, and rescheduled frequently. However, the data

stored in a PV (via a PVC) persists even if the Pod is deleted. It needs storage to read/write data, which it gets by mounting a PVC.

Here is the final part of the puzzle, linking the PVC / claims with actual pods, which will use these storage class and persistent volume providers to mount inside their own environments, take a good look at the `volumeMounts` and the `volumes` properties, in the manifest below. The `volumes` links the pod with the `pvc` the `volumeMounts` tells the pod where to mount / what path to mount that `pvc` into the container. That way we have just made the ephemeral pod, which has no context of persistent storage, into something that can store data, now when if the pods die and are re-created they will be able to retain a persistent state

```
apiVersion: v1
kind: Pod
metadata:
    name: nfs-pod
spec:
    containers:
        - name: nfs-container
          image: nginx
          volumeMounts:
              - name: nfs-storage
                mountPath: /usr/share/nginx/html
    volumes:
        - name: nfs-storage
          persistentVolumeClaim:
              claimName: nfs-pvc
---
apiVersion: v1
kind: Pod
metadata:
    name: ceph-pod
spec:
    containers:
        - name: ceph-container
          image: nginx
          volumeMounts:
              - name: ceph-storage
                mountPath: /usr/share/nginx/html
    volumes:
        - name: ceph-storage
          persistentVolumeClaim:
              claimName: ceph-pvc
---
apiVersion: v1
kind: Pod
metadata:
    name: aws-ebs-pod
spec:
    containers:
        - name: aws-ebs-container
          image: nginx
          volumeMounts:
              - name: aws-ebs-storage
```

```
                mountPath: /usr/share/nginx/html
    volumes:
        - name: aws-ebs-storage
          persistentVolumeClaim:
              claimName: aws-ebs-pvc
```

StorageClass

As far as K8s goes storage classes are resources in the `storage.k8s.io/v1` API group. The resource type in `StorageClass` and you define them in regular YAML format manifest files, and they are posted to the API server for deployment, you can use the `sc` short name to refer to them when using `kubectl`

As with all k8s yaml, kind tells the api server what type of object you are defining and the api version tells it which version of the schema to use, when creating it, metadata.name is an arbitrary string that lets you give the object a friendly name. This example is using fast-local.provisioner and tells Kubernetes which plugin to use and the parameters block let you fine tune the storage attributes, finally the `allowedTopologies` property lets you list where replicas should go. Also a few notes worth noting

Multiple `StorageClass` objects You can configure as many `StorageClass` as you would need. However each class can only relate to a single type of storage on a single back end. For example if you have a Kubernetes cluster with `StorageOS` and `Portworx` storage back end you will at needs least two `StorageClasses`, this is because you would need at the very least two provisioners/plugins which are for the two back ends, since they do not share the same internal working, therefore different plugin implementations are in order and required.

One the flip side each back end storage system can offer multiple classes tiers of storage each of which, needs its own `StorageClass` on Kubernetes A simple example which will be explored later on, is that a slower standard persistent disk and the faster SSD persistent disk tiers offered by the Google Cloud back end. These are typically implemented with the following SC on the `GKE`

1. `standard-rwo` for the slower standard disk
2. `premium-rwo` for the faster SSD

The following `StorageClass` defines a block storage volume on a `Commvault Hedvig` array that is replicated between data centers in `Sunderland and New York`, it will only work if you have `Commvault Hedvig` storage systems and appropriate replication configured on the storage system.

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
    name: sc-hedvig-rep
    annotations:
provisioner: io.hedvig.csi
parameters:
    backendType: "hedvig-block"
    rp: "DataCenterAware"
    dcName: "sunderlang,new-york"
```

Persistent Volumes There are few other important settings you can configure in a `StorageClass`, such as access mode, reclaim policy. Kubernetes supports three access modes for volumes - `READWRITEONCE (RWO), READWRITEMANY (RWM), READONLYMANY (ROM)`.

Access Mode The access mode type specifies that and by what a given storage class can be accessed,

- `ReadWriteOnce(RWO)` - defines a persistent volume, that can only be bound as R/W by a single Node, and by proxy a Pod, attempts to write or read data from another Pod, will fail, this is idea for stateful aps like databases where only one app can access the data at a time, to retain atomicity.
- `ReadWriteMany(RWX)` - defines a persistent volume, that can be bound as a R/W by multiple Nodes, and by proxy Pods, this behavior is strictly dependent on the underlying apps, since such concurrent reads and writes are unpredictable, and the underlying apps have to be able to handle this gracefully. A good use case is apps that need shared data, and that can publish that shared data in real time like content management systems, but the data they publish does not necessarily interfere with each other, and can be published without concurrent conflicts.
- `ReadOnlyMany(ROX)` - defines a persistent volume that can be bound as a R/O by multiple Nodes, and by proxy Pods, these are meant for read only, access and are mostly useful for app configurations, stateful read only configurations, which are used to bootstrap the apps and Pods and are also used during the runtime of these apps and services.

What are the general guidelines on which type of access mode is used where:

- Use `ReadWriteOnce (RWO)` for stateful applications like databases.
- Use `ReadOnlyMany (ROX)` for sharing static immutable data across multiple apps.
- Use `ReadWriteMany (RWX)` for applications that need shared read-write access, like file/ftp servers.

Reclaim policy A volume's `ReclaimPolicy` tell kubernetes how to deal with a persistent volume when its PVC is released. Two policies currently exist - `Delete and Retain`.

`Delete` is the most dangerous and its the default for Persistent volumes, It deletes the Persistent Volume and associated storage resource on the external storage system when the PVC is released. This means all data will be lost! You should obviously use this policy with caution.

`Retain` will keep the associated persistent volume object on the cluster as well as any data stored on the associated external asset. However other PVCs are prevented from using it in future. The obvious disadvantage is it requires manual clean up.

Skeleton To finalize the section about `StorageClass`, here is a brief overview of the skeleton spec structure of the `StorageClass` object, this is just a brief overview of what it supports as a k8s native object

```
kind: StorageClass
apiVersion: storage.k8s.io/v1
metadata:
    name: slow # the name of the storage class object
annotations:
    storageclass.kubernetes.io/is-default-class: "true"
provisioner: kubernetes.io/gce-pd # this is the plugin or provisioner spec/id
parameters:
    type: pd-standard # any parameters to enhance or configure the
       provisioner
reclaimPolicy: Retain
```

1. The `StorageClass` objects are immutable this means that you can not modify them after they are deployed
2. The metadata.name should be a meaningful name as it is how you and other objects refer to the class

3. The terms provisioner and plugin are used interchangeably
4. The parameters block is for plugin specific values and each plugin is free to support its own set of values configuring this section requires knowledge of the storage plugin and associated storage back end. Each provisioner usually provides documentation.

The `StorageClass` lets you dynamically create physical back end storage resources that get automatically mapped to a Persistent Volumes on Kubernetes. You define `StorageClasses` in YAML files that reference a plugin or provisioner and tie them to a particular tier of storage back end. For example high performance SSD storage in the AWS Mumbai Region. The `StorageClass` needs a name and you deploy it using the `kubectl` apply. Once deployed the `StorageClass` watches the API server for new PVC objects referencing its name. When matching `PVCs` appear the `StorageClass` dynamically creates the required asset on the back end storage system and maps it to a persistent volume on the K8s environment. Apps can then claim it with a PVC.

Application

As we have already established the `StorageClass` are usually created by system administrators, on most cloud platform, those `StorageClass` are mostly generic ones providing the needed basic capabilities for users and their pods to interact with a persistent storage medium.

Using existing `StorageClass` The following command will list all SC defined on a typical (in our example `GKE` cluster), based on the type of cloud platform provider and cluster those will likely be different

```
$ kubectl get sc
NAME          PROVISIONER            RECLAIMPOLICY        VOLUMEBINDINGMODE
premium-rwo   pd.csi.storage.gke.io  Delete               WaitForFirstConsumer
standard      (default)              kubernetes.io/gce-pd Delete Immediate
standard-rwo  pd.csi.storage.gke.io  Delete               WaitForFirstConsumer
```

There is quite a lot to learn from the output. First up all three SC were automatically created when the cluster was built by the actual cloud provider (Google in this case). This is common on hosted Kubernetes platforms, but your cluster may not have any The one on the second line is listed as the default, This means it will be used by any PVC that do not explicitly specify an SC, Default SC are only useful in development environments and times when you do not have specific storage requirements, in production environments you should explicitly use a named SC in your PVC that meets the requirements of the app. The `PROVISIONER` column shows two of the SC using the CSI plugin the other is using the legacy in-tree plugin, built into the source tree of k8s, The `RECLAIMPOLICY` is set to Delete as for all three. This means any PVC that use these `SC` will create `PV` and volumes that will e deleted when the PVC is deleted, This will result in data being lost, The alternative is Retain. Setting `VOLUMEBINDINGMODE` to immediate will create the volume on the external storage system as soon as the PVC is created, if you have multiple data centers or cloud regions, the volume might be created in a different data center or region than the Pod that eventually consume it. Setting the `WaitForFirstConsumer` will delay creation until the Pod using the PVC is created. This ensures that the volume will be created in the same data center or region as the Pod and actually the Node where the pod is running on.

You can use `kubectl` describe to get more detailed information, just as with any other k8s object, and `kubectl get sc <name> -o yaml` will show the full configuration in YAML format.

```
$ kubectl describe sc premium-rwo
Name:                  premium-rwo
IsDefaultClass:        No
Annotations:           components.gke.io/component-name=pdcsi-addon...
Provisioner:           pd.csi.storage.gke.io
```

```
Parameters:            type=pd-ssd
AllowVolumeExpansion:  True
MountOptions:          <none>
ReclaimPolicy:         Delete
VolumeBindingMode:     WaitForFirstConsumer
Events:                <none>
```

Let us create a new volume using the `premium-rwo StorageClass`, the premium one is a basic fast SSD storage, First let us list any existing persistent volumes and persistent volume claims, we will quickly see that there are none, created by us, which is normal, those are user managed objects, unlike most of the `StorageClass` objects which are rarely provisioned by regular users, outside of enterprise organizations, which have more exotic needs.

```
$ kubectl get pv
No resources found

$ kubectl get pvc
No resources found
```

The following PVC definition uses the `premium-rwo` storage class / driver and reserves or requests 10 gigs of storage, with the proper access type - `ReadWriteOnce`, this means that the persistent volume created by this PVC eventually, will only be accessible by only one single PVC, that is, and the command to create it following the manifest below

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc-prem
spec:
    accessModes:
        - ReadWriteOnce
    storageClassName: premium-rwo
    resources:
        requests:
            storage: 10Gi
```

```
$ kubectl apply -f pvc.yaml
persistentVolumeClaim/pvc-prem created
```

The following commands will show a PVC has been created, however it is in the pending state and no `PV - persistent volume` has been created on the actual underlying storage device, This is because the `premium-rwo StorageClass` volume binding mode is set to `WaitForFirstConsumer`, as we have seen above, when we described that `StorageClass`, meaning that it will not provision a volume until a Pod claims it.

```
$ kubectl get pv
No resources found

$ kubectl get pvc
NAME     STATUS  VOLUME CAPACITY ACCESS
pvc-prem Pending premium-rwo     1m
```

Create the Pod from the following manifest document, and that will then mount a volume via the `pvc-prem PersistentVolumeClaim` which will in turn actually provision the persistent volume on the storage device through the use of the `StorageClass` and drivers/plugin/provisioner

```
apiVersion: v1
kind: Pod
metadata:
    name: volpod
spec:
    volumes:
        - name: data
          persistentVolumeClaim:
              claimName: pvc-prem
    containers:
        - name: ubuntu-ctr
          image: ubuntu:latest
          command:
              - /bin/bash
              - "-c"
              - "sleep 60m"
          volumeMounts:
              - mountPath: /data
                name: data
```

This will actually force the persistent volume to be created since we now have a pod that is using it actually, the `volumeMounts` above show that the `pvc-prem` will be bound to /data in the container. The volumes section defines what volumes are defined for the pod, in this case it is just one, and that one uses the already created `claimName` `pvc-prem`. Give the pod some time to start and after that log out the output of `kubectl get pv`

```
# provision the pod, with the spec provided above, for this example we mostly
    care about the pv that will be created
$ kubectl apply -f pod.yml
pod/prempod created

# wait for some time before doing this, to allow for the pod and all
   resources to actually get provisioned
$ kubectl get pv
NAME         CAPACITY MODES RECLAIM POLICY STATUS  CLAIM
   STORAGECLASS
pvc-796af... 10Gi     RWO   Delete         Bound   default/pvc-prem premium-
   rwo
```

To drop the resources we have just created, one would simply do the following, however remember what will happen with the persistent volume, since the `StorageClass` defines is at Delete as the `ReclaimPolicy`, then the persistent volume will also be deleted once nothing is bound to it anymore.

```
# deletes the pod, however the persistent volume will also go away, since the
    StorageClass defines the ReclaimPolicy as delete
$ kubectl delete pod prempod
```

Creating new `StorageClass` What about creating a custom `StorageClass` object, that is then used to create a new volume just as the examples above. The storage class that we will create is defined in the following manifest file, and is of the following properties

- Fast SSD device, `pd-ssd`
- Replicated - `regional-pd`

- Create on demand - `volumeBindingMode: WaitForFirstConsumer`

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: sc-fast-repl
provisioner: pd.csi.storage.gke.io
parameters:
    type: pd-ssd
    replication-type: regional-pd
volumeBindingMode: WaitForFirstConsumer
reclaimPolicy: Retain
# allowedTopologies:
# - matchLabelExpressions:
#   - key: topology.kubernetes.io/zone
#     values:
#     - europe-west2-b
#     - europe-west2-c
```

`Remember that the parameters above, are strictly for the provisioner to use, in this case it is a proprietary google based storage provisioner/plugin - pd.csi.storage.gke.io these parameters are meaningless for any other provisioner generally`

The parameters field in a `StorageClass` is a key value pair map that allows you to pass configuration options to the provisioner, these parameters share specific to the provisioner and the underlying storage system, for example the AWS EBS - uses parameter names such as type, Ceph might use parameters like `pool`, `clusterID`, `imageFormat`, and NFS might use parameters like server and path. These parameters are not standardized across all CSI drivers or provisioners, each CSI driver or provisioner is defining its own set of supported parameters based on the capabilities of the storage backend. The CSI spec standardizes the API for communication between the k8s and the storage backend but it does not enforce specific parameters.

