Kubernetes Lighthouse

Rosario Scalia

January 28, 2024

Contents

Chapter 1

Introduction

Kubernetes (K8) is a popular open-source *container orchestrator* originally developed by Google and now maintained by the Cloud Native Computing Foundation (CNCF).

A container orchestrator is a piece of software that manages containers life cycle *at scale and in automated manner*.

Examples of services provisioned by container orchestrators are listed below:

- Traffic Load Balancing
- On-demand scalability
- Fault tollerance
- Auto-discovery to automatically discover and communicate among microservices' containers
- Containers Configuration
- API Gateway
- Protection for cluster resources and microservices APIs
- Automated applications update/rollback

Chapter 2

Kubernetes Architecture

A Kubernetes cluster is composed of several nodes (compute machines). Nodes can be categorized in the following 2 categories:

- Control Plane
- Worker

Said that, each node runs some *agents*: a piece of software that manage a particular *cluster or container feature*.

In the following, it is showed a graph that depict the overall Kubernetes architecture:

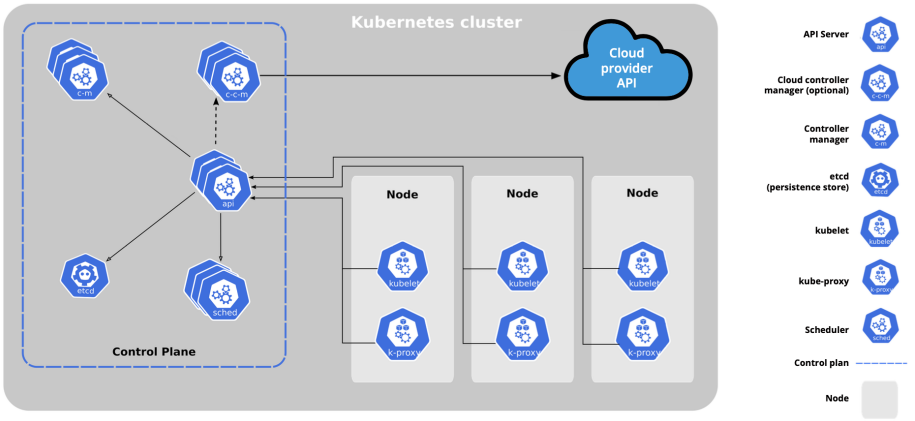

Figure 2.1: Kubernetes Arch

2.1 Control Plane Nodes

The control plane node provides a running environment for the control plane agents responsible for managing the state of a Kubernetes cluster, and it is the brain behind all operations inside the cluster. The control plane components are agents with very distinct roles in the cluster's management.

It is important to keep the control plane running at all costs. Losing the control plane may introduce downtime, causing service disruption to clients,

with possible loss of business. To ensure the control plane's fault tolerance, control plane node replicas can be added to the cluster, configured in High-Availability (HA) mode. While only one of the control plane nodes is dedicated to actively managing the cluster, the control plane components stay in sync across the control plane node replicas. This type of configuration adds resiliency to the cluster's control plane, should the active control plane node fail.

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

In the stacked key-value store topology, HA control plane node replicas ensure the key-value store's resiliency as well. However, that is not the case with external key-value store topology, where the dedicated key-value store hosts have to be separately replicated for HA, a configuration that introduces the need for additional hardware, hence additional operational costs.

Said that, a *Control Plane* node runs the following agents:

- API Server
- Scheduler
- Controller Managers
- KV Data Store
- Node Agent
- Proxy

together with a *Container Runtime*.

2.1.1 API Server

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

Overall API schema of the server is shown below:

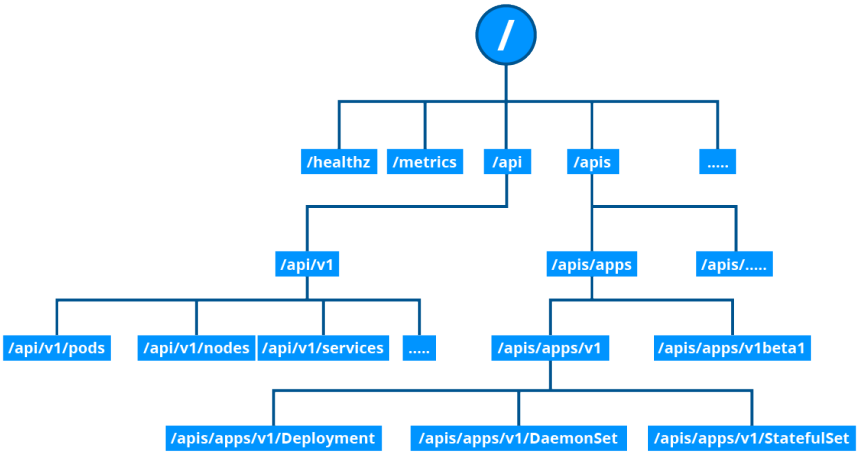

Figure 2.2: API Server endpoints schema

That endpoints allow operators and users to directly interact with the cluster. Using both CLI tools and the Dashboard UI, we can access the API server running on the control plane node to perform various operations to modify the cluster's state. The API Server is accessible through its endpoints by agents and users possessing the required credentials.

HTTP API directory tree of Kubernetes can be divided into three independent group types:

- **Core group (/api/v1):** This group includes objects such as Pods, Services, Nodes, Namespaces, ConfigMaps, Secrets, etc.
- **Named group:** This group includes objects in `/apis/$NAME/$VERSION` format. These different API versions imply different levels of stability and support.
- **System-wide:** This group consists of system-wide API endpoints, like `/healthz, /logs, /metrics, /ui`.

2.1.2 Scheduler

The role of the *kube-scheduler* is to assign new workload objects, such as pods encapsulating containers, to nodes - typically worker nodes. During the scheduling process, decisions are made based on current Kubernetes cluster state and new workload object's requirements. The scheduler obtains from the key-value store, via the API Server, resource usage data for each worker node in the cluster. The scheduler also receives from the API Server the new workload object's requirements which are part of its configuration data. Requirements may include constraints that users and operators set, such as scheduling work on a node labeled with disk==ssd key-value pair. The scheduler also takes into account Quality of Service (QoS) requirements, data locality, affinity, anti-affinity, taints, toleration, cluster topology, etc. Once all the cluster data is available, the scheduling algorithm filters the nodes with predicates to isolate the possible node candidates which then are scored with priorities in order

to select the one node that satisfies all the requirements for hosting the new workload. The outcome of the decision process is communicated back to the API Server, which then delegates the workload deployment with other control plane agents.

2.1.3 Controller Managers

The controller managers are components of the control plane node running controllers or operator processes to regulate the state of the Kubernetes cluster. Controllers are watch-loop processes continuously running and comparing the cluster's desired state (provided by objects' configuration data) with its current state (obtained from the key-value store via the API Server). In case of a mismatch, corrective action is taken in the cluster until its current state matches the desired state.

2.1.4 KV Data Store

etcd is an open source project under the Cloud Native Computing Foundation (CNCF). etcd is a strongly consistent, distributed key-value data store used to persist a Kubernetes cluster's state. New data is written to the data store only by appending to it, data is never replaced in the data store. Obsolete data is compacted (or shredded) periodically to minimize the size of the data store. Out of all the control plane components, only the API Server is able to communicate with the etcd data store.

KV store topology can be *Stacked* or *External*. A Stacked topology implies that KV store is directly coupled with each *Control Plane Node*; indeed, an External topology implies a separated *etcd* cluster that provides the KV service.

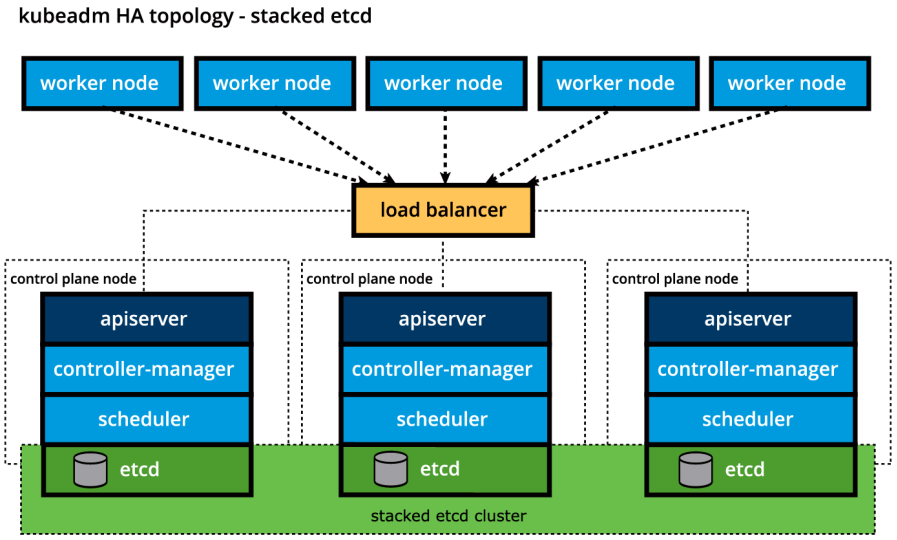

Figure 2.3: Stacked KV Data Store Topology

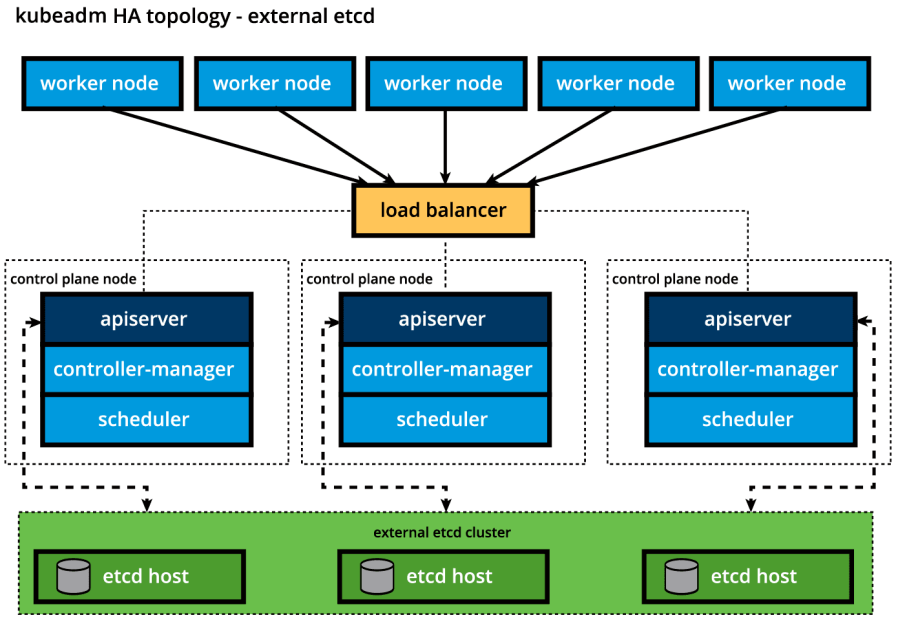

Figure 2.4: External KV Data Store Topology

2.2 Worker Nodes

A worker node provides a running environment for client applications. These applications are microservices running as application containers. In Kubernetes the application containers are encapsulated in Pods, controlled by the cluster control plane agents running on the control plane node. Pods are scheduled on worker nodes, where they find required compute, memory and storage resources to run, and networking to talk to each other and the outside world. A Pod is the smallest scheduling work unit in Kubernetes. It is a logical collection of one or more containers scheduled together, and the collection can be started, stopped, or rescheduled as a single unit of work.

Worker nodes runs the following agents:

- Node Agent
- Proxy

along with a *Container Runtime*.

2.2.1 Container Runtime

Although Kubernetes is described as a "container orchestration engine", it lacks the capability to directly handle and run containers. In order to manage a container's lifecycle, Kubernetes requires a container runtime on the node where a Pod and its containers are to be scheduled. Runtimes are required on all nodes of a Kubernetes cluster, both control plane and worker.

2.2.2 Node Agent

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

The kubelet connects to container runtimes through a plugin based interface - the Container Runtime Interface (CRI). The CRI consists of protocol buffers, gRPC API, libraries, and additional specifications and tools. In order to connect to interchangeable container runtimes, kubelet uses a CRI shim, an application which provides a clear abstraction layer between kubelet and the container runtime.

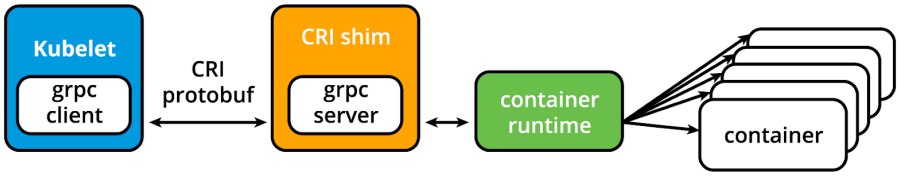

Figure 2.5: Kubelet-Container Runtime Workflow

2.2.3 Proxy

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

The kube-proxy is responsible for TCP, UDP, and SCTP stream forwarding or random forwarding across a set of Pod backends of an application, and it implements forwarding rules defined by users through Service API objects (explained later).

2.2.4 Add-Ons

Add-ons are cluster features and functionality not yet available in Kubernetes, therefore implemented through 3rd-party pods and services like the following:

- **DNS:** Cluster DNS is a DNS server required to assign DNS records to Kubernetes objects and resources
- **Monitoring:** Collects cluster-level container metrics and saves them to a central data store.
- **Logging:** Collects cluster-level container logs and saves them to a central log store for analysis.

2.3 Networking Concerns

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

- **Container-2-Container** communication inside Pods
- **Pod-to-Pod** communication on the same node and across cluster nodes
- **Service-to-Pod** communication within the same namespace and across cluster namespaces
- **External-to-Service** communication for clients to access applications in a cluster

2.3.1 Container-2-Container

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

When a grouping of containers defined by a Pod is started, a special infrastructure Pause container is initialized by the Container Runtime for the sole purpose of creating a network namespace for the Pod. All additional containers, created through user requests, running inside the Pod will share the Pause container's network namespace so that they can all talk to each other via localhost.

2.3.2 Pod-2-Pod

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

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

2.3.3 External-2-Pod

A successfully deployed containerized application running in Pods inside a Kubernetes cluster may require accessibility from the outside world. Kubernetes enables external accessibility through Services, complex encapsulations of network routing rule definitions stored in iptables on cluster nodes and implemented by kube-proxy agents.

By exposing services to the external world with the aid of kube-proxy, applications become accessible from outside the cluster over a virtual IP address and a dedicated port number.

Chapter 3

Kubernetes Building Blocks

Kubernetes became popular due to its advanced application lifecycle management capabilities, implemented through a *rich object model*, representing *different persistent entities* in the Kubernetes cluster. Those entities describe:

- What containerized applications we are running
- The nodes where the containerized applications are deployed
- Application resource consumption
- Policies attached to applications, like restart/upgrade policies, fault tolerance, ingress/egress, access control

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

An object definition manifest must include other fields that specify the version of the API we are referencing as the *apiVersion*, the object type as *kind*, and additional data helpful to the cluster or users for accounting purposes - the *metadata*. Examples of Kubernetes object types are Nodes, Namespaces, Pods, ReplicaSets, Deployments, DaemonSets, etc.

Anyway, when creating an object, the object's configuration data section from below the *spec* field has to be submitted to the Kubernetes API Server.

3.1 Nodes

Nodes are virtual identities assigned by Kubernetes to the systems part of the cluster - whether Virtual Machines, bare-metal, Containers, etc. These identities are unique to each system, and are used by the cluster for resources accounting and monitoring purposes, which helps with workload management throughout the cluster.

Each node is managed with the help of two Kubernetes node agents - **kubelet** and **kube-proxy**, while it also hosts a **container runtime**. The container runtime is required to run all containerized workload on the node - control plane

agents and user workloads. The kubelet and kube-proxy node agents are responsible for executing all local workload management related tasks - interact with the runtime to run containers, monitor containers and node health, report any issues and node state to the API Server, and managing network traffic to containers.

As said before, there are two distinct types of nodes: control plane and worker. A typical Kubernetes cluster includes at least one control plane node, but it may include multiple control plane nodes for Highly Available (HA) control plane

The control plane nodes run the control plane agents, such as the API Server, Scheduler, Controller Managers, and etcd in addition to the kubelet and kube-proxy node agents, the container runtime, and add-ons for container networking, monitoring, logging, DNS, etc.

Worker nodes run the kubelet and kube-proxy node agents, the container runtime, and add-ons for container networking, monitoring, logging, DNS, etc.

3.2 Namespaces

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

Generally, Kubernetes creates four Namespaces out of the box: **kube-system, kube-public, kube-node-lease,** and **default**. The kube-system Namespace contains the objects created by the Kubernetes system, mostly the control plane agents. The default Namespace contains the objects and resources created by administrators and developers, and objects are assigned to it by default unless another Namespace name is provided by the user. kube-public is a special Namespace, which is unsecured and readable by anyone, used for special purposes such as exposing public (non-sensitive) information about the cluster. The newest Namespace is kube-node-lease which holds node lease objects used for node heartbeat data. Good practice, however, is to create additional Namespaces, as desired, to virtualize the cluster and isolate users, developer teams, applications, or tiers.

Resource quotas help users limit the overall resources consumed within Namespaces, while *LimitRanges* help limit the resources consumed by Containers and their enclosing objects in a Namespace.

3.3 Pods

A Pod is the smallest Kubernetes workload object. It is the unit of deployment in Kubernetes, which represents a single instance of the application. A Pod is a logical collection of one or more containers, enclosing and isolating them to ensure that they:

- Are scheduled together on the same host with the Pod
- Share the same network namespace, meaning that they share a single IP address originally assigned to the Pod

- Have access to mount the same external storage (volumes) and other common dependencies

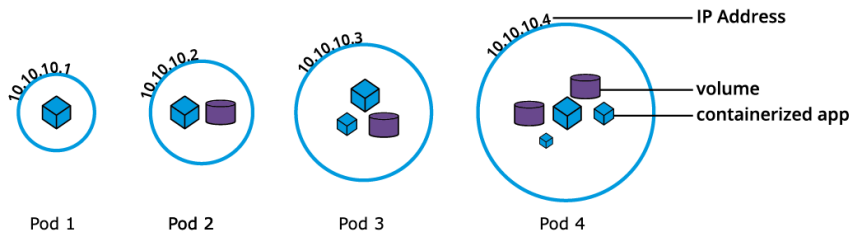

Figure 3.1: Pod schema sample

Pods are ephemeral in nature, and they do not have the capability to *self-heal themselves*. That is the reason they are used with controllers, or operators (controllers/operators are used interchangeably), which handle Pods' replication, fault tolerance, self-healing, etc.

Examples of controllers are *Deployments, ReplicaSets, DaemonSets, Jobs, etc.* When an operator is used to manage an application, the Pod's specification is nested in the controller's definition using the Pod Template.