```
# provision the storage class, remember that it is immutable once created
   there is no changing it
$ kubectl apply -f my-sc.yml
storageclass.storage.k8s.io/sc-fast-repl

# list the details of all storage classes, where we can see the new one being
    created
$ kubectl get sc
NAME         PROVISIONER            RECLAIMPOLICY VOLUMEBINDINGMODE
premium-rwo  pd.csi.storage.gke.io Delete        WaitForFirstConsumer
sc-fast-repl pd.csi.storage.gke.io Retain        WaitForFirstConsumer
```

Now to deploy a new manifest of a persistent volume claim that is going to use the new `StorageClass`, we can simply use the following manifest as follows. The manifest below is a simple manifest for a PVC that is using the custom storage class created above, just as with any other PVC, the volume size is specified and the `accessModes`,

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
    name: pvc2
spec:
```

```
    accessModes:
        - ReadWriteOnce
    storageClassName: sc-fast-repl
    resources:
        requests:
            storage: 20Gi
```

Finally the final steps is to create a new Pod, that is going to be using this PVC, similarly to before as we have created the custom storage class with `WaitForFirstConsumer`, that means that the volume would be only created on demand when at least one pod with a valid PVC that is bound to that `StorageClass` is configured to be used in a Pod.

```
apiVersion: v1
kind: Pod
metadata:
    name: volpod
spec:
    volumes:
        - name: data
          persistentVolumeClaim:
              claimName: pvc2
    containers:
        - name: ubuntu-ctr
          image: ubuntu:latest
          command:
              - /bin/bash
              - "-c"
              - "sleep 60m"
          volumeMounts:
              - mountPath: /data
                name: data
```

Since the Pod and PVC were both deployed with a manifest, we can simply either use the manifest files, to delete them, or manually delete the objects, usually the preferred way is to use the same manifest files used to create the objects, to remove them, to remove them, execute the following

```
# delete the pod
$ kubectl delete -f volpod.yml

# delete the persistent claim
$ kubectl delete -f pvc2.yml
```

Now even those are now deleted, the `kubectl get pv` will show that the persistent volumes still exist. This is because the storage class it was created from is using the Retain policy. This keep the persistent volume associated with the back end volumes and the data even when the PVC are deleted.

`ConfigMaps` & Secrets

Most business app comprise of two main parts - the app and the configuration. A simple example is a web server such as NGINX or httpd. Neither are very useful without a configuration. However when you combine them with a configuration they become extremely useful.

In the past we coupled the app and the configuration into a single easy to deploy unit. As we moved into the early days of cloud native microservices and apps we brought this model with us. However it is an anti-pattern in the cloud native world. You should de couple the app and the configuration.

The big picture

As already mentioned most apps need configuration to function correctly, This does not change in the k8s world, Let us consider a simple example - we would like to deploy modern apps to K8s and have 3 distinct environments, - Dev, Test, and Prod. The developers write and update the app, initial testing is performed in the dev environment, further testing is done in the test environment here more stringent rules are taken. Finally stable components graduate to the prod environment.

However each env, has subtle differences, this includes things such as number of nodes, configuration of nodes, network and security policies different sets of credentials and certificates and more. You currently package each app microservice with its configuration backed into the container. With this in mind you have to perform all of the following on every business app

- Build three distinct images, for each env, each of which will have backed in the different configurations for each env
- Store the images in three distinct repositories, (dev, test and prod)
- Run each version of the image in a specific environment

Every time you change the config of an app, even the smallest change like fixing a typo, you will need to build and package and entirely new image and perform some type of update to the entire app.

There are several drawbacks to the approach of storing the app and its configuration as a single artifact - container image. As your dev, test and prod environments have different characteristics each env needs its own image, A dev or test image will not work in the prod env, because of things like different security credentials and restrictions. This requires extra work to create an maintain 3x copies of each app. This complicates matters and increases the chances of mistakes, including things that work in dev and test but not prod.

How should it work then ? Well you should be able to build a single version of your app, that is shared across all three environments, you store a single image in a single repo, you run a single version of that image in all environments. To make this work you build your app and images as generically as possible with no embedded configuration, you then create and store configurations in a separate objects, and apply a configuration to the app at the time it is run. For example you have a single copy of a web server that you can deploy to all three environments. When you do deploy it to prod you apply the prod configuration. When you run it in dev you apply the dev configuration and so on.

In this model you create and test a single version of each app image, and you can even re use images across different apps. For example a hardened stripped down NGINX image can be used by lots of different apps, just load different configurations at run time

`ConfigMaps`

K8s provides an object called a `ConfigMap`, that lets you store configuration data outside of a Pod, it also lets you dynamically inject the config into a Pod at run-time. When we use the term Pod we really mean container, after all it is ultimately the container that receives the configuration data and runs the app, but the pod is the logical object that the `ConfigMap` is bound to, since the Pod is the overarching manager of the containers and images that are being run

`ConfigMaps` are first class objects in the k8s API, under the core API group and they are in v1 of the API. They are stable, and have been around for a while, you can operate on them with the usual `kubectl` commands and they can be defined and deployed via the usual manifest YAML file format

Premise `ConfigMap` are typically used to store non sensitive configuration data such as environment variables and entire configuration files, hostnames, server ports, account names etc. You should not use the `ConfigMap` to store sensitive data as certificates and password. K8s provides a different object called a Secret, for storing sensitive data. Secrets and `ConfigMaps` are very similar in design and implementation the major difference is that k8s takes steps to obscure the data stored in secrets. It makes no such efforts to obscure data stored in `ConfigMaps`

The way `ConfigMaps` are defined is as a map of key value pairs and we call each key value pair and entry.

- Keys are an arbitrary name that can be created from alphanumeric, dashes, dots and underscores
- Values can contain anything really, including multiple lines with carriage returns
- Keys and values are separated by a colon - `key: value`

More complex examples can store entire configuration files like that, below is the actual value that the `ConfigMap` stores:

```
key: conf
value:
    directive in;
    main block;
    http {
        server {
        listen 80 default_server;
        server_name *.nigelpoulton.com;
        root /var/www/nigelpoulton.com;
        index index.html
            location / {
                root /usr/share/nginx/html;
                index index.html;
            }
        }
}
```

Once the data is stored in a `ConfigMap` it can be injected into containers at run time via any of the following methods

- Environment variables
- Arguments to the container's startup command
- Files and volumes

All of the methods work seamlessly with existing app, in fact all an app sees is its configuration data in either - env variable, an argument or a file on the filesystem. The app is unaware the data originally came from a `ConfigMap` or a Secret type of object. The most flexible of the three methods is the volume option, whereas the most limited are the startup command.

A K8s native app is one that knows its running on k8s and can talk to the k8s API. As a result they can access `ConfigMap` data directly via the API without needing things like environment variables, startup arguments or variables and volumes. This can simplify app configuration, but the app will only run in k8s environment.

```
$ kubectl create configmap testmap1 \
    --from-literal shortname=AOS \
    --from-literal longname="Agents of Shield"

$ kubectl describe cm testmap1
```

```
Name:         testmap1
Namespace:    default
Labels:       <none>
Annotations:  <none>
Data
====
longname:
----
Agents        of Shield
shortname:
----
AOS
Events:       <none>
```

Creating The most basic example, creating a new config map object directly from the command line, without using any manifest files, this is solely to demonstrate how they can be created, but in the real world, the preferred creation method is always the YAML manifest way.

```
kind: `ConfigMap`
apiVersion: v1
metadata:
  name: multimap
data:
  given: Nigel
  family: Poulton
```

Here is a proper example, which uses the regular manifest file to create a new `ConfigMap`, notice that the config map is given a name, and also that the data section, specifies the number of key value pairs which are representing the data this config map provides.

```
kubectl apply -f multimap.yml
```

The manifest below uses the special pipe characters, that tells the k8s api to treat everything after the pipe as a literal value, meaning that the actual value of the key `config` is the multi line text, which we can see after the pipe character, that is cool since this is a good way to represent values which can be later on mounted as files form the config map. This is often used to inject complex configurations, such as json files, or even shell scripts

```
kind: `ConfigMap`
apiVersion: v1
metadata:
name: test-conf
data:
    config: |
        env = plex-test
        endpoint = 0.0.0.0:31001
        char = utf8
        vault = PLEX/test
        log-size = 512M
```

Injecting Binding the `ConfigMap` to a pod and injecting it is the next step. Looking at each of the 3 methods to inject `ConfigMaps`, and their pros and cons

Environment variables A common way to get `ConfigMap` data into a container is via environment variables. You create the `ConfigMaps` then you map its entries int environment variables in the container section of a Pod template. When the container is started the environment variables appear in the container as standard Linux or Windows environment variables, we use a combination of `configMapKeyRef`, where the name of the key and the name of the source `ConfigMap` must be provided

```
apiVersion: v1
kind: Pod
metadata:
    name: envpod
spec:
    containers:
        - name: ctr1
          image: busybox
          command: ["sleep"]
          args: ["infinity"]
          env:
              - name: FIRSTNAME
                valueFrom:
                    configMapKeyRef:
                        name: multimap
                        key: given
              - name: LASTNAME
                valueFrom:
                    configMapKeyRef:
                        name: multimap
                        key: family
```

When the Pod is schedules and the container started, the `FIRSTNAME` and `LASTNAME` will be created as standard Linux environment variables inside the container. Apps can use these like regular variables because they are. Run the following to deploy the pod from the manifest above,

```
$ kubectl appy -f envpod.yml

$ kubectl exec envpod -- env | grep NAME
HOSTNAME  = envpod
FIRSTNAME = Nigel
LASTNAME  = Poulton
```

A drawback to using `ConfigMaps` with environment variables is that environment variables are static, This means that updates made to the map are not reflected in running containers. For example if you update the values of the given and family entries in the `ConfigMap` environment variables in existing containers will not see those updates. This is a major reason environment variables are not very good

Startup arguments This concept is simple, you specify a startup command for a container, and then customize it with variables, the following pod template is showing that, how a single container is called with `args1`. Also if you play a close attention you will notice that this is also using the environment variables approach.

```
apiVersion: v1
kind: Pod
metadata:
    name: startup-pod
    labels:
        chapter: configmaps
spec:
    restartPolicy: OnFailure
    containers:
        - name: container
          image: busybox
          # that is the startup command for the container, in other words the
              ENTRYPOINT from Dockerworld
          command: ["/bin/sh", "-c", "echo First name $(FIRSTNAME) last name
             $(LASTNAME)", "wait"]
          env:
              - name: FIRSTNAME
                valueFrom:
                    configMapKeyRef:
                        name: multimap
                        key: given
              - name: LASTNAME
                valueFrom:
                    configMapKeyRef:
                        name: multimap
                        key: family
```

Volumes & Files Using the `ConfigMaps` with volumes is the most flexible option. You can reference entire configuration files as well as make updates to the `ConfigMap` and have them reflected in running containers. This means you can make changes to entries in the `ConfigMap` after you have deployed a container and those changes be seen in the container and available for running apps. The updates may take a minute or so to appear in the container. But how is the `ConfigMap` exposed as a volume you may ask ?

Well first what is the process of creating such a config map, that would be simply creating a `ConfigMap` object, afterwards this `ConfigMap` object is bound to or as a volume to a Pod, and mounted into the container. Entries in the `ConfigMap` would appear in the container is individual files

```
apiVersion: v1
kind: Pod
metadata:
    name: cmvol
spec:
    volumes:
        - name: volmap # this is the name of the volume and we have bound a
           value to it below
          configMap:
              name: multimap # the bound value and here we are re-using the
                 multimap we have created
    containers:
        - name: container # name the container, for the pod
          image: nginx # tell the container what image to use
```

```
        volumeMounts: # specifies a list of volumes to the pod
            - name: volmap # tell the pod which volume to use
              mountPath: /etc/name # tell the pod where to exactly mount
                 this volume
```

Now what would the effect of this be, the values inside the `multimap ConfigMap` are two key value pairs. The config map has two keys given and family, these will be used to create the names of the files, while the values for these keys will be used to populate the files, therefore the contents of the files will be Nigel and Poulton respectively. This is quite powerful since we can define the configuration for any application and any changes to the config map will correctly reflect the changes in the file.

Secrets

Secrets are almost identical to `ConfigMaps`, they hold app configuration data that is injected into containers at run-time. However Secrets are designed for sensitive data such as passwords, certificates and OAuth tokens.

Security & Secrets Are secrets really secure, The quick answer to this question is no, but there is a slightly longer answer. Despite being designed for sensitive data, kubernetes does not encrypt Secrets. It merely obscures them as base 64 encoded values that can easily be decoded, fortunately it is possible to configure encryption at rest with `EncryptionConfiguration` objects, and most service meshes encrypt network traffic. A typical workflow for Secret is as follow.

1. The secret is created and persisted to the cluster store as an un-encrypted object
2. A pod that uses it gets scheduled to a cluster node
3. The secret is transferred over the network un-encrypted to the node
4. The kubelet on the node starts the Pod and its containers
5. The Secret is mounted into the container via an in memory tmpfs file system and decoded from base64 to plain text
6. The app consumes it
7. When the pod is deleted the secret is deleted from the node.

While it is possible to encrypt the secret in the cluster store and leverage a service mesh to encrypt it in flight on the network, it is always mounted as plain text in the Pod, container. This is so the app can consume it without having to perform decryption or base64 decoding operations. Also the use of in memory tmpfs file systems mean they are never persisted to disk on a node. So to cut a long story short, no secrets are not very secure, but you can take extra steps to make them secure. They are also limited to 1MB of size. An obvious use case for Secrets is a generic TLS termination proxy for use across your dev, test and prod, environments. You create a standard image, and load the appropriate TLS keys at run time for each environment.

By default every pod gets a secret mounted into it as a volume which it uses to authenticate itself if it talks to the API server, if we take a look at any running pod in our cluster we will be able to use the describe command from kubectl and we can actually see the `defulat-token-s9nmx` which is a secret value that is mounted into the container automatically by the kubectl so it can communicate with the API server, that is simple API token, to call the REST endpoints with.

```
$ kubectl describe pod <podname>

Name:                                                cmvol
Namespace:                                           default
<Snip>
Containers:
ctr:
```

```
Container                                              ID: containerd://0
   de32d677251cbbda3ebe53e8...
Image:                                                 nginx
Mounts:
 /etc/name                                             from volmap (rw)
 /var/run/secrets/kubernetes.io/serviceaccount from default-token-s9nmx (ro)
 <Snip>
 Volumes:
 default-token-s9nmx:
 Type:                                                 Secret (a volume populated by
    a Secret)
 SecretName:                                           default-token-s9nmx
 Optional:                                             false
 QoS                                                   Class: BestEffort
<Snip>
```

Also note where this is mounted, /var/run/secrets/kubernetes.io/, this is a common way for k8s to prefix its own resources with kubernetes.io, in this case the secret is mounted under /var/run/secrets and the kubernetes specific folder

```
$ kubectl get secrets

NAME                  TYPE                                  DATA  AGE
default-token-s9nmx kubernetes.io/service-account-token 3     21d
```

You might get an output similar to this one, which will show the list of currently active secrets, one of which will be the actual certificate we, attached to the Pod itself, and used by the Pod to communicate with the REST server of the control plane.

```
$ kubectl describe secret default-token-s9nmx

Name:                                  default-token-s9nmx
Namespace:                             default
Labels:                                <none>
Annotations:                           kubernetes.io/service-account.name:
   default
kubernetes.io/service-account.uid: c5b5a4b3-3c5c...
Type:                                  kubernetes.io/service-account-token
Data
====
    token:                                 eyJhbGciOiJSUzI1NiIsIm...
    ca.crt:                                570 bytes
    namespace:                             7 bytes
```

Making secrets As we have already seen the secrets are not encrypted in the cluster store (etcd) they are not encrypted in flight, not encrypted on the network and not encrypted when surfaced in a container. There are ways to encrypt them, but at the end of the day the container has to have a way to read these secrets in plain text, either by decrypting them itself, or having the secret decrypted when delivered or mounted into the container. Now by default when we create secrets in the manifest file, the contents of the manifest file or more precisely the `data` section of the manifest must contain the base64 version of the secret or data

```
apiVersion: v1
```

```
kind: Secret
metadata:
    name: tkb-secret
    labels:
        chapter: configmaps
type: Opaque
data: # provide a base64 encoded data, the api server will accept it as is
    username: bmlnZWxwb3VsdG9u
    password: UGFzc3dvcmQxMjM=
# stringData: # provide the raw data, the api server will base64 encode it
   itself
#   username: myusername
#   password: mypassword
```

```
# to apply the secret manifest from above
$ kubectl apply -f secret.yml
```

Here in this example the `data` contains values that are already encoded in base64, this is crucial to remember because, the `data` field does not contain a value that is not encoded in base64, the API server will reject the secret, meaning posting the manifest will fail, to provide the raw un-encoded data in raw plain text one should use the `stringData` field, instead of the `data` one.

```
# to decode the value and see the actual value
$ echo <base-64-encoded-vlaue> | base64 -d
```

Using secrets The most flexible way to inject a secret into a pod (container) is via a special type of volume called a secret volume. The following YAML describe a single container Pod with a Secret volume called "secret-vol" based on the secret manifest from above, created in the previous step.

```
apiVersion: v1
kind: Pod
metadata:
    name: secret-pod
    labels:
        topic: secrets
spec:
    volumes:
        - name: secret-vol
          secret:
              secretName: tkb-secret
    containers:
        - name: secret-ctr
          image: nginx
          volumeMounts:
              - name: secret-vol
                mountPath: /etc/secret
                readOnly: true
```

The main difference between the `ConfigMap` type volumes and the Secret type volumes is that the Secret volumes are automatically mounted as read only, and also the data in these volumes is decoded from base64 to base plain text before hand, so they can be mounted as the actual value the base64 encoding is representing. To apply this pod we do the usual

```
# create a pod where the secret is mounted under /etc/secret
$ kubectl apply -f secret-pod.yml
```

StatefulSet

For the purposes of this discussion a stateful app is one that creates and saves valuable data, an example might be an app that saves data about client sessions and uses it for future sessions. Other examples include databases and other data stores. `StatefulSet` are the logical equivalent of Deployments in the K8s world. While the spec structure is similar, there are some differences:

- `StatefulSet`:
- Includes fields like `serviceName` (to associate with a headless service).
- Supports `volumeClaimTemplates` for creating `PersistentVolumeClaims` (PVCs) for each pod.
- Does not support the strategy field for updates (it always uses a rolling update strategy).
- `Deployment`:
- Includes a strategy field to define update strategies (e.g., RollingUpdate or Recreate).
- Does not have `serviceName` or `volumeClaimTemplates`.

Theory

It is often useful to compare the `StatefulSets` with Deployments both are first class API objects and follow the typical kubernetes controller architecture, they are both implemented as controller that operate reconciliation loops watching the state of the cluster, via the API server and moving the observed state into sync with the desired state. Deployments and `StatefulSets` also support self healing scaling updates and more. However there are some vital differences between `StatefulSets` and Deployments. `StatefulSets` guarantee:

- Predictable and persistent Pod names
- Predictable and persistent DNS hostnames
- Predictable and persistent volume bindings

These three properties form the sate of Pod, sometimes referred to as its sticky ID. `StatefulSets` ensure this state/id is persisted across failures scaling and other scheduling operations making them ideal for apps that require unique pods that are not interchangeable.

A quick example failed Pods managed by a `StatefulSet` will be replaced by new Pods with the exact same Pod name the exact same DNS hostname, and the exact same volumes. This is true even if the replacement pod is started on a different cluster node. The same is not true of Pods managed by a Deployment.

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
    name: tkb-sts
spec:
    replicas: 2
    selector:
        matchLabels:
            app: mongo
    serviceName: "tkb-sts"
    template:
```

```
    metadata:
        labels:
            app: mongo
    spec:
        containers:
            - name: ctr-mongo
              image: mongo:latest
```

The name of the `StatefulSet` is `tkb-sts` and it defines three pod replicas running the `mongo:latest` image. You post this to the API server it is persistent to the cluster store the replicas are assigned to cluster nodes and the `StatefulSet` controller monitors the state of the cluster making sure observed state matches the desired state, that is the big picture, let us take a closer look at some the major characteristics of `StatefulSet` before walking through an example

Naming

All Pods managed by a `StatefulSet` get predictable and persistent names. These names are vital and are the core of how Pods are started self healed, scaled, deleted and attached to volumes and more. The format of the `StatefulSet` Pod names is `<StatefulSetName>-<Integer>`. The integer `is a zero based index ordinal which is just a fancy way of saying number starting from 0`. The first Pod created by a `StatefulSet` always get index ordinal 0, and each subsequent Pod gets the next highest. Assuming the previous YAML snippet the first Pod created will be called `tks-sts-0`, the second will be called `tsb-sts-1`. Be ware that `StatefulSet` names need to be a valid DNS names so no exotic characters are allowed, this rule and its reasons will come in play later.

Creation

Another fundamental characteristic of the `StatefulSet` is that controlled and ordered way they start and stop Pods. `StatefulSet` create one Pod at a time, and always wait for previous Pods to be running and ready before creating the next. This is different than from Deployments that use a `ReplicaSet` controller to start all Pods at the same time, causing potential race conditions.

As per the previous YAML snippet `tkb-sts-0` will be started first and must be running and ready before the `StatefulSet` controller starts `tkb-sts-1`. The same applies to subsequent Pods - `tkb-sts-1` needs to be running and ready before `tkb-sts-2` starts and so on.

Scaling operations are also governed by the same ordered startup rules. For example scaling from 3 to 5 replicas will start a new Pod called `tkb-sts-3` and wait for it to be in the running and ready state, before creating `tkb-sts-4` Scaling down follows the same rules in reverse, the controller terminates the Pod with the highest index ordinal (number) first waits for it to fully terminate before terminating the Pod with the next highest ordinal.Knowing the order in which Pods will be scaled down as well knowing that Pods will not be terminated in parallel is a game changer for many stateful apps. For example clustered apps that store data can potentially lose data if multiple replicas go down at the same time. `StatefulSet` guarantee this will never happen You can also inject other delays via things like `terminationGracePeriodSeconds`, to further control the scaling down process. All in all `StatefulSets` bring a lot to the table for clustered apps that create and store data.

Finally it is worth noting that `StatefulSets` controllers do their own self healing and scaling, this is architecturally different to Deployments which use a separate `ReplicaSet` helper controller for these operations

`All in all StatefulSet provide a robust predictable and easy to manage control flow over the deployment lifecycle of pods. This is crucial for services which require more fine grained control over their own lifecycle and other dependent components`

Deleting

There are two major things to consider when deleting a `StatefulSet` object,

Firstly deleting a `StatefulSet` does not terminate Pods in order, With this in mind you may want to scale a `StatefulSet` to 0 replicas before deleting it You can also use the `terminationGracePeriodSeconds` to further control the way Pods are terminated. It is common to set this to at least 10 seconds to give apps running in the Pods a chance to flush local buffers and safely commit any writes that are still in-flight

Volumes

Volumes are an important part of the `StatefulSet` Pod stick ID and state. When `StatefulSet` Pod is created any volumes it needs are created at the same time and named in a special way that connects them to the right Pod. That means that each volume created by the PVC. Volumes are appropriately decoupled from Pods via the normal Persistent Volume Claims system. This means volumes have separate lifecycle to Pods, allowing them to survive Pod failures and termination operations. For example any time a `StatefulSet` Pod fails or is terminated associate volumes are unaffected. This allows replacement Pods to attach to the same storage as the Pods they are replacing. This is true even if replacement Pods are schedule to different cluster nodes.

The same is true for scaling operations if a `StatefulSet` Pod is deleted as part of a scale down operation subsequent scale up operations will attach new Pods to the surviving volumes that match their names, this behavior can be a life saver if you accidentally delete a `StatefulSet` Pod especially if it is the last replica.

So here is something to take a good note of, of how `Deployments` and `StatefulSets` differ in their usage of the persistent volumes and persistent volume claims:

If you have a Deployment, every replica of the Deployment is identical, aside from its name. In particular, every replica will share the same `PersistentVolumeClaim` and the same underlying `PersistentVolume`.

Conversely, each replica of a `StatefulSet` gets its own `PersistentVolumeClaim`, assuming you use the `volumeClaimTemplates`: field to declare the PVC. If anything causes the `StatefulSet` to scale up, the new Pod will get a new empty `PersistentVolume`. If it scales down, the `PersistentVolume` is preserved, and if it scales up again, the previous `PersistentVolume` is reused.