Example of Pod definition ,in YAML file, is listed below:

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx-pod
5   labels:
6     run: nginx-pod
7 spec:
8   containers:
9   - name: nginx
10    image: nginx:1.22.1
11    ports:
12    - containerPort: 80
```

The *apiVersion* field must specify **v1** for the Pod object definition. The second required field is *kind* specifying the Pod object type. The third required field *metadata*, holds the object's name and optional labels and annotations. The fourth required field *spec* marks the beginning of the block defining the desired state of the Pod object - also named the PodSpec.

The contents of spec are evaluated for scheduling purposes, then the kubelet of the selected node becomes responsible for running the container image with the help of the container runtime of the node. The Pod's name and labels are used for workload accounting purposes.

Our Pod creates a single container running the `nginx:1.22.1` image pulled from a container image registry, in this case from Docker Hub. The `containerPort` field specifies the container port to be exposed by Kubernetes resources for inter-application access or external client access.

3.4 Labels

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

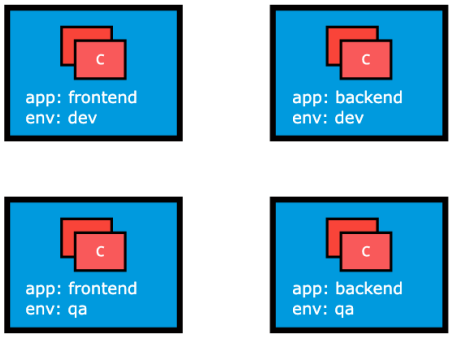

Figure 3.2: Labels example

3.5 Labels Selectors

Controllers, or operators, and Services, use label selectors to select a subset of objects. Kubernetes supports two types of Selectors:

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

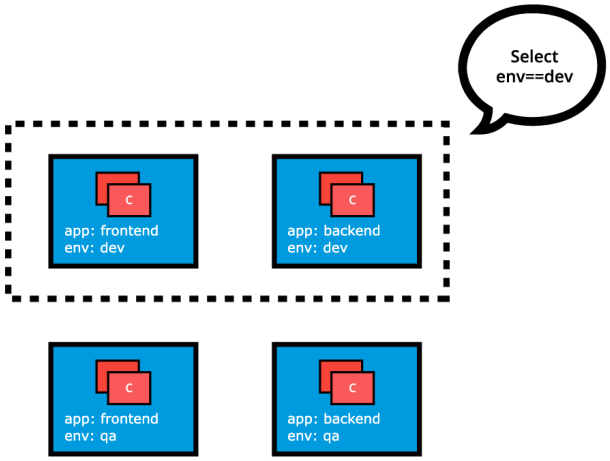

Figure 3.3: Labels Selectors

3.6 Replication Controllers

Kubernetes is shipped with a set of *replication controllers* that ensures a specified number of replicas of a Pod is running at any given time, by constantly comparing the actual state with the desired state of the managed application.

If there are more Pods than the desired count, the replication controller randomly terminates the number of Pods exceeding the desired count, and, if there are fewer Pods than the desired count, then the replication controller requests additional Pods to be created until the actual count matches the desired count.

Generally, we do not deploy a Pod independently, as it would not be able to re-start itself if terminated in error because a Pod misses the much desired self-healing feature that Kubernetes otherwise promises. The recommended method is to use one of the replication controllers to run and manage Pods.

Another important feature of replication controllers is *application updates*.

Available replication controllers are:

- Deployments
- ReplicaSets
- DaemonSets

3.6.1 ReplicaSet

A *ReplicaSet* implements the replication and self-healing aspects said before. ReplicaSets support both equality- and set-based Selectors.

When a single instance of an application is running there is always the risk of the application instance crashing unexpectedly, or the entire server hosting the application crashing. If relying only on a single application instance, such a crash could adversely impact other applications, services, or clients. To avoid

such possible failures, we can run in parallel multiple instances of the application, hence achieving high availability. The lifecycle of the application defined by a Pod will be overseen by a controller - the ReplicaSet. With the help of the ReplicaSet, we can scale the number of Pods running a specific application container image. Scaling can be accomplished manually or through the use of an autoscaler.

Below we graphically represent a ReplicaSet, with the replica count set to 3 for a specific Pod template. Pod-1, Pod-2, and Pod-3 are identical, running the same application container image, being cloned from the same Pod template. For now, the current state matches the desired state. Keep in mind, however, that although the three Pod replicas are said to be identical - running an instance of the same application, same configuration, they are still distinct in identity - Pod name, IP address, and the Pod object ensures that the application can be individually placed on any worker node of the cluster as a result of the scheduling process.

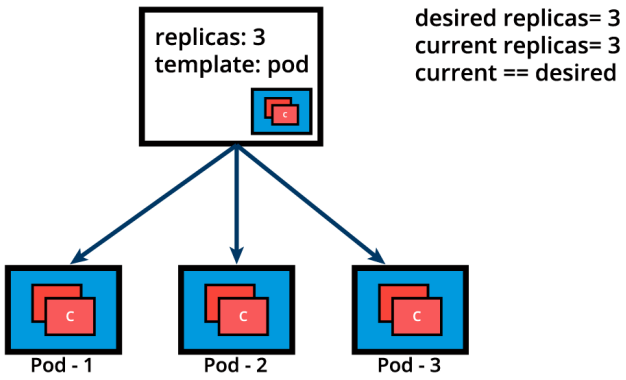

Figure 3.4: ReplicaSets example

Below is reported an example of ReplicaSets object definition:

```
1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    name: frontend
5    labels:
6      app: guestbook
7      tier: frontend
8  spec:
9    replicas: 3
10   selector:
11     matchLabels:
12       app: guestbook
13   template:
14     metadata:
15       labels:
16         app: guestbook
17     spec:
18       containers:
19       - name: php-redis
20         image: gcr.io/google_samples/gb-frontend:v3
```

As you can see, pod definition *is nested* in ReplicaSets one throught the *template* tag.

ReplicaSets can be used independently as Pod controllers but they only offer a limited set of features. A set of complementary features are provided by Deployments, the recommended controllers for the orchestration of Pods.

Deployments manage the creation, deletion, and updates of Pods. A Deployment automatically creates a ReplicaSet, which then creates a Pod. There is no need to manage ReplicaSets and Pods separately, the Deployment will manage them on our behalf.

3.6.2 Deployment

Deployment objects provide declarative updates to Pods and ReplicaSets. The *DeploymentController* is part of the control plane node's controller manager, and as a controller it also ensures that the current state always matches the desired state of our running containerized application.

It allows for *seamless application updates and rollbacks*, known as the default `RollingUpdate` strategy, through rollouts and rollbacks, and it directly manages its ReplicaSets for application scaling. It also supports a disruptive, less popular update strategy, known as `Recreate`.

Below, is showed a Deployment object definition:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5    labels:
6      app: nginx-deployment
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: nginx-deployment
12   template:
13     metadata:
14       labels:
15         app: nginx-deployment
16     spec:
17       containers:
18       - name: nginx
19         image: nginx:1.20.2
20         ports:
21         - containerPort: 80
```

In the following example, a new Deployment creates ReplicaSet A which then creates 3 Pods, with each Pod Template configured to run one `nginx:1.20.2` container image. In this case, the ReplicaSet A is associated with `nginx:1.20.2` representing a state of the Deployment. This particular state is recorded as Revision 1.

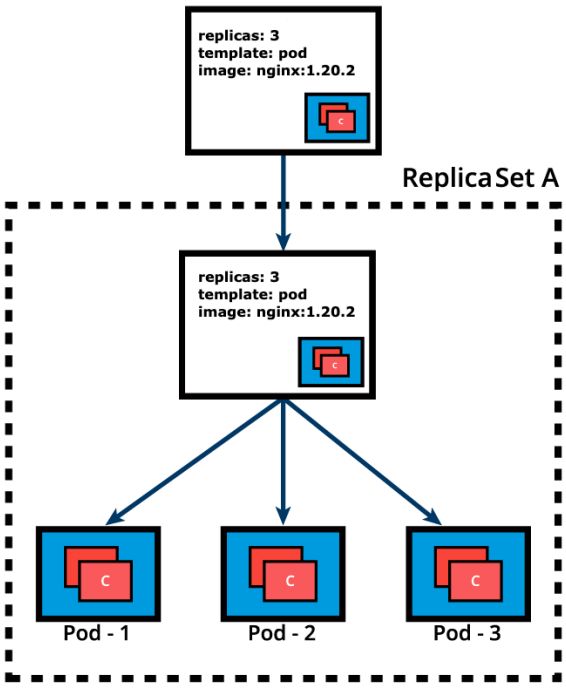

Figure 3.5: Deploy example

In time, we need to push updates to the application managed by the Deployment object. Let's change the Pods' Template and update the container image from `nginx:1.20.2` to `nginx:1.21.5`. The Deployment triggers a new ReplicaSet B for the new container image versioned 1.21.5 and this association represents a new recorded state of the Deployment, Revision 2.

3.6.2.1 Rolling Update

The *seamless transition* among 2 *Revisions* is called *Rolling Update*.

A rolling update is triggered when we update specific properties of the Pod Template for a deployment. While planned changes such as updating the container image, container port, volumes, and mounts would trigger a new Revision, other operations that are dynamic in nature, like scaling or labeling the deployment, do not trigger a rolling update, thus do not change the Revision number.

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

Once ReplicaSet New is ready, the Deployment starts actively managing it. However, the Deployment keeps its prior configuration states saved as Revisions which play a key factor in the rollback capability of the Deployment - returning to a prior known configuration state.

In the above example, if the performance of the new nginx:1.21.5 is not satisfactory, the Deployment can be rolled back to a prior Revision, in this case from Revision 2 back to Revision 1 running nginx:1.20.2 once again.

3.6.3 DaemonSets

DaemonSets are operators designed to manage node agents. They resemble ReplicaSet and Deployment operators when managing multiple Pod replicas and application updates, but the DaemonSets present a distinct feature that enforces a single Pod replica to be placed per Node, on all the Nodes. In contrast, the ReplicaSet and Deployment operators by default have no control over the scheduling and placement of multiple Pod replicas on the same Node.

DaemonSet operators are commonly used in cases when we need to collect monitoring data from all Nodes, or to run a storage, networking, or proxy daemons on all Nodes, to ensure that we have a specific type of Pod running on all Nodes at all times.

Whenever a Node is added to the cluster, a Pod from a given DaemonSet is automatically placed on it. Although it ensures an automated process, the DaemonSet's Pods are placed on all cluster's Nodes by the controller itself, and not with the help of the default Scheduler. When any one Node crashes or it is removed from the cluster, the respective DaemonSet operated Pods are garbage collected. If a DaemonSet is deleted, all Pod replicas it created are deleted as well.

Below, is showed a DaemonSets object definition:

```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: fluentd-agent
5    namespace: kube-system
6    labels:
7      k8s-app: fluentd-agent
8  spec:
9    selector:
10     matchLabels:
11       k8s-app: fluentd-agent
12   template:
13     metadata:
14       labels:
15         k8s-app: fluentd-agent
16     spec:
17       containers:
18       - name: fluentd-agent
19         image: quay.io/fluentd_elasticsearch/fluentd:v2.5.2
```

3.7 Services

A containerized application deployed to a Kubernetes cluster (a Pod) may need to reach other such applications (other Pods), or it may need to be accessible to other applications and possibly clients.