```
Important to note how the names of the unique PVC objects are generated when using the
volumeClaimTemplates, since each of the pvc objects is created for each pod replica, the
 name is generated from the name defined under volumeClaimTemplates, in the StatefulSet
spec, and using the ordinal number of the pod it is attached to.
```

Handling Failures

The `StatefulSet` controller observes the state of the cluster and attempts to keep observed state in sync with the desired state. The simplest example is a Pod failure, if you have a `StatefulSet` called `tkb-sts` with 5 replicas, and `tkb-sts-3` fails, the controller will start a replacement Pod with the same name and attach it to the same volumes it was already attached to. However if a failed Pod recovers after Kubernetes has replaced it you will have two identical Pods trying to write to the same volume. This can result in data corruption. As a result the `StatefulSet` controller is extremely careful how ti handles failures.

Possible node failures are very difficult to deal with. For example, if Kubernetes loses contact with a node, how does it know if the node has failed / is down and will never recover or if its temporary glitch, such as network partition a crashed kubelet or the node is simply rebooting. To complicate matters further the controller can not even force the Pod to terminate as the local kubelet may never receive the instruction to do that. With all of this in ind, manual intervention is needed before K8s will replace Pods on failed nodes.

```
Unlike deployments, where once the Pod is considered failed, it will never be re-started
 or ever re instated back into the k8s environment, meaning that a brand new one will be
 created and the old one will be completely de commissioned, in the StatefulSet world,
that is not the case a pod might actually recover so the stateful state controller has
to be more conservative when managing failed Pods, unlike its Deployment and ReplicaSet
counterpart
```

Services

We have already said that `StatefulSet` are for apps that need Pods to be predictable and long lived as a result other parts of the app as well as other apps may need to connect directly to individual Pods. To make this possible `StatefulSet` use a headless service to create predictable DNS names for every pod replica they manage. Other apps can then query DNS service for the pull list of Pod replicas and use these details to connect directly to the Pods. The following YAML snippet shows a headless Service called `mongo-prod` that is listed in the `StatefulSet` YAML.

```
apiVersion: v1
kind: Service
metadata:
    name: mongo-prod
spec:
    clusterIP: None
    selector:
        app: mongo
        env: prod
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
    name: sts-mongo
spec:
    serviceName: mongo-prod
```

Let us explain the terms headless Service and governing Service. A headless Service is just a regular K8s service object without an IP address (see that the `ClusterIP` is not set, or rather is set to none). This tells K8s that this service will and must not receive any virtual `ClusterIP` address. It become a `StatefulSet` governing service when you list it in the `StatefulSet` manifest under `spec.serviceName`. When the two objects are combined like this the Service will create `DNS SRV` records for each Pod replica that matches the label selector of the headless Service. Other Pods and apps can then find members of the `StatefulSet` by performing DNS lookup against the name of the headless Service.

Network traffic

Unlike regular services, the ones represented by `StatefulSet`, will not have a virtual `ClusterIP` address, the headless service,

Skeleton

The snippets below represent the general skeleton and structure of a deployment stage for a `StatefulSet` , which includes the creation and definition of all components that tie into the `StatefulSet` - like PVC, `StorageClasses`, headless Services and more.

First lets create the most low level structure - `StorageClass`, that will be the entry point for the PVC later, even though most cloud providers do have default SC objects, it is good to see how it all ties together, of course in the real world you will end up using the SC provided by the cloud provider most likely.

Nothing fancy, the `StorageClass` below, just defines that the store is of type SSD, which is using the plugin - `pd.csi.storage.gke.io`. This is specific to the google cloud provider services, but it will be pretty similar for other providers as well, in this case the `StorageClass` represents a basic fast storage - backed by a solid state drive

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
    name: flash
provisioner: pd.csi.storage.gke.io
volumeBindingMode: WaitForFirstConsumer
allowVolumeExpansion: true
parameters:
    type: pd-ssd
```

Here is the headless service definition, we know it is headless by the fact that the `ClusterIP` here is set to `none`, this is pretty much the only difference with the regular k8s service objects

```
apiVersion: v1
kind: Service
metadata:
    name: dullahan
    labels:
        app: web
spec:
    ports:
        - port: 80
          name: web
    clusterIP: None
    selector:
        app: web
```

Finally the `StatefulSet`, this one, has a bit more meat to it. Here we can see some of the most prominent differences compared to the Deployment object, like the `terminationGracePeriodSeconds` fields, also notice that most of the spec of the Pod in the `template` section is pretty much the same as in the Deployment object, the `template` section as we know already defines the properties of the Pod object that would be created by the k8s environment

Now take a look at the most interesting part that differs significantly from the way we define Pod templates in the Deployments, that is `volumeClaimTemplates`, you may notice that the format and spec of this section matches perfectly with the spec of a `PersistentVolumeClaim` object, that is because it is, however since we have already mentioned above, we know that the `PersistentVolumeClaim` for `StatefulSet` have to be bound to Pod instances, unlike with Deployments' Pods.

That is why the PVC is defined in the spec of the `StatefulSet`, that way for each new replica of the Pod, a new PVC will be created and by proxy a new fresh volume which will be independent of the volumes created for the other PVCs and Replicas, the PVC will be bound to the name of the Pod. That is why it is also important that the name of the pod template - `spec.serviceName` to be DNS format compliant

```
apiVersion: apps/v1
```

```
kind: StatefulSet
metadata:
    name: tkb-sts
spec:
    replicas: 3
    selector:
        matchLabels:
            app: web
    serviceName: "dullahan"
    template:
        metadata:
            labels:
                app: web
        spec:
            terminationGracePeriodSeconds: 10
            containers:
                - name: ctr-web
                  image: nginx:latest
                  ports:
                      - containerPort: 80
                        name: web
                  volumeMounts:
                      - name: nginx-data
                        mountPath: /usr/share/nginx/data
                      - name: config-data
                        mountPath: /usr/share/nginx/config
    volumes:
        - name: config-data
          persistentVolumeClaim:
              claimName: config-data
    volumeClaimTemplates:
        - metadata:
              name: nginx-data
          spec:
              accessModes: ["ReadWriteOnce"]
              storageClassName: "flash"
              resources:
                  requests:
                      storage: 1Gi
```

Note that we can still use the regular way to defined volumes for `StatefulSet`, the regular way being the one supported by the Deployment object, however these will not be bound to the underlying Pods as with stateful services, what would be the use case for the basic volume and PVC definition ? Well we could bind read only configuration data for example, that does not require stateful and Pod bound volumes. It is just important to keep in mind that the `volumeClaimTemplates` DOES NOT REPLACE the regular method of defining PVCs for pods through the `spec.volumes` section

Given the `volumeClaimTemplates` form above and the name for that defined in `metadata.name`, `nginx-data`, the names of the PVC objects that will be created under the hood and tied to the Pod replicas will use the base name `nginx-data-<ordinal>`, the ordinal number will be the same as for the pod the PVC is bound to. Remember that the name of the Pods are actually generated from the name of the `StatefulSet`, and also

appended an ordinal value which represents their replication order or in other words ID. That means that a pod with an ordinal or 3 was the 4th Pod replica to be started, remember that these ordinals start from 0.

The key takeaway here is to really understand that unlike Deployments, the `StatefulSet` are really tied, tightly coupled with the underlying service which exposes the Pods created by the `StatefulSet`, that can be seen in the fact that the headless service name is actually present in the spec of the `StatefulSet`, unlike with Deployments where the interlinking part is the Pod, between a Deployment and a Service, here the Service is directly linked to the `StatefulSet` expressed in the property `spec.serviceName`, that is because other elements of the `StatefulSet` are also tied in to the name of the headless service - like the name resolution of the dynamic PVC objects that are created for each Pod replica in the `StatefulSet`

Finally the DNS names that would be resolvable for each Pod will be generated using the `StatefulSet` name along with the service name and the ordinal of the Pod, in this case, having the example above we will have the following DNS names, `tbk-sts-0.dullahan.default.svc.cluster.local`, and so on - the general structure that the DNS name will follow is `tbk-sts-<n>.dullahan.default.svc.cluster.local`, where `<n>` is going to be the ordinal index of the Pod replica starting from 0, the `tbk-sts` is the name of the `StatefulSet` object, and the `dullahan` is the name of the headless service

Security

Kubernetes is API centric and the API is served through the API server, below we will inspect how a typical API request to the API server is processed on the control plane, and what security measures does the kubernetes API server take to make sure that no unauthorized parties access the API. Through the use of Role Based Access Control (RBAC)

Theory

All of the following make CRUD style requests to the API server - operators and developers using the `kubectl`, Pods, Kubelets, Control plane service & controllers and more. The usual flow that the request follow is that - of a subject (user,group) -> api server -> authentication -> authorization -> admission. Consider a quick example where a user called "grant" is trying to create a Deployment object called "hive" in the namespace "terran". User "grant" issues a kubectl command to create the Deployment. This generates a request to the API server, with the user's credential embedded, thanks to the magic of TLS the connection between the client and the API server is secure. The authentication module determines whether its grant-ward or an impostor. After that the authorization module (RBAC) determines whether grant-ward is allowed to create Deployments in the "terran" Namespace. If the request passes authentication and authorization admission control checks and applies policies and the request is finally accepted and executed.

It is a lot like flying on a plane, You travel to the airport and authenticate yourself with a Photo Id, usually your passport. You then present a ticket authorizing you to board the plane and occupy a particular seat. If you pass authentication and are authorized to board admission controls may then check and apply airline policies such as not taking hot food on board restricting your hand luggage and prohibiting alcohol in the cabin. After all of that you are finally allowed to board the plane and take your set.

Authentication

Authentication is about providing your identity. You might see or her it shortened to `authN`. At the heart of authentication are credentials. All requests to the API server have to include credentials, and the authentication layer is responsible for verifying them. If verification fails the API server returns an HTTP 401 error and the request is denied. If it passes it moves on to authorization.

The authentication layer in Kubernetes is pluggable and popular modules include client certs, `webhooks` and integration with external identity management systems, such as Active Directory (AD) and cloud based

Identify Access Management (IAM). In fact it is impossible to create user accounts in Kubernetes as it does not have its own built in identity database, instead Kubernetes forces you to use an external system, this is great as Kube does not install yet another identity management silo.

Out of the box most Kubernetes clusters support client certificates but in the real world you will want to integrate with your chosen cloud or corporate identity management system. Many of the hosted Kubernetes services make it easy to integrate with their native identity management systems.

Authentication

Cluster Cluster details and credentials are stored in a `kubeconfig` file. Tools like `kubectl` read this file to know which cluster to send commands to as well as which credentials use, it is usually stored in the following locations:

- Windows: `C: \Users\<user>\.kube\config`
- Unix: `/home/<user>/.kube/config`

Many Kubernetes installations can automatically merge cluster endpoint details and credentials into your existing `kubeconfig`, for example every `GKE` cluster provides a `gcloud` command that will merge the necessary cluster details and credentials to your local `kubeconfig` config file. The following is an example do not try and run it. `gcloud container clusters get-credentials tkb --zone europe-west1-c --project <project>`.

Here is what a `kubeconfig` file looks like. As you can see it defines a cluster and a user, combines them into a context and sets the default context for all `kubectl` commands

```
apiVersion: v1
kind: Config
clusters:
    - cluster:
      name: prod-shield
      server: https://<url-or-ip-address-of-api-server>:443
      certificate-authority-data: …LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0LS0tCg
         ==
users:
    - name: <username>
      user:
          as-user-extra: {}
          token: …eyJhbGciOiJSUzI1NiIsImtpZCI6IlZwMzlSZY3uUQ
contexts:
    - context:
      name: shield-admin
      cluster: prod-shield
      namespace: default
      user: <username>
current-context: shield-admin
```

You can see it is divided into 4 top level sections, so what is the structure of this file:

The `clusters` section defines one or more Kubernetes clusters, each one has a friendly name and API server endpoint, and the public key of its certificate authority. The cluster in the example is exposing the secure API endpoint on port 443, but it is also common to see it exposed on 6443

The `users` section defines one or more users. Each user requires a name and token. The token is often a X.509 encrypted value which is user's ID.

The `contexts` section combines users and clusters and the current-context is the cluster and user `kubectl` will use for all commands

Assuming the previous `kubeconfig` all `kubectl` commands will go to the prod-shield cluster and authenticate as the "" user. The authentication module is responsible for determining if the user really is `<username>`, and if using client certificates it will determine if the certificate is signed by a trusted CA.

Authorization

Authorization happens immediately after successful authentication and you will sometimes see it shortened to `authZ`, Kubernetes authorization is pluggable and you can run multiple authorization modules on a single cluster. As soon as any of the modules authorization requests is made, it moves on to admission control.

Access control The most common authorization module is RBAC - role based access control. At the highest level it is all about three things - users, actions and resources. Which users can perform which actions against which resources in the cluster. The following table shows a few examples.

User	Action	Resource
Bao	create	Pods
Kalila	list	Deployments
Josh	delete	ServiceAccounts

RBAC is enabled on most Kubernetes clusters and has been stable in general availability since Kubernetes 1.8. It is a least privilege deny by default system. This means all actions are denied by default and you enable specific actions by creating an allow rule. In fact Kubernetes does not support any deny rules, it only supports allow rules. This might seem like a small thing, but it makes the Kubernetes world and RBAC much simpler to implement and troubleshoot, and safer.

Two concepts are vital to understanding Kubernetes `RBAC` - `Roles` and `RoleBindings`:

`Roles` define a set of permissions and `RoleBindings` grant those permissions to users. The following resource manifest defines a Role object it is called read deployments and grants permissions to get, watch, and list Deployments objects in the shield namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
    namespace: shield
    name: read-deployments
rules:
    - apiGroups: ["apps"]
      resources: ["deployments"]
      verbs: ["get", "watch", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
    name: read-deployments
    namespace: shield
subjects:
    - kind: User
      name: sky # This is the authenticated user, that was given the role
```

```
      apiGroup: rbac.authorization.k8s.io
roleRef:
    kind: Role
    name: read-deployments # This is the Role from above to bind to the user
    apiGroup: rbac.authorization.k8s.io
```

If both of these are deployed to a cluster an authenticated user called sky will be able to run commands such as `kubectl get deployments -n shield` It is important to understand that the username listed in the `RoleBindings` has to be a string and has to match the username that was successfully authenticated.

Roles & Resources The previous `Role` object has three properties, - `apiGroups, resources and verbs` . Together these define which actions are allowed against which objects, `apiGroups` and `resources` define the object, and `verbs` define the actions. The example allows read access (get, watch, and list) against Deployment objects. The following table shows `apiGroups` and resources combinations.

```
apiGroup          resource      Kubernetes API path""
                 pods          /api/v1/namespaces/{namespace}/pods""
                 secrets       /api/v1/namespaces/{namespace}/secrets"
storage.k8s."io storageclass /apis/storage.k8s.io/v1/storageclasses""
apps             deployments  /apis/apps/v1/namespaces/{namespace}/deployments
```

An empty set of double quotes `""` in the `apiGroup` field indicates the core `apiGroup`, all other api sub-groups need specifying as a string enclosed in double quotes. The `apiGroup` is important since it namespaces the different kubernetes resources and have to be specified for a given resource that belongs to the given `apiGroup`, unless it is part of the default one which is represented as already mentioned by the empty quotes `""`. Kubernetes uses a standard set of verbs to describe the actions a subject can perform on a resource. Verb names are self explanatory and case sensitive, the following table lists them and demonstrates the REST based nature of the API by showing how they map to HTTP methods, it also lists some common HTTP response codes.

HTTP method	Kubernetes verbs	Common responses
POST	create	201 created, 403 Access Denied
GET	get, list and watch	200 OK, 403 Access Denied
PUT	update	200 OK, 403 Access Denied
PATCH	patch	200 OK, 403 Access Denied
DELETE	delete	200 OK, 403 Access Denied

The kubernetes verbs column lists the verbs you use in the rules section of a Role object. Running the following command shows all API resources supported on your cluster. It also shows API group and supported verbs and is a great resource for helping build rule definitions

```
$ kubectl api-resources --sort-by name -o wide
NAME                        SHORTNAMES  APIVERSION                     NAMESPACED
    KIND                       VERBS
                                                                  CATEGORIES
apiservices                             apiregistration.k8s.io/v1      false
        APIService                 create,delete,deletecollection,get,list,
   patch,update,watch  api-extensions
bindings                                v1                             true
         Binding                    create
```

```
certificatesigningrequests csr         certificates.k8s.io/v1       false
        CertificateSigningRequest create,delete,deletecollection,get,list,
   patch,update,watch
clusterrolebindings                     rbac.authorization.k8s.io/v1 false
        ClusterRoleBinding        create,delete,deletecollection,get,list,
   patch,update,watch
clusterroles                            rbac.authorization.k8s.io/v1 false
        ClusterRole               create,delete,deletecollection,get,list,
   patch,update,watch
deployments                deploy       apps/v1                      true
         Deployment                create,delete,deletecollection,get,list,
   patch,update,watch  all
ingresses                  ing          networking.k8s.io/v1         true
         Ingress                   create,delete,deletecollection,get,list,
   patch,update,watch
pods                       po           v1                           true
         Pod                       create,delete,deletecollection,get,list,
   patch,update,watch  all
replicasets                rs           apps/v1                      true
         ReplicaSet                create,delete,deletecollection,get,list,
   patch,update,watch  all
replicationcontrollers     rc           v1                           true
         ReplicationController     create,delete,deletecollection,get,list,
   patch,update,watch  all
resourcequotas             quota        v1                           true
         ResourceQuota             create,delete,deletecollection,get,list,
   patch,update,watch
roles                                   rbac.authorization.k8s.io/v1 true
         Role                      create,delete,deletecollection,get,list,
   patch,update,watch
secrets                                 v1                           true
         Secret                    create,delete,deletecollection,get,list,
   patch,update,watch
services                   svc          v1                           true
         Service                   create,delete,deletecollection,get,list,
   patch,update,watch  all
statefulsets               sts          apps/v1                      true
         StatefulSet               create,delete,deletecollection,get,list,
   patch,update,watch  all
storageclasses             sc           storage.k8s.io/v1            false
        StorageClass              create,delete,deletecollection,get,list,
   patch,update,watch
```

This table represents the abridged version of the original, which contains many more objects which the kubernetes environment manages by default, the table above only lists most of the objects which were already looked at so far, and also those being one of the most important ones

Also take a note of the `SHORTNAMES` column which is an alias for the actual resource name, all resources which define a `SHORTNAME` can be referred by it in all contexts such as manifest files, shell commands and so on.

And if we take a good look with the role manifest definition above, we can clearly see that the rules, objects, verbs and resources match with the table output above such as

```
rules:
    - apiGroups: ["apps"]
      resources: ["deployments"]
      verbs: ["get", "watch", "list"]
```

To refer to all objects one can use the asterisk (*), that would bind the verbs to all objects which are exposed by the Kubernetes environment, For example the following rule block grants all actions on all resources in every API group, (basically cluster admin). It is just for demonstration purposes and you would probably never want to do this in the actual world, or in actual production grade Kubernetes clusters

```
rules:
    - apiGroups: ["*"]
      resources: ["*"]
      verbs: ["*"]
```

Cluster roles So far we have seen how to create `Roles` and `RoleBindings`. However, Kubernetes actually has 4 RBAC objects, `Roles, ClusterRoles, RoleBindings and ClusterRoleBindings`.

`Roles` and `RoleBindings` are namespaced objects. This means they can only be applied to a single Namespace, `ClusterRole` and `ClusterRoleBindings` are cluster wide objects and apply to all Namespaces. All 4 are defined in the same API sub group and their YAML structures are almost identical.

A powerful pattern is to define Roles at the cluster level and bind them to a specific Namespace via `RoleBinding`. This lets you define common roles once and re use them across multiple Namespaces. For example the following YAML defines the same read-deployments role but this time at the cluster level, this can be re-used in selected Namespaces via a regular `RoleBinding`

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
    name: read-deployments
rules:
    - apiGroups: ["apps"]
      resources: ["deployments"]
      verbs: ["get", "watch", "list"]
```

Look closely at the previous YAML manifest, the only difference with the earlier one is that the kind here is defined as `ClusterRole`, instead of a Role, and it does not have a metadata.namespace property, since there is no meaning in namespacing object in this case the cluster role that should be cluster wide.

Existing resources As you might have figured out, there is a set of pre-defined Roles and Bindings, granting permissions to an all powerful user. Many will also configure `kubectl` to operated under the context of that user. The following example walks you through the pre-defined and pre-created user roles and bindings, on a docker desktop cluster the names of Pods and RBAC objects will be different on other clusters, but the principles will be the same and it gives you an idea of how things are made to work.

Mnikube runs API server in Pod in the kube-system namespace. It has an –authorization flag that tells Kubernetes which authorization modules to use. The following command shows the node and RBAC modules are both enabled.

```
$ kubectl get pods -n kube-system
NAME                               READY   STATUS    RESTARTS      AGE
coredns-668d6bf9bc-mtpmm           1/1     Running   1 (23h ago)   23h
etcd-minikube                      1/1     Running   1 (23h ago)   23h
```

```
kube-apiserver-minikube             1/1     Running   1 (23h ago)   23h
kube-controller-manager-minikube    1/1     Running   1 (23h ago)   23h
kube-proxy-z9gf9                    1/1     Running   1 (23h ago)   23h
kube-scheduler-minikube             1/1     Running   1 (23h ago)   23h
storage-provisioner                 1/1     Running   3 (22h ago)   23h

$ kubectl describe pod/kube-apiserver-minikube -n kube-system | grep
   authorization
--authorization-mode=Node,RBAC
```

You wont be able to interrogate the API server like this on a hosted Kubernetes cluster, This is because critical control plane features like this are hidden from you. `Minikube` also updates your kube config files with the cluster called `minikube`, here is how one config file might look like for `Minkube` that would be located at `$HOME/.kube/config`

```
apiVersion: v1
clusters:
    - cluster:
          certificate-authority: /home/asmodeus/.minikube/ca.crt
          extensions:
              - extension:
                    last-update: Sat, 08 Mar 2025 16:27:39 EET
                    provider: minikube.sigs.k8s.io
                    version: v1.35.0
                name: cluster_info
          server: https://127.0.0.1:60703
      name: minikube
contexts:
    - context:
          cluster: minikube
          extensions:
              - extension:
                    last-update: Sat, 08 Mar 2025 16:27:39 EET
                    provider: minikube.sigs.k8s.io
                    version: v1.35.0
                name: context_info
          namespace: default
          user: minikube
      name: minikube
current-context: minikube
kind: Config
preferences: {}
users:
    - name: minikube # Here is the name of the user that is created
      user:
          client-certificate: /home/asmodeus/.minikube/profiles/minikube/
             client.crt # client credentials
          client-key: /home/asmodeus/.minikube/profiles/minikube/client.key #
              client credentials
```

Note that we can also see the client's certificate and the private key, by client that would imply the client that interfaces with the API server, in our case that is `kubectl`, this is mandatory to be able to contact the

API server, especially on non local cluster deployments, this provides the authorization step, in this case this allows the API server to establish that the client instance connecting to it is authorized to talk to it

RABC is enabled and a user kubeconfig is created, the user is called minikube, now let us take a look at the cluster role and the cluster role bindings, objects which are pre-configured to grant permissions to that user. Here are some of the cluster role bindings which are bound to the ClusterRole named cluster-admin, these are not all.

```
$ kubectl get clusterrolebindings | grep cluster-admin
NAME                                                   ROLE

   AGE
cluster-admin                                          ClusterRole/
   cluster-admin                                                23h
minikube-rbac                                          ClusterRole/
   cluster-admin                                                23h

$ kubectl describe clusterrolebindings minikube-rbac
Name:         minikube-rbac
Labels:       <none>
Annotations:  <none>
Role:
  Kind:  ClusterRole # the cluster role type
  Name:  cluster-admin # the name of the cluster role
Subjects:
  Kind            Name     Namespace
  ----            ----     ---------
  ServiceAccount  default  kube-system
```

As a result of these bindings all commands in a default on premise minikube installation are executed with the cluster-admin permissions. This might be OK for development environments and on premise testing environments, but certainly not okay for production ones. How let us see what exactly is defined for this cluster role in its manifest file

```
$ kubectl describe clusterrole cluster-admin
Name:         cluster-admin
Labels:       kubernetes.io/bootstrapping=rbac-defaults
Annotations:  rbac.authorization.kubernetes.io/autoupdate: true
PolicyRule:
  Resources  Non-Resource URLs  Resource Names  Verbs
  ---------  -----------------  --------------  -----
  *.*        []                 []              [*]
             [*]                []              [*]
```

As you can see it allowed everything, for every resource and every verb or action that can be performed on the cluster, which is normal for that type of environment as mentioned already.

Authorization ensures that already authenticated users are allowed to carry out the actions they are attempting. RBAC is a popular Kubernetes authorization module and implements least privilege access based on a deny by default model where all actions are assumed to be denied unless a rule exists that allows it, The model is similar to a whitelist firewall where everything is blocked and you open up access by creating allow rules. Kubernetes RBAC uses Roles and ClusterRole to create permissions and it uses

RoleBinding and ClusterRoleBinding to grant those permissions to users, in other words to bind them to the users, these bindings objects serve as a middle man to allow us to create different permutations and combinations of role objects to grant to users, this way we can achieve the most granular access granting and permission model

Admission control

Admission control runs immediately after successful authentication and authorization, and it is all about policies there two types of admission controllers - Mutating and Validating. The names are self explanatory, Mutating controllers check for compliance and can modify requests whereas validating controllers check for policy compliance but cannot modify requests Mutating controllers always run first and both types only apply to requests that will modify state. Requests to read state are not subjected to admission control.

Assume a quick example where all new and updated objects to your cluster must have the env-prod label. A mutating controller can check for the presence of the label and add it if it does not exist, on the flip side a validating controller can only reject the request if it does not exist. The following command on a `minikube` cluster shows the API server is configured to use the `NodeRestriction` admission controller, amongst others

```
$ kubectl describe pod/kube-apiserver-minikube -n kube-system | grep
   admission
--enable-admission-plugins=NamespaceLifecycle,LimitRanger,ServiceAccount,
   DefaultStorageClass,DefaultTolerationSeconds,NodeRestriction,
   MutatingAdmissionWebhook,ValidatingAdmissionWebhook,ResourceQuota
```

Most real world clusters will have a lot more admission controllers enabled. There are lots of admission controllers but the `AlwaysPUllImage` controller is a great example. It is a mutating controller that sets the `spec.containers.imagePullPolicy` of all new Pods to always. This means that images for all containers in all Pods will always be pulled form the registry. This accomplishes quite a few things, including the following

- Preventing the use of locally cached images that could be malicious
- Preventing other Pods and processes using locally cached images
- Forcing the container runtime to present valid credentials to the registry to get the image

If any admission controller rejects a request the request is immediately rejected without checking other admission controllers. However if all admission controllers approve the request it gets persisted to the cluster store, As previously mentioned there are lots of admission controllers and they are becoming more and more important in real world production clusters

General Summary

The authentication layer is responsible for validating the identity of the request. Client certificates are commonly used to integration with `AD` and other `IAM` services is recommended for production clusters. Kubernetes does not have its own identity database, meaning it does not store or manage user accounts or credentials.