This is problematic because the container does not expose its ports to the cluster's network, and it is not discoverable either. The solution would be a simple port mapping, as offered by a typical container host.

However, due to the complexity of the Kubernetes framework, such a simple port mapping is not that "simple". The solution is much more sophisticated, with the involvement of the kube-proxy node agent, IP tables, routing rules, cluster DNS server, all collectively implementing a micro-load balancing mechanism that exposes a container's port to the cluster's network, even to the outside world if desired.

This mechanism is called a Service, and it is the recommended method to expose any containerized application to the Kubernetes network. The benefits of the Kubernetes Service becomes more obvious when exposing a multi-replica application, when multiple containers running the same image need to expose the same port.

This is where the simple port mapping of a container host would no longer work, but the Service would have no issue implementing such a complex requirement.

Chapter 4

Installation & Supported Platforms

Kubernetes installation can be performed in the following ways:

- **All-in-One Single-Node Installation:** In this setup, all the control plane and worker components are installed and running on a single-node. While it is useful for learning, development, and testing, it is not recommended for production purposes.
- **Single-Control Plane and Multi-Worker Installation:** In this setup, we have a single-control plane node running a stacked etcd instance. Multiple worker nodes can be managed by the control plane node.
- **Single-Control Plane with Single-Node etcd, and Multi-Worker Installation:** In this setup, we have a single-control plane node with an external etcd instance. Multiple worker nodes can be managed by the control plane node.
- **Multi-Control Plane and Multi-Worker Installation:** In this setup, we have multiple control plane nodes configured for High-Availability (HA), with each control plane node running a stacked etcd instance. The etcd instances are also configured in an HA etcd cluster and multiple worker nodes can be managed by the HA control plane.
- **Multi-Control Plane with Multi-Node etcd, and Multi-Worker Installation:** In this setup, we have multiple control plane nodes configured in HA mode, with each control plane node paired with an external etcd instance. The external etcd instances are also configured in an HA etcd cluster, and multiple worker nodes can be managed by the HA control plane. This is the most advanced cluster configuration recommended for production environments.

Furthermore, Kubernetes can be installed on (1) Bare Metal (2) Public Cloud (3) Private Cloud (4) On Public Cloud as *Hosted Solution*.

From distribution point of view, it exists several *Kubernetes distributions*: a piece of software that automatically setups a Kubernetes cluster avoiding user the *manual hard installation and system bootstrapping*.

Chapter 5

Minikube K8 Distribution

For the installation guide, we choose the *Minikube* Kubernetes distribution on *Canonical Ubuntu* operative system.

Minikube is one of the easiest, most flexible and popular methods to run an all-in-one or a multi-node local Kubernetes cluster, isolated by Virtual Machines (VM) or Containers, run directly on our workstations. Minikube is the tool responsible for the installation of Kubernetes components, cluster bootstrapping, and cluster tear-down when no longer needed. It includes additional features aimed to ease the user interaction with the Kubernetes cluster, but nonetheless, it initializes for us a fully functional, non-production, Kubernetes cluster extremely convenient for learning purposes. Minikube can be installed on native macOS, Windows, and many Linux distributions.

5.1 Minikube Isolation

Minikube is one of the easiest, most flexible and popular methods to run an all-in-one or a multi-node local Kubernetes cluster directly on our local workstations. It installs and runs on any native OS such as Linux, macOS, or Windows. However, in order to fully take advantage of all the features Minikube has to offer, a Type-2 Hypervisor or a Container Runtime should be installed on the local workstation, to run in conjunction with Minikube. The role of the hypervisor or container runtime is to offer an isolated infrastructure for the Minikube Kubernetes cluster components, that is easily reproducible, easy to use and tear down. This isolation of the cluster components from our daily environment ensures that once no longer needed, the Minikube components can be safely removed leaving behind no configuration changes to our workstation, thus no traces of their existence. This does not mean, however, that we are responsible for the provisioning of any VMs or containers with guest operating systems with the help of the hypervisor or container runtime. Minikube includes the necessary adapters to interact directly with the isolation software of choice to build all its infrastructure as long as the Type-2 Hypervisor or Container Runtime is installed on our workstation.

The isolation software can be specified by the user with the `--driver` option, otherwise Minikube will try to find a preferred method for the host OS of the workstation.

5.2 Minikube Startup

Once decided on the isolation method, the next step is to determine the required number of Kubernetes cluster nodes, and their sizes in terms of CPU, memory, and disk space.

Keep in mind that Minikube now supports all-in-one single-node and multi-node clusters. Regardless of the isolation method and the expected cluster and node sizes, a local Minikube Kubernetes cluster will ultimately be impacted and/or limited by the physical resources of the host workstation. We have to be mindful of the needs of the host OS and any utilities it may be running, then the needs of the hypervisor or the container runtime, and finally the remaining resources that can be allocated to our Kubernetes cluster.

Anyway, Kubernetes nodes are expected to access the internet as well, for software updates, container image downloads, and for client accessibility.

Following the node(s)' provisioning phase, Minikube invokes *kubeadm*, to bootstrap the Kubernetes cluster components inside the previously provisioned node(s). We need to ensure that we have the necessary hardware and software required by Minikube to build our environment.

5.3 Minikube Requirements

- **Docker**

5.4 Installation Guide (Ubuntu)

Installation steps for Minikube:

```
1 curl -LO https://storage.googleapis.com/minikube/releases/latest/
      minikube-linux-amd64
2 sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Installation steps for Kubectl:

```
1 curl -LO "https://dl.k8s.io/release/$(curl -L -s https
      ://dl.k8s.io/release/stable.txt)/bin/linux/amd64/
      kubectl"
2 sudo install -o root -g root -m 0755 kubectl /usr/
      local/bin/kubectl
```

Installation steps for enabling GPU support:

```
1 sudo sysctl net.core.bpf_jit_harden
2 INSTALL NVIDIA-CONTAINER-TOOLKIT
3 sudo nvidia-ctk runtime configure --runtime=docker
4 sudo systemctl restart docker
```

Osservazione 1 (enabling GPU support). If previous `sysctl net.core...` command returns value different than 0, execute the following commands:

```
echo "net.core.bpf_jit_harden=0" | sudo tee -a /etc/sysctl.conf
sudo sysctl -p
```

Installation steps for bash minikube auto completion:

```
sudo apt install bash-completion
source /etc/bash_completion
source <(minikube completion bash)
minikube completion bash
```

Installation steps for bash Kubectl auto completion:

```
sudo apt install -y bash-completion
source /usr/share/bash-completion/bash_completion
source <(kubectl completion bash)
echo 'source <(kubectl completion bash)' >>~/.bashrc
```

For a sanity check about installation, try issuing the following commands:

Listing 5.1: Minikube Install Sanity Check

```
minikube start
minikube status
```

If installation is accomplished you will see an output like the following:

Listing 5.2: Minikube Ready Installation

```
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
```

5.5 Minikube Profiles

Now that we have familiarized ourselves with the default minikube start command, let's dive deeper into Minikube to understand some of its more advanced features.

The minikube start by default selects a driver isolation software, such as a hypervisor or a container runtime, if one (VitualBox) or multiple are installed on the host workstation. In addition it downloads the latest Kubernetes version components. With the selected driver software it provisions a single VM named minikube (with hardware profile of CPUs=2, Memory=6GB, Disk=20GB) or container to host the default single-node all-in-one Kubernetes cluster. Once the node is provisioned, it bootstraps the Kubernetes control plane (with the default kubeadm tool), and it installs the latest version of the default container runtime, Docker, that will serve as a running environment for the containerized applications we will deploy to the Kubernetes cluster. The minikube start command generates a default minikube cluster with the specifications described above and it will store these specs so that we can restart the default cluster whenever desired. The object that stores the specifications of our cluster is called a profile.

As Minikube matures, so do its features and capabilities. With the introduction of profiles, Minikube allows users to create custom reusable clusters that can all be managed from a single command line client.

5.6 Kubectl Config

To access the Kubernetes cluster, the kubectl client needs the control plane node endpoint and appropriate credentials to be able to securely interact with the API Server running on the control plane node. While starting Minikube, the startup process creates, by default, a configuration file, config, inside the .kube directory (often referred to as the kubeconfig), which resides in the user's home directory. The configuration file has all the connection details required by kubectl. By default, the kubectl binary parses this file to find the control plane node's connection endpoint, along with the required credentials. Multiple kubeconfig files can be configured with a single kubectl client.

5.7 Accessing Application by Name

In order to provide an easy access to our applications, we will need to update the host configuration file (`/etc/hosts` on Mac and Linux) on our workstation with a mapping among cluster *Control Plane* node IP and *related application hosts defined in the relative Ingress object*:

```
1 $CONTROL_PLANE_NODE_IP $APP_HOST_NAME_1 $APP_HOST_NAME_2
```

In this way, we are simulating a *full flagged DNS service* and in turn user could access application using something like `http://example.com/start` supposing exists an host of name *example.com* that points to a particular *Cluster Service* that expose an *HTTP* endpoint.

5.8 Docker and Images Registries

Minikube is shipped by his docker artifact, for this reason each `image pull` from a different docker instance (like the one installed on the host machine) will not have effect for minikube. In order to pull missing images, you have to specify images on the *Pod definitions* and it will be retrieved from the *Docker Hub* registry as default behaviour.

Another option is login to *each target node* via SSH , using `minikube ssh` command, and pull/make proper images from there.

5.9 Volume Sharing in Multi-Node setups (Ubuntu)

In order to share a common storage among pod's containers, user have to setup an *NFS* service in the host OS.

In order to do it, perform the following steps:

1. Install the service

```
1         sudo apt install nfs-kernel-server -y
```

2. Create shared folder on `/mnt/` dir

```
1         sudo mkdir -p /mnt/SHARFLD && sudo chown -R nobody:
              nogroup /mnt/SHARFLD/
```

3. Define shared folder export adding the following line to the `/etc/exports` file