The authorization layer checks whether the authenticated user is authorized to carry out the action in the request. This layer is also pluggable and the most common module is RBAC. The RBAC module comprises 4 objects that define permissions and assigns them to the users.

Admission control, kicks in after the client/user is authenticated and authorized and is responsible for enforcing policies. Validating admission controller reject request if htey do not conform to the policy, where as mutating admissions controllers can modify the incoming request to the API server, in other words to mutate the incoming object manifest to enhance or override the structure of the document according to some policies,

This section will describe a few fast and quick ways to obtain Kubernetes. Will also introduce you to `kubectl`, the Kubernetes command line tool.

Kubernetes API

Understanding the Kubernetes API, and how it works is vital to mastery of Kubernetes. However it can be extremely confusing, if you are new to the API.

Theory

As already seen Kubernetes is API centric, the entire functionality is based around interacting with the API server, through the user level client, in this case `kubectl`. This means that everything in Kubernetes is about the API. And everything goes through the API server. We will get into the details soon, but for now let us just look at the big picture.

Clients `send` request to kubernetes to create, read, update and delete objects such as Pods and Services and so on. For the most part you will use `kubectl` to send these requests however, you can craft them in code or use the API testing and development tools to generate them The point is no matter how you generate requests they go to the API sever where they are authenticated and authorized, assuming they pass these checks they are executed on the cluster. If a create request is posted, the object is deployed to the cluster and the serialized state of it is persisted to the cluster store (etcd).

Serialization

Kubernetes serializes objects such a Pods and Services, as JSON strings to be send over HTTP. The process happens in both directions with clients like `kubectl` serializing the object when posting to the api server and the api server serializing the response back to clients. In the case of Kubernetes the serialized state of objects are also persisted to the cluster store which is usually based on the etcd database service

So in Kubernetes serialization is the process of converting an object to into a JSON string to be sent over an HTTP connection and persisted to the cluster store. However as well as JSON Kubernetes also supports Protobuf as a serialization schema. This is faster and moreefficient and scales better than JSON. But it is not user friendly, when it comes to introspection and troubleshooting. At the time of writing Protobuf is mainly sued for internal cluster traffic, whereas son is used when communicating with external clients.

One final thing on serialization, when clients send requests to the API server they use the content type head to list the serialization schema they support. For example a client that only supports JSON will specify `Content-Type: application/json` in the HTTP header of the request. Kubernetes will honour this with a serialized response in JSON

The API server

The API server exposes the API over a secure RESTful, interface using HTTPS. It acts as the front end to the API and is a bit like Grand Central for Kubernetes everything talks to everything else via the REST API calls to the API server.

- ALL `kubectl` commands go to the API server (creating, retrieving, updating deleting objects)
- ALL node `kubelets` watch the API server for new tasks and report status to the API server
- ALL control lane services communicate via the API server, NOT directly to each other

The API server is a Kubernetes control plane service. This usually means that it runs as a set of Pods in the kube-system Namespace on the control plane nodes of your cluster. If you build and manage your own Kubernetes clusters you need to make sure the control plane is highly available and has enough performance to keep the API server up and running and responding quickly to requests. If you are using the hosted Kubernetes cluster the way the API server is implemented including the performance and availability will be hidden away from you.

The main job of the API server is to make API available to clients inside and outside the cluster. It uses TLS to encrypt the client connection and i leverages of a bunch o authentication and authorization mechanisms to ensure only valid requests are accepted and actioned upon. Requests from internal and external sources all have to pass through the same authentication and authorization.

The API is RESTful. This is a jargon for a modern web API that accepts CRUD style requests via standard HTTPS methods. Every standard HTTP method like PUT, DELETE, UPDATE, POST can be mapped to a corresponding verb in the Kubernetes world as far as role verbs are concerned.

It is common for the API server to be exposed on port 443 or 6443 but its possible to configure it to operate on whatever port you require, running the following command shows the address and port of your Kubernetes cluster is exposed on

```
$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:60703
CoreDNS is running at https://127.0.0.1:60703/api/v1/namespaces/kube-system/
   services/kube-dns:dns/proxy
To further debug and diagnose cluster problems, use 'kubectl cluster-info
   dump'.
```

A few words on how the RESTful nature of the client maps to the KUBERNETES API server. REST is short for - `REpresentational State Transfer` and its the de-facto standard for communicating with web based API. System such as Kubernetes that use REST are often referred to as RESTful. REST requests comprise a verb and a path to a resource. Verbs related to actions and are the standard HTTP methods you saw in the previous chapters. The following example shows a `kubectl` command and the associated REST path that will list all pods in the shield namespace.

```
kubectl get pods --namespace shield
```

```
GET /api/v1/namespaces/shield/pods
```

To visualise this, start a `kubectl` proxy and use curl to generate the request. The `kubectl` proxy command exposes the api on your localhost adapter and takes care of the authentication process. You can use a different port.

```
$ kubectl proxy --port 9000 & # start the proxy process in the background

$ curl http://localhost:9000/api/v1/namespaces/shield/pods
{
    "kind": "PodList",
    "apiVersion": "v1",
    "metadata": {
    "resourceVersion": "484812"
    },
    "items": []
}
```

The example returned an empty list because there are no pods in that namespace. Responses from the API server include common HTTP response codes content type and the actual payload, as you learned earlier the chapter, Kubernetes uses JSON as its preferred content type, as a result the previous `kubectl` get command will result in an HTTP 200 (OK) response code, the content type will be application/json and the payload will be serialized to JSON, list all Pods in the shield Namespace, Run one of the previous curl commands again but with the flag -v - which sends for verbose, and that will list the headers which are being sent and received from and to the API server.

```
$ curl -v http://localhost:9000/api/v1/namespaces/shield/pods
> GET /api/v1/namespaces/shield/pods HTTP/1.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/json
< X-Kubernetes-Pf-Flowschema-Uid: 499d0001-d874-4b06-ba...c37f7
< X-Kubernetes-Pf-Prioritylevel-Uid: aeb490e6-1890-41ab...94e82
<
{
    "kind": "PodList",
    "apiVersion": "v1",
    "metadata": {
    "resourceVersion": "487845"
    },
    "items": []
}
```

The API

The API is where all Kubernetes resources are defined, it is large, modular. When Kubernetes was originally created teh API was monolithic in design with all resources existing in a single global namespace, however as kubernetes grew it became necessary to divide the API into smaller more manageable groups, these are `core, apps, rbac and netowrking`

The core

The resources in the core group are mature objects that were created in the early days of Kubernetes before the API was divided into groups. They tend to be fundamental objects such as Pods, Nodes, Services, Deployments, Secrets and so on. The are located in the API below /api/v1 (now it should be clear what /v1 refers to, the core api, the one that Kubernetes started with). The following table lists some example paths for resources in the core group, you will sometimes see and hear these paths referred to as REST path.

Resources	Path
Pods	/api/v1/namespaces/{namespace}/pods/
Services	/api/v1/namespaces/{namespace}/services/
Nodes	/api/v1/nodes/
Namespaces	/api/v1/namespaces/

Notice that some objects are namespaced and some are not Namespaced objects have a longer REST paths as you have to include two additional name segments, For example listing all pods in the shield namespace requires the following path

```
GET /api/v1/namespaces/shield/pods/
```

Expected HTTP response code from read requests are either 200, or 401. On the topic of these rest paths, GVR stand for group-version-resource and can be a good way to remember the structure of the REST paths, in the Kubernetes API

Named Groups

The named API groups are the future of the API, these all new resources go into named groups, Sometimes we refer to them as sub-groups. Each of the named groups is a collection of related resources. For example the apps groups is where all resources that manage apps workloads such as `Deployments ReplicaSets, DaemonSets, StatefulSets` are defined. Likewise, the `networking.k8s.io` group is where `Ingresses and Ingress Classes`, Networking policies and other network related resources exist, notable exceptions to this pattern are older resources in the core groups that came along afterwards, like Pods were they invented today, would probably go in the apps groups, and Services would go in the networking.k8s.io, group. Resources in the named groups live below the /apis/{group-name}/{version}/path. The following table lists some examples

Resource	Path
Ingress	/apis/networking.k8s.io/v1/namespaces/{namespace}/ingresses/
RoleBinding	/apis/rbac.authorization.k8s.io/v1/namespaces/{namespace}/rolebindings/
ClusterRole	/apis/rbac.authorization.k8s.io/v1/clusterroles/
StorageClass	/apis/storage.k8s.io/v1/storageclasses/

Notice how the URI paths for named groups start with /apis and include the name of the groups. This is different to the core group that starts with /api in the singular and does not include a group name, in fact in places you will see the core API groups referred to by empty double quotes. This is because when the API was first designed no thought was given to groups - everything was just in the API. Dividing the API into smaller groups makes it more scalable and easier to navigate it also makes it easier to extend. The following command are good for seeing API related info for your cluster. `kubectl api-resources` is great for seeing which resources are available on your cluster as well on which API groups they are served from. It also shows resource short names and whether objects are namespaced or cluster scoped. The command `kubectl api-versions` will on the other hand show you what API versions are supported on your cluster, it does not list which resources belong to which API it is good for finding out whether you have things such as alpha, APIS enabled or not.

```
$ kubectl api-resources
NAME                              SHORTNAMES                          APIVERSION
                NAMESPACED KIND
bindings                          v1                                  true
                      Binding
--------------------------------------------------------------------------------


configmaps                        cm                                  v1
                        true       ConfigMap
endpoints                         ep                                  v1
                        true       Endpoints
events                            ev                                  v1
                        true       Event
limitranges                       limits                              v1
                        true       LimitRange
namespaces                        ns                                  v1
                        false      Namespace
nodes                             no                                  v1
                        false      Node
persistentvolumeclaims            pvc                                 v1
                        true       PersistentVolumeClaim
```

```
persistentvolumes                 pv                                   v1
                        false      PersistentVolume
pods                              po                                   v1
                        true       Pod
podtemplates                      v1                                   true
                      PodTemplate
replicationcontrollers            rc                                   v1
                        true       ReplicationController
resourcequotas                    quota                                v1
                        true       ResourceQuota
secrets                           v1                                   true
                      Secret
serviceaccounts                   sa                                   v1
                        true       ServiceAccount
services                          svc                                  v1
                        true       Service
--------------------------------------------------------------------------------

ingressclasses                    networking.k8s.io/v1                 false
                     IngressClass
poddisruptionbudgets              pdb                                  policy/v1
                  true       PodDisruptionBudget
clusterrolebindings               rbac.authorization.k8s.io/v1         false
                     ClusterRoleBinding
clusterroles                      rbac.authorization.k8s.io/v1         false
                     ClusterRole
rolebindings                      rbac.authorization.k8s.io/v1         true
                      RoleBinding
roles                             rbac.authorization.k8s.io/v1         true
                      Role
priorityclasses                   pc                                   scheduling.
   k8s.io/v1    false      PriorityClass
csidrivers                        storage.k8s.io/v1                    false
                     CSIDriver
csinodes                          storage.k8s.io/v1                    false
                     CSINode
csistoragecapacities              storage.k8s.io/v1                    true
                      CSIStorageCapacity
storageclasses                    sc                                   storage.k8s
   .io/v1       false      StorageClass
volumeattachments                 storage.k8s.io/v1                    false
                     VolumeAttachment
```

```
$ kubectl api-versions
admissionregistration.k8s.io/v1
apiextensions.k8s.io/v1
apiregistration.k8s.io/v1
apps/v1
authentication.k8s.io/v1
authorization.k8s.io/v1
autoscaling/v1
```

```
autoscaling/v2
batch/v1
certificates.k8s.io/v1
coordination.k8s.io/v1
discovery.k8s.io/v1
events.k8s.io/v1
flowcontrol.apiserver.k8s.io/v1
networking.k8s.io/v1
node.k8s.io/v1
policy/v1
rbac.authorization.k8s.io/v1
scheduling.k8s.io/v1
storage.k8s.io/v1
v1
```

This one command can be quote useful when one would like to inspect the different kind and version fields of api resources, supported on your cluster, the output is trimmed but it should give the general idea of what one might receive as output

```
$ for kind in `kubectl api-resources | tail +2 | awk '{ print $1 }'`; \
            do kubectl explain $kind; done | grep -e "KIND:" -e "VERSION:"
KIND:     Namespace
VERSION:  v1
KIND:     Node
VERSION:  v1
---------------------------------
KIND:     HorizontalPodAutoscaler
VERSION:  autoscaling/v1
KIND:     CronJob
VERSION:  batch/v1beta1
KIND:     Job
VERSION:  batch/v1
---------------------------------
```

The command could be used on actual self hosted and no premise hosted Kubernetes cluster and services. This could turn out to be useful for debugging or auguring information on resources and their source API

Thread modeling

Thread modeling is the process of identifying vulnerabilities so you can put measures in place to prevent and mitigate them. This section will introduce the popular `STRIDE` model and shows how it can applied to Kubernetes. `STRIDE` defines the six categories of potential thread. The word `STRIDE` is an abbreviation which stands for the following:

- Spoofing
- Tampering
- Repudiation
- Information disclosure
- Denial of service
- Elevation of privilege

While the model is good it is important to keep in mind that it is just a model, and models do not guarantee to cover all possible threats possible. However they are a good at providing a structured way to look at things,

for the rest of the section, we will take a look at each of the six thread categories in turn, and how we can prevent and mitigate them.