```
1          /mnt/SHARFLD *(rw,sync,no_subtree_check,no_root_squash,
               insecure)
```

4. Export shared folder on NFS

```
1          sudo exportfs -a && sudo systemctl restart nfs-kernel-
               server
```

5. Sanity check

```
1          $ sudo exportfs -v
2          /mnt/SHARFLD <world>
3          (rw,wdelay,insecure,no_root_squash,no_subtree_check,sec
               =sys,rw,insecure,no_root_sq
```

6. From now, you can setup proper *PersistentVolumes* and respective *PersistentVolumeClaims* that reference the shared folder

Chapter 6

K8 Main Features

6.1 Authentication & Authorization

Every API request reaching the API server has to go through several control stages before being accepted by the server and acted upon. In this chapter, we will learn about the Authentication, Authorization, and Admission Control stages of the Kubernetes API access control.

To access and manage Kubernetes resources or objects in the cluster, we need to access a specific API endpoint on the API server. Each access request goes through the following access control stages:

- **Authentication:** Authenticate a user based on credentials provided as part of API requests.
- **Authorization:** Authorizes the API requests submitted by the authenticated user.
- **Admission Control:** Software modules that validate and/or modify user requests.

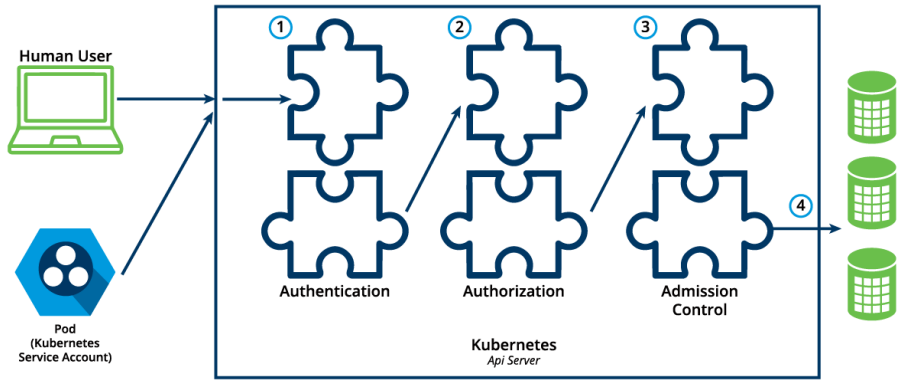

Figure 6.1: Authenticaton & Authorization K8 Schema

6.1.1 Authentication

Kubernetes does not have an object called user, nor does it store usernames or other related details in its object store. However, even without that, Kubernetes can use usernames for the Authentication phase of the API access control, and to request logging as well.

Kubernetes supports two kinds of users:

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

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

For authentication, Kubernetes uses a series of authentication modules:

- **X509 Client Certificates:** To enable client certificate authentication, we need to reference a file containing one or more certificate authorities by passing the `-client-ca-file=SOMEFILE` option to the API server. The certificate authorities mentioned in the file would validate the client certificates presented by users to the API server.
- **Static Token File:** We can pass a file containing pre-defined bearer tokens with the `-token-auth-file=SOMEFILE` option to the API server. Currently, these tokens would last indefinitely, and they cannot be changed without restarting the API server.
- **Bootstrap Tokens:** Tokens used for bootstrapping new Kubernetes clusters.
- **Service Account Tokens:** Automatically enabled authenticators that use signed bearer tokens to verify requests. These tokens get attached to Pods using the Service Account Admission Controller, which allows in-cluster processes to talk to the API server.
- **OpenID Connect Tokens:** OpenID Connect helps us connect with OAuth2 providers, such as Azure Active Directory, Salesforce, and Google, to offload the authentication to external services.
- **Webhook Token Authentication:** With Webhook-based authentication, verification of bearer tokens can be offloaded to a remote service.
- **Authenticating Proxy:** Allows for the programming of additional authentication logic.

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

6.1.2 Authorization

After a successful authentication, users can send the API requests to perform different operations. Here, these API requests get authorized by Kubernetes using various authorization modules that allow or deny the requests.

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

Common kind of authorizers are listed below:

- **Node:** Node authorization is a special-purpose authorization mode which specifically authorizes API requests made by kubelets. It authorizes the kubelet's read operations for services, endpoints, or nodes, and writes operations for nodes, pods, and events. For more details, please review the Node authorization documentation.
- **Attribute-Based Access Control (ABAC):** With the ABAC authorizer, Kubernetes grants access to API requests, which combine policies with attributes. To enable ABAC mode, we start the API server with the `--authorization-mode=ABAC` option, while specifying the authorization policy with `--authorization-policy-file=PolicyFile.json`.
- **Webhook:** In Webhook mode, Kubernetes can request authorization decisions to be made by third-party services, which would return true for successful authorization, and false for failure.
- **Role-Based Access Control (RBAC):** In general, with RBAC we regulate the access to resources based on the Roles of individual users. In Kubernetes, multiple Roles can be attached to subjects like users, service accounts, etc. While creating the Roles, we restrict resource access by specific operations, such as create, get, update, patch, etc. These operations are referred to as verbs.

6.1.2.1 RBAC

In RBAC, we can create two kinds of Roles:

- **Role:** A Role grants access to resources within a specific Namespace.
- **ClusterRole:** A ClusterRole grants the same permissions as Role does, but its scope is cluster-wide.

Example of role definition is listed below:

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: Role
3 metadata:
4   namespace: lfs158
5   name: pod-reader
6 rules:
7 - apiGroups: [""] # "" indicates the core API group
8   resources: ["pods"]
9   verbs: ["get", "watch", "list"]
```

The manifest above defines a *pod-reader* role, which has access only to read the Pods of *lfs158* Namespace. Once the role is created, we can bind it to users with a *RoleBinding* object. There are two kinds of RoleBindings:

- **RoleBinding:** It allows us to bind users to the same namespace as a Role. We could also refer to a ClusterRole in RoleBinding, which would grant permissions to Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.
- **ClusterRoleBinding:** It allows us to grant access to resources at a cluster-level and to all Namespaces.

A sample of RoleBinding definition is listed beloew:

```
1 apiVersion: rbac.authorization.k8s.io/v1
2 kind: RoleBinding
3 metadata:
4   name: pod-read-access
5   namespace: lfs158
6 subjects:
7 - kind: User
8   name: bob
9   apiGroup: rbac.authorization.k8s.io
10 roleRef:
11   kind: Role
12   name: pod-reader
13   apiGroup: rbac.authorization.k8s.io
```

To enable the RBAC mode, we start the API server with the `--authorization-mode=RBAC` option, allowing us to dynamically configure policies.

6.2 Service Discovery & Load Balancer

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

In this section, we will learn about *Service* objects used to abstract the communication between cluster internal microservices (Pod-2-Pod com), or with the external world (Client-2-Pod com).

A Service offers a single DNS entry for a *stateless containerized application* managed by the Kubernetes cluster, regardless of the number of replicas.

Furthermore, a Service provides also a *common load balancing access point* to the *stateless containerized application*.

In this scenario, an application is *set of pods logically grouped and managed by a controller* such as a Deployment, ReplicaSet, or DaemonSet.

An example of *Service* object definition is listed below:

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: frontend-svc
5  spec:
6    selector:
7      app: frontend
8    ports:
9    - protocol: TCP
10     port: 80
11     targetPort: 5000
```

In this example, we are creating a `frontend-svc` Service by selecting all the Pods that have the Label `key=app` set to `value=frontend`. By default, each Service receives an IP address routable only inside the cluster, known as `ClusterIP`. Anyway, proper service's pods are picked by mean of Label selector concept.

While the Service forwards traffic to Pods, we can select the targetPort on the Pod which receives the traffic. In our example, the frontend-svc Service receives requests from the user/client on port: 80 and then forwards these requests to one of the attached Pods on the targetPort: 5000.

It is very important to ensure that the value of the targetPort, which is 5000 in this example, matches the value of the containerPort property of the Pod spec section.

A logical set of a Pod's IP address, along with the targetPort is referred to as a Service endpoint. Endpoints are created and managed automatically by the Service, not by the Kubernetes cluster administrator.

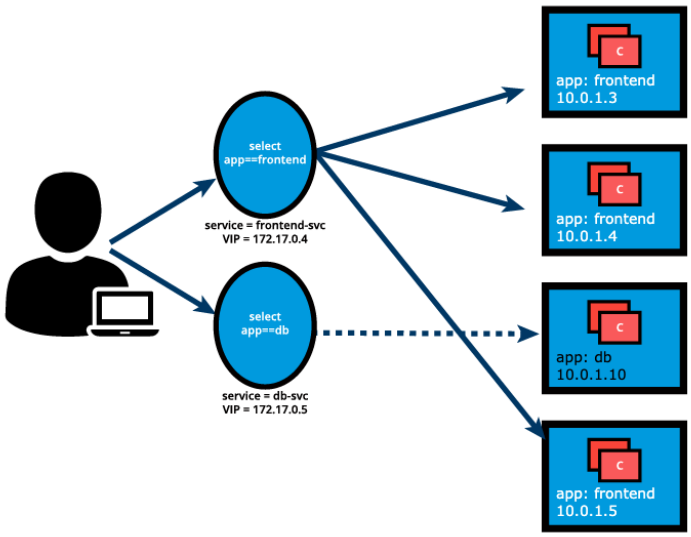

Figure 6.2: Service Example

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

6.2.1 Traffic Policies

The kube-proxy node agent together with the iptables implement the load-balancing mechanism of the Service when traffic is being routed to the application Endpoints. Due to restricting characteristics of the iptables this load-balancing is random by default. This means that the Endpoint Pod to receive the request forwarded by the Service will be randomly selected out of many replicas. This mechanism does not guarantee that the selected receiving Pod is the closest or even on the same node as the requester, therefore not the most efficient mechanism. Since this is the iptables supported load-balancing mechanism, if we desire better outcomes, we would need to take advantage of traffic policies.

Traffic policies allow users to instruct the kube-proxy on the context of the traffic routing. The two options are Cluster and Local:

- **Cluster:** allows kube-proxy to target all ready Endpoints of the Service in the load-balancing process
- **Local:** isolates the load-balancing process to only include the Endpoints of the Service located on the same node as the requester Pod. While this sounds like an ideal option, it does have a shortcoming - if the Service does not have a ready Endpoint on the node where the requester Pod is running, the Service will not route the request to Endpoints on other nodes to satisfy the request.

Both the Cluster and Local options are available for requests generated internally from within the cluster, or externally from applications and clients running outside the cluster.

6.2.2 Service Discovery

As Services are the primary mode of communication between containerized applications managed by Kubernetes, it is helpful to be able to discover them at runtime. Kubernetes supports two methods for discovering Services:

- **Enviroment Variables:** As soon as the Pod starts on any worker node, the kubelet daemon running on that node adds a set of environment variables in the Pod for all active Services. With this solution, we need to be careful while ordering our Services, as the Pods will not have the environment variables set for Services which are created after the Pods are created.

  For example, if we have an active Service called redis-master, which exposes port 6379, and its ClusterIP is 172.17.0.6, then, on a newly created Pod, we can see the following environment variables:

```
1 REDIS_MASTER_SERVICE_HOST=172.17.0.6
2 REDIS_MASTER_SERVICE_PORT=6379
3 REDIS_MASTER_PORT=tcp://172.17.0.6:6379
```

```
4           REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379
5           REDIS_MASTER_PORT_6379_TCP_PROTO=tcp
6           REDIS_MASTER_PORT_6379_TCP_PORT=6379
7           REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6
```

- **DNS:** Kubernetes has an add-on for DNS, which creates a DNS record for each Service and its format is `my-svc.my-namespace.svc.cluster.local`. Services within the same Namespace find other Services just by their names.

  If we add a Service `redis-master` in `my-ns` Namespace, all Pods in the same my-ns Namespace lookup the Service just by its name, redis-master.

  Pods from other Namespaces, such as `test-ns`, lookup the same Service by adding the respective Namespace as a suffix, such as `redis-master.my-ns` or providing the FQDN of the service as `redis-master.my-ns.svc.cluster.local`. This is the most common and highly recommended solution.

6.2.3 Service Type

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

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

Access scope is decided by *ServiceType* property, defined when creating the Service.

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

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

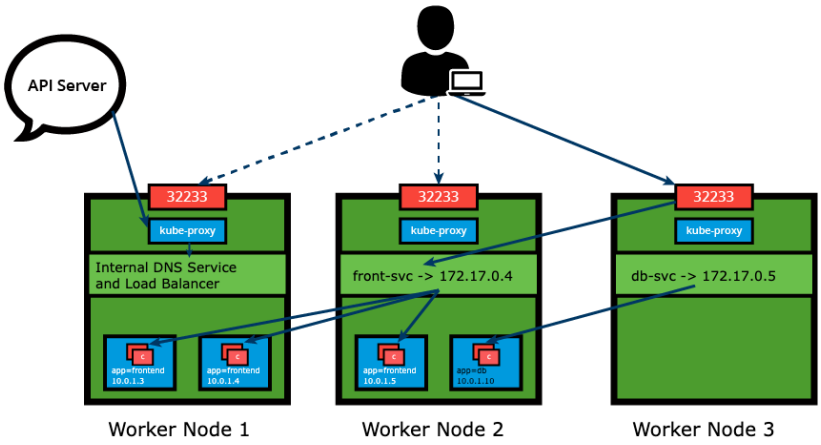

Figure 6.3: NodePort sample

The NodePort ServiceType is useful when we want to make our Services accessible from the external world. The end-user connects to any worker node on the specified high-port, which proxies the request internally to the ClusterIP of the Service, then the request is forwarded to the applications running inside the cluster.

Let's not forget that the Service is load balancing such requests, and only forwards the request to one of the Pods running the desired application.

An example of NodePort service is listed below:

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: my-service
5  spec:
6    selector:
7      app: myapp
8    type: NodePort
9    ports:
10   - name: http
11     protocol: TCP
12     port: 8080
13     targetPort: 80
14     nodePort: 31080
15   - name: https
16     protocol: TCP
17     port: 8443
18     targetPort: 443
19     nodePort: 31443
```

The my-service Service resource exposes Pods labeled app==myapp with possibly one container listening on ports 80 and 443, as described by the two targetPort fields. The Service will be visible inside the cluster on its ClusterIP and ports 8080 and 8443 as described by the two port fields, and it will also be accessible to incoming requests from outside the cluster on the two nodePort fields 31080 and 31443. When manifests describe multiple ports, they need to

be named as well, for clarity, as described by the two name fields with values http and https respectively. This Service is configured to capture traffic on ports 8080 and 8443 from within the cluster, or on ports 31080 and 31443 from outside the cluster, and forward that traffic to the ports 80 and 443 respectively of the Pods running the container.

6.2.4 Summary

For container-container communication scenarios, you can use the pair < `127.0.0.1 , PORT` > in order to establish the communication.

For Pod-2-Pod communication scenarios, you can use the pair < `ClusterIP/Serv_Name , SERVICE_PORT` > in order to establish the communication.

For External-2-Pod communication scenarios, you can use the pair < `Virtual IP , HIGH_PORT` > in order to establish the communication.

For External-2-Pod communication with API Gateway scenarios, you can use only the *Hostname* or *Control Plane IP Address* in order to establish the communication. Proper service (app feature) will be picked based on the *request format* defined in the Ingress object (explained later).

In order to simplify communication, for *container-2-container* and *Pod-2-Pod* you can replace IP address with the container/service name because Kubernetes provides the *DNS service*.

6.3 Storage Management

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

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

To overcome this problem, Kubernetes uses Volumes, storage abstractions that allow various storage technologies to be used by Kubernetes and offered to containers in Pods as storage media. A Volume is essentially a mount point on the container's file system backed by a storage medium. The storage medium, content and access mode are determined by the Volume Type.

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

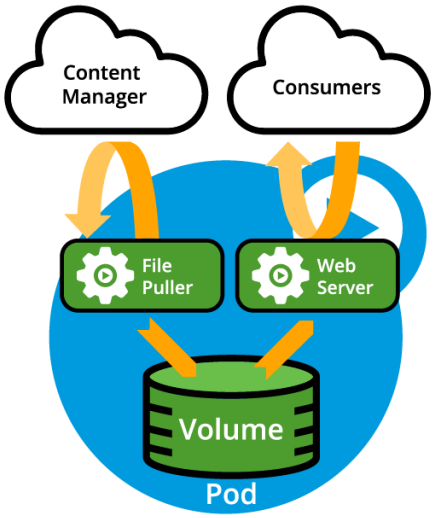

Figure 6.4: Volume schema

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

- **hostPath:** With the hostPath Volume Type, we can share a directory between the host and the Pod. If the Pod is terminated, the content of the Volume is still available on the host.
- **gcePersistentDisk:** With the gcePersistentDisk Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod
- **configMap:** With configMap objects, we can provide configuration data, or shell commands and arguments into a Pod.
- **secret:** With the secret Volume Type, we can pass sensitive information, such as passwords, to Pods.

6.3.1 Persistent Volume

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

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

A *Persistent Volume* is a storage abstraction backed by several storage technologies, which could be local to the host where the Pod is deployed with its application container(s), network attached storage, cloud storage, or a distributed

storage solution. A Persistent Volume is statically provisioned by the cluster administrator.

A *PersistentVolumeClaim* (PVC) is a request for storage by a user. Users request for PersistentVolume resources based on storage class, access mode, size, and optionally volume mode. There are four access modes: ReadWriteOnce (read-write by a single node), ReadOnlyMany (read-only by many nodes), ReadWriteMany (read-write by many nodes), and ReadWriteOncePod (read-write by a single pod). The optional volume modes, filesystem or block device, allow volumes to be mounted into a pod's directory or as a raw block device respectively. Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.

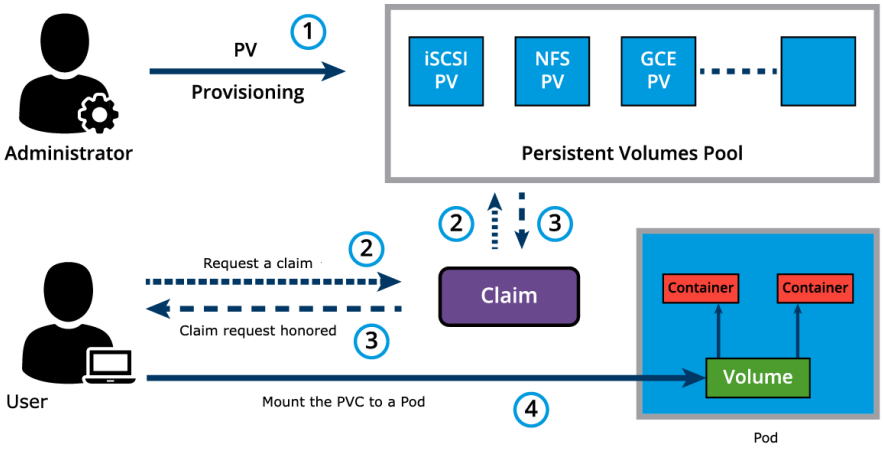

Figure 6.5: Storage Provisioning

An example of *hostPath* volume definition in a deployment controller is listed below:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    creationTimestamp: null
5    labels:
6      app: blue-app
7    name: blue-app
8  spec:
9    replicas: 1
10   selector:
11     matchLabels:
12       app: blue-app
13   strategy: {}
14   template:
15     metadata:
16       creationTimestamp: null
17       labels:
18         app: blue-app
19         type: canary
20     spec:
21       volumes:
```

```
22        - name: host-volume
23          hostPath:
24            path: /home/docker/blue-shared-volume
25        containers:
26        - image: nginx
27          name: nginx
28          ports:
29          - containerPort: 80
30          volumeMounts:
31          - mountPath: /usr/share/nginx/html
32            name: host-volume
33        - image: debian
34          name: debian
35          volumeMounts:
36          - mountPath: /host-vol
37            name: host-volume
38          command: ["/bin/sh", "-c", "echo Welcome to BLUE App! > /
                host-vol/index.html ; sleep infinity"]
39  status: {}
```

6.4 Configuration

While deploying an application, we may need to pass such runtime parameters like configuration details, permissions, passwords, keys, certificates, or tokens.

Similarly, when we want to pass sensitive information, we can use the Secret API resource. In this chapter, we will explore ConfigMaps and Secrets.

6.4.1 ConfigMaps

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

An example of ConfigMap object is listed below:

```
1 apiVersion: v1
2 kind: ConfigMap
3 metadata:
4   name: customer1
5 data:
6   TEXT1: "Customer1_Company"
7   TEXT2: "Welcomes You"
8   COMPANY: "Customer1 Company Technology Pct. Ltd."
```

After creating map with the *apply* command of `kubectl`, you have to include the configuration in the container's Pod Template definition.

An example of this is listed below:

```
1 ...
2   containers:
3   - name: myapp-full-container
4     image: myapp
```

```
5     envFrom:
6     - configMapRef:
7       name: full-config-map
8 ...
```

Another option is reference specific keys of the configmap as container enviroment variables:

```
1 ...
2   containers:
3   - name: myapp-specific-container
4     image: myapp
5     env:
6     - name: SPECIFIC_ENV_VAR1
7       valueFrom:
8         configMapKeyRef:
9           name: config-map-1
10          key: SPECIFIC_DATA
11    - name: SPECIFIC_ENV_VAR2
12      valueFrom:
13        configMapKeyRef:
14          name: config-map-2
15          key: SPECIFIC_INFO
16 ...
```

6.4.2 Secrets

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

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

It is important to keep in mind that by default, the Secret data is stored as plain text inside etcd, therefore administrators must limit access to the API server and etcd. However, Secret data can be encrypted at rest while it is stored in etcd, but this feature needs to be enabled at the API server level.

An example of Secret object definition is listed below:

```
1 apiVersion: v1
2 kind: Secret
3 metadata:
4   name: my-password
5 type: Opaque
6 stringData:
7   password: mysqlpassword
```

Usage of this secret is demostrated in the following deployment definition:

```
1  ....
2  spec:
3    containers:
4    - image: wordpress:4.7.3-apache
5      name: wordpress
6      env:
7      - name: WORDPRESS_DB_PASSWORD
8        valueFrom:
9          secretKeyRef:
10           name: my-password
11           key: password
12 ....
```

6.5 API Gateway

In an earlier chapter, we saw how we can access our deployed containerized application from the external world via Services.

In this chapter, we will explore the Ingress API resource, which represents another layer of abstraction, deployed in front of the Service API resources, offering a unified method of managing access to our applications from the external world.

6.5.1 Motivation & Definition

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

According to kubernetes.io: An Ingress is a collection of rules that allow inbound connections to reach the cluster Services. To allow the inbound connection to reach the cluster Services, Ingress configures a Layer 7 HTTP/HTTPS *load balancer* for Services.

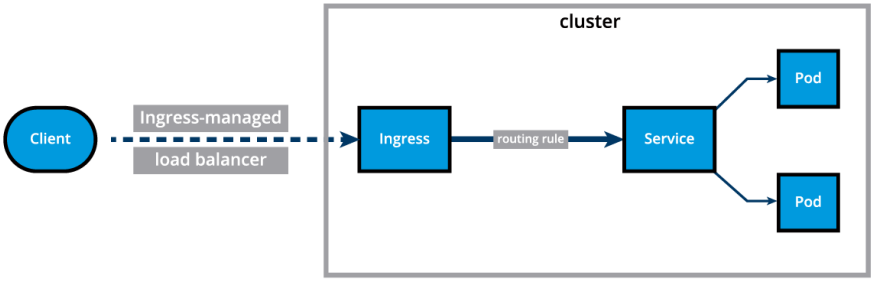

Figure 6.6: Ingress API Gateway Schema

6.5.2 Name-Based Virtual Hosting

With Ingress, users do not connect directly to a Service. Users reach the Ingress endpoint, and, from there, the request is forwarded to the desired Service.

A particular example of Ingress implementation is the *Name-Based Virtual Hosting*. It implies the mapping among *some host names* and *some services at proper ports on our application*. This means that client requests are routed to proper services based on the *host name* value. In addiction, *host names* will become the access points to our application features employed by clients.

An example of that kind of ingress object is the following snippet:

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    annotations:
5      kubernetes.io/ingress.class: "nginx"
6    name: virtual-host-ingress
7    namespace: default
8  spec:
9    rules:
10   - host: blue.example.com
11     http:
12       paths:
13       - backend:
14           service:
15             name: webserver-blue-svc
16             port:
17               number: 80
18         path: /
19         pathType: ImplementationSpecific
20   - host: green.example.com
21     http:
22       paths:
23       - backend:
24           service:
25             name: webserver-green-svc
26             port:
27               number: 80
28         path: /
29         pathType: ImplementationSpecific
```

In the example above, user requests to both `blue.example.com` and `green.example.com` would go to the same Ingress endpoint, and, from there, they would be forwarded to `webserver-blue-svc`, and `webserver-green-svc`, respectively.

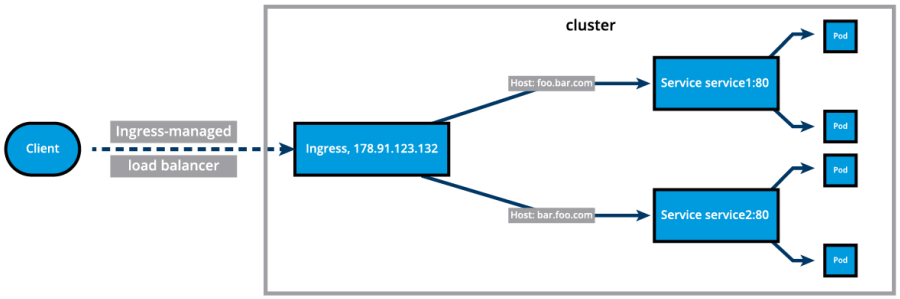

Figure 6.7: Ingress API Gateway Name-Based Virtual Hosting

6.5.3 Fanout

We can also define Fanout Ingress, this means that http requests to our application are routed to a specific service at a specific port based on a *filtering criteria* on the *http request*.

In the example definition and the diagram below, fanout ingress implies that when requests to `example.com/blue` and `example.com/green` would be forwarded to `webserver-blue-svc` and `webserver-green-svc`, respectively:

```
1  apiVersion: networking.k8s.io/v1
2  kind: Ingress
3  metadata:
4    annotations:
5      kubernetes.io/ingress.class: "nginx"
6    name: fan-out-ingress
7    namespace: default
8  spec:
9    rules:
10   - host: example.com
11     http:
12       paths:
13       - path: /blue
14         backend:
15           service:
16             name: webserver-blue-svc
17             port:
18               number: 80
19         pathType: ImplementationSpecific
20       - path: /green
21         backend:
22           service:
23             name: webserver-green-svc
24             port:
25               number: 80
26         pathType: ImplementationSpecific
```

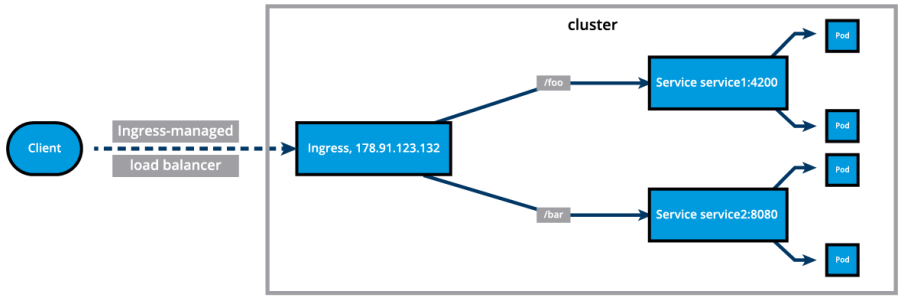

Figure 6.8: Ingress API Gateway Fanout

6.5.4 Ingress Controller

An Ingress Controller is an application watching the Control Plane Node's API server for changes in the Ingress resources and updates the Layer 7 Load Balancer accordingly. Ingress Controllers are also know as Controllers, Ingress Proxy, Service Proxy, Revers Proxy, etc. Kubernetes supports an array of Ingress Controllers, and, if needed, we can also build our own. GCE L7 Load Balancer Controller and Nginx Ingress Controller are commonly used Ingress Controllers. Other controllers are Contour, HAProxy Ingress, Istio Ingress, Kong, Traefik, etc. In order to ensure that the ingress controller is watching its corresponding ingress resource, the ingress resource definition manifest needs to include an ingress class annotation with the name of the desired controller kubernetes.io/ingress.class: "nginx" (for an nginx ingress controller).

Chapter 7

Kubernetes Misc

So far, in this course, we have spent most of our time understanding the basic Kubernetes concepts and simple workflows to build a solid foundation.

To support enterprise-class production workloads, Kubernetes also supports multi-node pod controllers, stateful application controllers, batch controllers, auto-scaling, resource and quota management, package management, security contexts, network and security policies, etc. In this chapter, we will briefly cover a limited number of such advanced topics, since diving into specifics would be out of scope for this course.

7.1 Annotations

With Annotations, we can attach arbitrary non-identifying metadata to any objects, in a key-value format. Unlike Labels, annotations are not used to identify and select objects. Annotations can be used to: Store build/release IDs, PR numbers, git branch, etc.

7.2 Quota and Limits Management

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

An additional resource that helps limit resources allocation to pods and containers in a namespace, is the LimitRange, used in conjunction with the ResourceQuota API resource.

7.3 Autoscaling

While it is fairly easy to manually scale a few Kubernetes objects, this may not be a practical solution for a production-ready cluster where hundreds or thousands of objects are deployed. We need a dynamic scaling solution which

adds or removes objects from the cluster based on resource utilization, availability, and requirements.

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

7.4 Jobs

A Job creates one or more Pods to perform a given task. The Job object takes the responsibility of Pod failures. It makes sure that the given task is completed successfully. Once the task is complete, all the Pods have terminated automatically

Starting with the Kubernetes 1.4 release, we can also perform Jobs at scheduled times/dates with CronJobs, where a new Job object is created about once per each execution cycle.