In 2019 the `CNCF` (Cloud Native Computing Foundation) commissioned a third party security audit of Kubernetes. There were several findings including threat modeling manual code reviews, dynamic penetration testing and cryptography review. All findings were given a difficulty and severity level, and all high severity

Spoofing

Spoofing is pretending to be somebody else with the aim of gaining extra privilege on a system. Kubernetes is comprised of lots of small components that work together, these include control plane service such as the API server, controller manager scheduler cluster store, and others. It also includes Node components such as the kubelet and container runtime. Each of these has its own set of privileges that allow it to interact with and even modify the cluster, even though Kubernetes implements a least privilege model, spoofing the identity of any of these can cause problems

If you read the RBAC and API security section, you will know that Kubernetes requires all components to authenticate, via a cryptographically signed certificates. This is good and Kubernetes makes it easy to auto rotate certificates and the likes. However it is vital you consider the following.

1. A typical Kubernetes installation will auto generate a self signed certificate authority (CA). This is the CA that will issue certificates to all cluster components. And while its better than nothing on this own its probably not enough for production environments
2. Mutual TLS, is only as secure as the CA issuing the certificates, compromising the CA can render the entire `mTLS` layer ineffective, so keep the CA secure.

A good practice is to ensure that certificates issued by the internal Kubernetes CA are only used and trusted within the Kubernetes cluster. This requires careful approval of certificate signing requests and you need to make sure the Kubernetes CA does not get added as a trusted CA for any system outside of Kubernetes

As mentioned in previous sections all internal and external requests to the API server are subject to authentication and authorization checks, as a result in API server needs a way to authenticate internal and external sources, a good way to do this is having two trusted key pairs:

- one for authenticating internal systems
- the other for authenticating external systems

In this model you would use the cluster's self signed CA to issue keys to internal systems, you would also configure Kubernetes to trust one or more trusted 3rd party CAs to issue keys to external systems.

As well as spoofing access to the cluster there is also the threat of spoofing an app for app-to-app communications. This is when one Pod spoofs another. Fortunately you can leverage Secrets to mount certificates into the Pods that are used to authenticate Pod identity.

While on the topic of Pods every Pod has an associated `ServiceAccount` that is used to provide an identity, for the Pod within the cluster. This is achieved by automatically mounting a service account token into every Pod as a Secret. Two points to note:

1. The service account token allows access to the API server
2. Most Pods probably do not need to access the API server

With these two points in mind, it is often recommended to set `automountServiceAccountToken` to false for Pods that you know do not need to communicate wit the API server. The following Pod manifest shows how to do this. ###

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

```
metadata:
    name: service-account-example-pod
spec:
    serviceAccountName: some-service-account
    automountServiceAccountToken: false
<Snip>
```

If the Pod does need to talk to the API server, the following non default configurations are worth exploring. - `expiratoinSeconds` and `audience`. These two let you force a time when the token will expire as well as restrict the entities in works with. The following example inspired from official Kubernetes docs sets and expiry period of one hour and restricts it to the vault audience in a projected volume.

```
apiVersion: v1
kind: Pod
metadata:
    name: nginx
spec:
    containers:
        - image: nginx
    name: nginx
    volumeMounts:
        - mountPath: /var/run/secrets/tokens
          name: vault-token
          serviceAccountName: my-pod
volumes:
    - name: vault-token
      projected:
          sources:
              - serviceAccountToken:
                path: vault-token
                expirationSeconds: 3600
                audience: vault
```

Tampering

Tampering is the act of changing something in a malicious way so you can cause one of the following, denial of service, and elevation of privilege. Tampering can hard to avoid so a common counter measure is to make it obvious when something has been tampered with. A common example outside of information security is packaging medication. Most over the counter drugs are packaged with tamper proof seals. jthese make it easy to see if the product has been tampered with. Let us have a quick look at some of the cluster components that can be tampered with.

Kubernetes components All of the following Kubernetes components, if tampered with can cause harm or issues - `etcd`, configuration files, container runtime binaries, container images, kubernetes binaries and more. Generally speaking tampering happens either in transit or at rest. In transit refers to data while it is being transmitted over the network, where as at rest refers to data stored in memory or on disk. TLS is a great tool for protecting against in transit tampering, as it provides built in integrity guarantees you will be warned if the data has been tampered with. The following recommendations can also help prevent tampering with data when it is at rest in Kubernetes

. Restrict access to the servers that are running Kubernetes components i.e the control plane.

- Restrict access to repositories that store Kubernetes configuration files
- Only perform remote bootstrapping over SSH
- Restrict access to your image repository and associated repositories

This is not and exhaustive list, but if you implement it you will greatly reduce the changes of having your data tampered with while at rest. As well as the items listed it is goog production hygiene to configure auditing and alerting for important binaries and config files. If configured and monitored correctly these can help detect potential tampering attacks. The following example uses a common Linux audit daemon to audit access to the docker binary it also audits attempts to the change the binary file attributes

```
sh $ auditctl -w /usr/bin/docker -p wxa -k audit-docker
```

Kubernetes applications As well as infrastructure components, apps components are also potential tampering targets. A good way to prevent live Pod from being tampered with is setting its filesystems to read only. This guarantees filesystem immutability and can be accomplished through a Pod Security Policy or the `securityContext` section of a Pod manifest file.

You can make a container root filesystem read only by setting the `readOnlyFilesystem` property, As previously mentioned, this can be set via a `PodSecurityPolicy`, object or in Pod manifest files. The same can be done for other filesystems that are mounted into containers, via the `allowedHostPaths` property

The following YAML manifest shows how to use both settings in a Pod manifest spec, the `allowedHostPaths` section makes sure anything mounted beneath /test will be read only. The two examples represent the same functionality, one is simply a separate object represented by the `PodSecurityPolicy`, which is relatively new in the Kubernetes spec and that is why it is not under `v1`, the stable `apiVersion`, but rather under a new beta version api path `policy/v1beta1`. In the future when the spec of that object is cleaned up, it will be put under the `policy/v1` `apiVersion` path

```
apiVersion: v1
kind: Pod
metadata:
    name: readonly-test
spec:
    securityContext:
        readOnlyRootFilesystem: true
        allowedHostPaths:
            - pathPrefix: "/test"
              readOnly: true
---
apiVersion: policy/v1beta1 # Will change in a future versions, when the
   object spec gets out of the beta stage
kind: PodSecurityPolicy
metadata:
    name: tampering-example
spec:
    readOnlyRootFilesystem: true
    allowedHostPaths:
        - pathPrefix: "/test"
          readOnly: true
```

Repudiation

At a very high level Repudiation is creating doubt about something. Non Repudiation is providing proof about something. In the context of information security non Repudiation is proving certain actions were carried out by certain individuals. Digging a little deeper non-repudiation includes the ability to provide

- `What` happened
- `When` it happened
- `Who` made it happen
- `Where` is happened
- `Why` is happened
- `How` it happened

Answering the last two usually requires the correlation of several events over a period of time fortunately auditing of Kubernetes API server events can usually help answer these questions the Following is an example of an API server audit event (you may need to manually enable auditing on your API server).

```
{
    "kind": "Event",
    "apiVersion": "audit.k8s.io/v1",
    "metadata": { "creationTimestamp": "2020-03-03T10:10:00Z" },
    "level": "Metadata",
    "timestamp": "2020-03-03T10:10:00Z",
    "auditID": "7e0cbccf-8d8a-4f5f-aefb-60b8af2d2ad5",
    "stage": "RequestReceived",
    "requestURI": "/api/v1/namespaces/default/persistentvolumeclaims",
    "verb": "list",
    "user": {
        "username": "fname.lname@example.com",
        "groups": ["system:authenticated"]
    },
    "sourceIPs": ["123.45.67.123"],
    "objectRef": {
        "resource": "persistentvolumeclaims",
        "namespace": "default",
        "apiVersion": "v1"
    },
    "requestReceivedTimestamp": "2010-03-03T10:10:00.123456Z",
    "stageTimestamp": "2020-03-03T10:10:00.123456Z"
}
```

Although the API server is central to most things in Kubernetes it is not the only components that requires auditing for non repudiation, at a minimum you should collect audit logs from container runtimes, kubelets and the apps running on your cluster. This is without even mentioning network firewalls and the likes.

Once you start auditing multiple components you quickly need a centralized location to store and correlate events, a common way to do this is deploying an agent to all nodes via a `DeamonSet`. The agent collects logs (runtime, kubelet, apps…)

If you do this it is vital the centralized log store is secure, if the security of the store is compromised you can no longer trust the logs, and their contents can be repudiated. To provide non repudiation relative to tampering with binaries and configuration files it might be useful to use an audit daemon that watches for write actions on certain files and directories on your Kubernetes Masters and Nodes. For example earlier in the chapter you saw an example that enabled auditing of changes to the docker binary, with this enabled

starting a new container with the docker run command will generate an event like this:

```
type=SYSCALL msg=audit(1234567890.123:12345): arch=abc123 syscall=59 success=
   yes exit=0 a0=12345678abc\
a1=0 a2=abc12345678 a3=a items=1 ppid=1234 pid=12345 auid=0 uid=0 gid=0 euid
   =0 suid=0 fsuid=0 egid=0 \
s\
gid=0 fsgid=0 tty=pts0 ses=1 comm="docker" exe="/usr/bin/docker" subj=
   system_u:object_r:container_runt\
ime_exec_t:s0 key="audit-docker"
type=CWD msg=audit(1234567890.123:12345): cwd="/home/firstname"
type=PATH msg=audit(1234567890.123:12345): item=0 name="/usr/bin/docker"
   inode=123456 dev=fd:00 mode=0\
100600 ouid=0 ogid=0 rdev=00:00 obj=system_u:object_r:
   container_runtime_exec_t:s0
```

Audit logs like this when combined and correlated with Kubernetes audit features create a comprehensive and trustworthy picture that can not be repudiated.

Information Disclosure

Information disclosure is when sensitive data is leaked, there are lots of ways it can happen including hacked data stores and API that unintentionally expose sensitive data.

Protecting cluster data In the Kubernetes world the entire configuration of the cluster is stored in the cluster store (usually etcd). This includes network and storage configuration as well as passwords and other sensitive data in Secrets. For obvious reasons this makes the cluster store a prime target for information disclosure attacks. As a minimum you should limit and audit access to the nodes hosting the cluster store, as will be seen in the next paragraph, gaining access to the cluster node can allow the logged-on user to bypass some of the security layers

Kubernetes 1.7 introduced encryption of Secrets but does not enable it by default. Even when this becomes default, the data encryption key (DEK) is stored on the same node as the Secret. This means gaining access to a node lets you to bypass encryption completely. This is especially worrying on nodes that host the cluster store (etcd nodes).

Fortunately Kubernetes 1.11 enabled a beta feature that lets you store key encryption keys (KEK), outside of your Kubernetes cluster. These types of keys are used to encrypt and decrypt data encryption keys and should be safely guarded. You should seriously consider Hardware security modules (HSM) or cloud based key management stores (KMS) for storing your encryption keys. Keep an eye on upcoming versions of Kubernetes for further improvements of Secrets.

Protecting data in Pods As previously mentioned, Kubernetes has an API resource called a Secret that is the preferred way to store and share sensitive data such as passwords. For example a front end container accessing an encrypted back end database can have the key to decrypt the database mounted as a secret. This is far better solution than storing the decryption keys in plain text file or environment variable.