7.5 Kubernetes Federation

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

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

7.6 Security Contexts and Pod Security Admission

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

In order to apply security settings to multiple Pods and Containers cluster-wide, we can use the Pod Security Admission, a built-in admission controller for Pod Security that is enabled by default in the API Server.

7.7 Network Policies

Kubernetes was designed to allow all Pods to communicate freely, without restrictions, with all other Pods in cluster Namespaces. In time it became clear

that it was not an ideal design, and mechanisms needed to be put in place in order to restrict communication between certain Pods and applications in the cluster Namespace. Network Policies are sets of rules which define how Pods are allowed to talk to other Pods and resources inside and outside the cluster. Pods not covered by any Network Policy will continue to receive unrestricted traffic from any endpoint.

7.8 Monitoring & Logging

In Kubernetes, we have to collect resource usage data by Pods, Services, nodes, etc., to understand the overall resource consumption and to make decisions for scaling a given application.

Another important aspect for troubleshooting and debugging is logging, in which we collect the logs from different components of a given system. In Kubernetes, we can collect logs from different cluster components, objects, nodes, etc. Unfortunately, however, Kubernetes does not provide cluster-wide logging by default, therefore third party tools are required to centralize and aggregate cluster logs. A popular method to collect logs is using Elasticsearch together with Fluentd with custom configuration as an agent on the nodes.

The third-party troubleshooting tools are addressing a shortcoming of Kubernetes with regards to its logging capability. Although we can extract container logs from the cluster, we are limited only to logs of currently running containers, and in the case of several consecutive container restarts due to failures - the logs of the very last failed container (using the `-p` or `--previous` flags).

7.9 Helm

To deploy a complex application, we use a large number of Kubernetes manifests to define API resources such as Deployments, Services, PersistentVolumes, PersistentVolumeClaims, Ingress, or ServiceAccounts. It can become counter productive to deploy them one by one. We can bundle all those manifests after templatizing them into a well-defined format, along with other metadata. Such a bundle is referred to as Chart. These Charts can then be served via repositories, such as those that we have for rpm and deb packages.

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

Helm is a CLI client that may run side-by-side with kubectl on our workstation, that also uses kubeconfig to securely communicate with the Kubernetes API server.

The helm client queries the Chart repositories for Charts based on search parameters, downloads a desired Chart, and then it requests the API server to deploy in the cluster the resources defined in the Chart.

7.10 Application Deploy Strategies

A method presented earlier for new application release rollouts was the Rolling Update mechanism supported by the Deployment operator. The Rolling

Update mechanism, and its reverse - the Rollback, are practical methods to manage application updates by allowing one single controller, the Deployment, to handle all the work it involves. However, while transitioning between the old and the new versions of the application replicas, the Service exposing the Deployment eventually forwards traffic to all replicas, old and new, without any possibility for the default Service to isolate a subset of the Deployment's replicas.

Because of the traffic routing challenges these update mechanisms introduce, many users may steer away from the one Deployment and one Service model, and embrace more complex deployment mechanism alternatives.

The Canary strategy runs two application releases simultaneously managed by two independent Deployment controllers, both exposed by the same Service.

The users can manage the amount of traffic each Deployment is exposed to by separately scaling up or down the two Deployment controllers, thus increasing or decreasing the number of their replicas receiving traffic.

The Blue/Green strategy runs the same application release or two releases of the application on two isolated environments, but only one of the two environments is actively receiving traffic, while the second environment is idle, or may undergo rigorous testing prior to shifting traffic to it. This strategy would also require two independent Deployment controllers, each exposed by their dedicated Services, however, a traffic shifting mechanism is also required. Typically, the traffic shifting can be implemented with the use of an Ingress.

Chapter 8

Recurrent Commands

8.1 Cluster Management

8.1.1 Start & Stop

- Start a previously created Cluster

```
minikube -p PROFILE_NAME
```

- Start a new Cluster with default config

```
minikube start
```

- Start a new Cluster with default config creating a profile

```
minikube start ... --profile PROF_NAME
```

- Start a new Cluster with a particular Kubernetes version

```
minikube start --kubernetes-version=v1.25.1 --profile
    PROF_NAME
```

- Start a new Cluster with docker isolation driver

```
minikube start --driver=docker --profile PROF_NAME
```

- Start a new Cluster specifying number of nodes

```
minikube start --nodes=2 --profile PROF_NAME
```

- Start a new Cluster specifying number of CPUs

```
minikube start --cpus=2 --profile PROF_NAME
```

- Start a new Cluster specifying memory size

```
minikube start --memory=4g --profile PROF_NAME
```

- Start a new Cluster specifying disks sizes

```
minikube start --disk-size=10g --profile PROF_NAME
```

- Start a new Cluster specifying container network interface

```
minikube start --cni=calico --profile PROF_NAME
```

- Start a new Cluster specifying container runtime

```
minikube start --container-runtime=cri-o --profile
    PROF_NAME
```

- Start a new Cluster enabling all GPU available

```
minikube start --gpus all --profile PROF_NAME
```

- Stop Cluster a cluster

```
minikube stop -p PROFILE_NAME
```

8.1.2 Info

- Get clusters list (list of profiles)

```
minikube profile list
```

- Get status of a cluster

```
minikube status -p PROFILE
```

- Get Minikube version

```
minikube version
```

- Kubectl version

```
kubectl version --client
```

- Get node list by cluster

```
minikube node list -p PROFILE
```

- Get control plane node IP of a cluster

```
minikube -p PROFILE ip
```

- Get IP of a node in a cluster

```
minikube -p PROFILE ip -n NODE_NAME
```

- Get Kubectl config

```
kubectl config view
```

- Get cluster info with Kubectl

```
kubectl cluster-info
```

8.1.3 Manage

- Set a cluster (profile) as target of the minikube command

```
minikube profile PROFILE_NAME
```

- Copy file from host to a minikube cluster node

```
minikube cp <source file path> <target node name>:<
    target file absolute path>
```

- Change the default memory limit (requires a restart)

```
minikube config set memory 9001
```

- Delete a cluster

```
minikube delete -p PROFILE
```

- Delete all of the minikube clusters

```
minikube delete --all
```

- Open an SSH shell on control plane node

```
minikube ssh
```

- Open an SSH shell on a particular node

```
minikube ssh -n NODE
```

- List all services providing external URL access if any

```
minikube service --all
```

- Enable Ingress Controller (it is disabled for default and it is needed for API Gateway)

```
minikube addons enable ingress
```

- List minikube addons

```
minikube addons list
```

- Get log of a container

```
kubectl logs pod-name -c container-name
```

- Get log of a failed container

```
kubectl logs pod-name -c container-name -p
```

- Execute a command in a container

```
kubectl exec pod-name -c container-name -- COMMAND
```

- Attach shell of a container to current terminal

```
kubectl exec pod-name -c container-name -it -- /bin/
    bash
```

- Get events of a Pod

```
kubectl get events -l SELECTOR
```

8.2 Namespaces

- View namespaces

```
kubectl get namespaces
```

- Create a namespace

```
kubectl create namespace NAME
```

- Reference a nemespace in any kubectl command

```
kubectl -n NAMESPACE ...
```

8.3 Pods

- View Pods

```
kubectl get pods
```

- View Pods with full info

```
kubectl get pods -o wide
```

- View Pods using selectors

```
kubectl get pods -l K=V
```

- Describe Pods state

```
kubectl describe pods NAME
```

- Delete a pod

```
kubectl delete pods NAME
```

8.4 Deployments

- View Deployments

```
kubectl get deploy
```

- View replica sets

```
kubectl get rs
```

- Describe Deploy state

```
kubectl describe deploy NAME
```

- View rollout history

```
kubectl rollout history deploy DEPLOY_NAME
```

- View detail of a revision

```
kubectl rollout history deploy DEPLOY_NAME --revision
    NUMBER
```

- Rollback to a particular revision

```
kubectl rollout undo deployment DEPL_NAME --to-revision
    =NUMBER
```

8.5 DaemonSets

- View Deployments

```
kubectl get ds
```

- Delete a dameonsets

```
kubectl delete daemonsets.apps DAMEON_NAME
```

8.6 Auth & Authorization

- Get current namespaces roles

```
kubectl -n NAMESPACE get roles
```

- Get current namespaces roles binding

```
kubectl -n NAMESPACE get rolebindings
```

8.7 Services

- Get current services and end-points

```
kubectl get svc,ep --show-labels
```

8.8 Ingress

- Get ingress(s)

```
kubectl get ingress
```

8.9 K8 Objects

- View basic application info by label in current namespace:
  - Ingress
  - Services
  - Endpoints
  - Deploy
  - Pods
  - ReplicaSets
  - DaemonSets
  - PersistentVolumes

  - PersistenVolumeClaims

```
kubectl get ingress,svc,ep,deploy,pods,rs,ds,
    persistentvolume,persistentvolumeclaims -l KEY=VAL
```

- View basic application info by label in all namespaces
  - Ingress
  - Services
  - Endpoints
  - Deploy
  - Pods
  - ReplicaSets
  - DaemonSets
  - PersistentVolumes
  - PersistenVolumeClaims

```
kubectl get ingress,svc,ep,deploy,pods,rs,ds,
    persistentvolume,persistentvolumeclaims -l KEY=VAL
      -A
```

- Create/Update a Kubernetes file defined Resource

```
kubectl apply -f FILE_NAME.yaml
```

- Replace an existing Kubernetes File defined Resource

```
kubectl replace --force -f FILE_NAME.yaml
```

- Delete an existing Kubernetes file defined Resource

```
kubectl delete -f FILE_NAME.yaml
```

- Describe a K8 Object

```
kubectl describe OBJECT_KIND OBJECT_NAME
```

- Delete a K8 Object

```
kubectl delete OBJECT_KIND OBJECT_NAME
```

Bibliography

[1] Kubernetes Documentation

[2] Minikube Documentation

[3] The Linux Foundation - Introduction to Kubernetes (code. LFS158x)