It is also common to store data and configuration information outside of Pods and containers in persistent volumes and config maps. If the data on these is encrypted keys for decrypting them should also be stored in Secrets. With all of this in mind it is vital that you consider the caveats outlined in the previous section relative to Secrets and how their encryption keys are stored. You do not want to do the hard work of locking the house but leaving the key on the door

Denial of Service

Denial of Service is all about making something unavailable. There are many types of `DoS` attacks, but a well known variation is overloading a System to the point it can no longer service requests. In the Kubernetes world a potential attack might be to overload the API server so that cluster operations grind to a halt, even essential system services have to communicate via the API server

Protecting cluster resources It is a time honored best practice to replicate essential control plane service on a multiple nodes for a high availability. Kubernetes is no different and you should run multiple Masters in an HA configuration, for your production environments. Doing this prevents a single Master from becoming a single point of failure. In relation to certain types of denial of service attacks, an attacker may need to attack more than one Master to have a meaningful impact. You should also consider replicating control plane nodes across availability zones. This may prevent a denial of service attack on the network of a particular availability zone from taking down your entire control plane. The same principle applies to worker nodes. Having multiple worker nodes not only allows the scheduler to spread your app over multiple nodes and availability zones, it may also render DOS attacks on a single node or zone ineffective (or less effective). You should also configure appropriate limits for the following:

- Memory
- Processor
- Storage
- Kubernetes objects

Placing limits on things can help prevent important systems resources from being starved therefore preventing potential denial of service attacks. Limiting Kubernetes objects includes things like limiting the number of `ReplicaSets`, Pods, Services, Secrets and `ConfigMaps` in particular Namespace. Here is an example manifest file/snippet that limits the number of Pod objects in the `skippy` namespace to 100

```
apiVersion: v1
kind: ResourceQuota
metadata:
    name: pod-quota
spec:
    hard:
        pods: "100"
```

There is another feature also called `podPidsLimit`, which restricts the number of processes a Pod can create. Assume a scenario where a Pod is the target of a fork bomb attack, this is a specialised attack where a rogue process creates as many new processes as possible in an attempt to consume all resources on a system and grind it to a halt. Placing a limit on the number of processes a Pod can create will prevent the Pod from exhausting the resources of the node and confine the impact of the attack to the Pod. Once the `podPidsLimit` is exhausted a Pod will typically be restarted.

Protecting the API server The API server exposes a RESTful interface over a TCP socket making it susceptible to botnet based denial of service attacks, the following may be helpful in either preventing or mitigating such attacks.

- Highly available masters. Having multiple API server replicas running on multiple nodes across multiple availability zones
- Monitoring and alerting API server requests based on sane thresholds
- Using things like firewalls to limit API server exposure to the internet.

As well as botnet DOS attacks an attacker may also attempt to spoof a user or other control plane service in an attempt to cause an overload. Fortunately, Kubernetes has robust authentication and authorization controls to prevent spoofing. However even with a robust RBAC model, it is vital that you safeguard access to accounts with high privileges.

Protecting the cluster store Cluster configuration is stored in etcd, making it vital that etcd be available and secure. The following recommendations help accomplish this:

- Configure an HA etcd cluster with either 3 or 5 nodes
- Configure monitoring and alerting of requests to etcd
- Isolate etcd at the network level so that only members of the control plane can interact with it

A default installation of Kubernetes installs etcd on the same servers as the rest of the control plane. This is usually fine for development and testing however large production clusters should seriously consider a dedicated etcd, cluster. This will provide better performance and greater resilience. On the performance front etcd is probably the most common choking point for large Kubernetes clusters. With this in mind, you should perform testing to ensure the infrastructure it runs on is capable of sustaining performance at scale, a poorly performing etcd can be as bad as an etcd cluster under a sustained attack. Operating a dedicated etcd cluster also provides additional resilience by protecing it from other parts of the control plane that might be compromised. Monitoring and alerting etcd should be based on sane thresholds and a good place to start is by monitoring etcd log entries.

Protecting application components Most Pods expose their main service on the network and without additional controls in place anyone with access to the network can perform a DOS attack on the POD. Fortunately, Kubernetes provides Pod resource requests limits to prevent such attacks from exhausting Pod and Node resources, As well as these the following will be helpful.

- Define Kubernetes Network Policies to restrict Pod to Pod and Pod to external communications
- Utilize mutual TLS and API token based authentication for application level authentication reject any unauthenticated requests

Elevation of privilege

Privilege escalation is gaining higher access than what is granted usually in order to cause damage or gain unauthorized access. Let us look at a few way to prevent this in a Kubernetes environment.

Protecting the API server Kubernetes offers several authorization modes, that help safeguard access to the API server. These include - RBAC, Webhook, Node. You should run multiple authorizers at the same time. For example a common best practice is to always have RBAC and node enabled.

RBAC mode lets you restrict API operations to sub sets of users. These users can be regular users accounts as well as system services. The idea is that all requests to the API server must be authenticated and authorized. Authentication ensures that requests are coming from the validated user, whereas authorization ensures that validated user is allowed to perform the requested operation. For example, can Lily create Pods ? In this example Lily is the user, create is the operation, and Pods is the resource. Authentication makes sure that it really is Lily that is making the request and authorization determines if she is allowed to create Pods.

Webhook mode lets you offload authorization to an external REST based policy engine. However it requires additional effort to build and maintain the external engine. It also makes the external engine a potential single point of failure for every request, to the API server. For example if the external webhook system becomes unavailable you may not be able to make any requests to the API server. With this in mind, you should be rigorous in vetting and implementing any webhook authorization service

Protecting Pods The next few sections will look at a few of the technologies that help reduce the risk of elevation of privilege attacks against Pods and containers, we will look at the following - preventing processes from running as root, dropping capabilities filtering syscalls.

Root processes The root user is the most powerful user on a Linux system and it always User ID 0. Therefore running application processes as root is almost always a bad idea as it grants the app process full access to the container. This is mad even worse by the fact that the root user of container often has unrestricted root access on the host system as well. Fortunately Kubernetes lets you force container processes to run as unprivileged non root users. The following Pod manifest configures all containers that are part of this Pod to run processes as UID 1000. If the Pod has multiple containers all processes in all containers will run as UID 1000.

```
apiVersion: v1
kind: Pod
metadata:
    name: demo
spec:
    securityContext: # Applies to all containers in this Pod
        runAsUser: 1000 # Non-root user
containers:
    - name: demo
      image: example.io/simple:1.0
```

The `runAsUser` is one of the many settings that can be configured as part of what we refer to as `PodSecurityPolicy`. It is possible for two or more Pods to be configured with the same `runAsUser`. When this happens, the containers from both Pods will run with the same security context and potentially have access to the same resources. This might be fine if they are replicas of the same Pod or container. However there is a high chance this will cause problems if they are different containers. For example two different containers with R/W access to the same host directory or volume can cause data corruption bot writing to the same dataset without coordinating write operations). Shared security contexts also increase the possibility of compromised container tampering with a dataset it should not have access to

With this in mid, it is possible to use the `securityContext.runAsUser` property at the container level instead of the Pod level:

```
apiVersion: v1
kind: Pod
metadata:
name: demo
spec:
    securityContext: # Applies to all containers in this Pod
        runAsUser: 1000 # Non-root user
containers:
    - name: demo
      image: example.io/simple:1.0
      securityContext:
          runAsUser: 2000 # Overrides the Pod setting
```

This example sets the UID to 1000 at the pod level, but overrides it at the container level, so that processes in one particular container run a UID 2000. Unless otherwise specified all other containers in the Pod will use UID 1000. A couple of other things that might help get around the issue of multiple Pods and containers using the same UID include - enabling user namespaces, maintaining a map of UID usage.

User namespaces is a Linux kernel technology that allows a process to run as root within a container, but run as a different user outside of the container. For example a process can run as UID 0 (the root user) in the container). But get mapped to UID 1000 on the host. This can be a good solution for processes that need to run as root inside the container but you should check it has full support from your version of Kubernetes and your container runtime. Maintaining a map of UID usage is a clunky way to prevent multiple different Pods and containers using overlapping UID, it is a bit of a hack and requires strict adherence to a gated release process for releasing Pods into production.

Drop capabilities While user namespaces allow container processes to run as root inside the container but not on the host machine it remains a fact that most processes do not really need to run as full root inside the container. However it is equally true that many processes do require more privileges than a typical non root user. What is needed is a way to grant the exact set of privileges to process requires in order to run. Enter capabilities.

Time for some theory and background history. We have already said that the root user is the most powerful user on the Linux system. However its power is a combination of lots of small privileges that we call capabilities. For example the `SYS_TIME`, capability allows a user to set the system clock, whereas the NET_ADMIN capability allows a user to perform network related operations such as modifying the local routing table and configuring local interfaces, the root user holds every capability and is therefore extremely powerful.

Having a modular set of capabilities like this allows you to be extremely granular when granting permissions. Instead of an all or nothing (root or non-root) approach you can grant a process the exact set of capabilities it requires to run.

There are currently over 30 capabilities and choosing the right ones can be daunting. With this in mind, an out of the box Docker runtime drops over half of them by default. This is a sensible default that is designed to allow most processes to run, without leaving the keys in the front door. While sensible defaults like these are better than nothing they are often not good enough for a lot of production environments.

A common way to find the absolute minimum set of capabilities an app requires is to run it in a test environment, with all capabilities dropped. This will cause the app to fail and log messages about the missing permissions. You map those permissions to capabilities add them to the app Pod spec and run the app again. You rinse and repeat this process until the app runs properly with the minimum set of capabilities.

As good as this is there are a few things to consider. Firstly you must perform extensive testing of your app. The last thing you want is a production edge case that you had not accounted for in your test environment. Such occurrences can crash your app in production. Secondly every fix and update to your app requires the exact same extensive testing against the capability set.

With these considerations in mind, it is vital that you have testing procedures and production release process that can handle all of this. By default Kubernetes implements the default set of capabilities implemented by your chosen container runtime. However you can override this in a pod security policy, or as part of the container `securityContext`

```
apiVersion: v1
kind: Pod
metadata:
    name: capability-test
spec:
    containers:
    - name: demo
      image: example.io/simple:1.0
      securityContext:
        capabilities:
```

```
        add: ["NET_ADMIN", "CHOWN"]
```

Filter syscalls Seccomp short for secure computing is similar in concept to capabilities but works by filtering syscalls rather than capabilities. The way a Linux process asks the kernel to perform an operation is by issuing a syscall . Seccomp lets you control which syscalls a particular container can make to the host kernel. As with capabilities a least privilege model is preferred where the only syscalls a container is allowed to make are those the ones it needs to in order to function and run properly. Seccomp went GA in Kubernetes in 1.19 and can be used in different ways based on the following seccomp profiles.

1. Non-blocking: Allow a Pod to run and records every syscall it makes to an audit log, you can use to create a custom profile. The idea is to run your app, Pod in a `dev/test` environment and make it do everything designed to do. When you are done you will have a log file listing every syscall the Pod needs in order to run. You then use this to create a custom profile that only allows the syscalls the app needs least privilege
2. Blocking: Blocks all syscalls, it is extremely secure, but prevents Pod from doing anything useful
3. Runtime: forced a Pod to use the seccomp profile, defined by its container runtime. This is a common place to start if you have not created a custom profile yet. Profiles that ship with container runtimes (like Docker and containerd) are not the most secure in the world, but they are not wide open either. They are usually designed to be balance of usable and secure and they are thoroughly tested.
4. Custom: A profile that only allows the syscalls your app needs in order to run. Everything else is blocked. It is a common to extensively test your app in dev/test with a non blocking profile that records all syscalls to a log. You then use this log to identify the syscalls your app makes and build the customized profile. The danger with this approach is that your app has some edge case that you miss with your testing. If you this happens your app can fail in production when i when it hits an edge case and uses a syscall not captured in the logs during testing

Privilege escalation The only way to create a new process in Linux is for one process to clone itself and then load a new instructions to the new process (very simplified, but the original process is called the parent process and the copy is the child, this is called process forking).

By default Linux allows a child process to claim more privileges than its parent. This is usually a bad idea. In fact you will often want a child process to have the same or less privileges than its parent. This is especially true for containers as their security configurations are defined against their initial configuration and not against potentially escalated privileges.

Fortunately it is possible to prevent privilege escalation through a `PodSecurityPolicy` or the `securityContext` property of an individual container.

Pod security policies As you have seen throughout the sections one can enable security settings on a per Pod basis by setting security context attributes in individual Pod manifest files. However this approach does not scale well, it requires developers and operators to remember to do this for every Pod manifest spec, and is prone to errors. Pod security Policies offer a better way.

Pod security policies allow you to define a security settings at the cluster level. You can then apply them to targeted set of Pods as part of the deployment process. This approach scales better requires less effort from developers and admins and is less prone to error, it also lends itself to situations where you have a team dedicated to securing apps in production.

Pod Security Policies are implemented as an admission controller and in order to use them, a Pod `ServiceAccount` must be authorized to use it. Once this is done policies are applied to new requests to create Pods as they pass through the API admission chain.

Real world security

In the previous sections it was described how to thread model the Kubernetes security using the `STRIDE` model. In this section the common security related challenges will be covered that are likely to be encountered in the real world, when implementing a Kubernetes cluster.

While every Kubernetes deployment is different there are many similarities as a result the examples you will see will apply to most Kubernetes deployments large and small. Now then we are not offering cookbook style solutions, instead we will be looking at things from the kind of high level view a security architect has. The section is split into the following sub-sections

- CI/CD pipeline
- Infrastructure and networking
- Identity and access management
- Security monitoring and auditing

CI/CD pipeline

Containers are a revolutionary app packaging and runtime technology. On the packaging front they bundle apps code and dependencies into an image, as well as code and dependencies images contain the commands required to run the app. This has enabled containers to hugely simplify the process of building sharing and running apps, it also overcome the infamous - it worked on my laptop.

However containers make running dangerous code easier than ever before, With this in mind let us look at some ways you can secure the flow of app code from a developer laptop to production servers.

Image Repositories