Container Orchestration :-

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

What is container orchestration :-

In the **quality assurance (QA) environments**, we can get away with running containers on a single host to develop and test applications. However, when we go to production, we do not have the same liberty, as we need to ensure that our applications:

- Are fault-tolerant
- Can scale, and do this on-demand
- Use resources optimally
- Can discover other applications automatically, and communicate with each other
- Are accessible from the external world
- Can update/rollback without any downtime.

Container orchestrators are the tools which group hosts together to form a cluster, and help us fulfill the requirements mentioned above.

Nowadays, there are many container orchestrators available, such as:

- **Docker Swarm**
  Docker Swarm is a container orchestrator provided by Docker, Inc. It is part of Docker Engine.
- **Kubernetes**
  Kubernetes was started by Google, but now, it is a part of the Cloud Native Computing Foundation project.
- **Mesos Marathon**
  Marathon is one of the frameworks to run containers at scale on Apache Mesos.

- **Amazon ECS**
  Amazon EC2 Container Service (ECS) is a hosted service provided by AWS to run Docker containers at scale on its infrastructrue.
- **Hashicorp Nomad**
  Nomad is the container orchestrator provided by HashiCorp.

Kubernetes :-

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

- **Automatic binpacking**
  Kubernetes automatically schedules the containers based on resource usage and constraints, without sacrificing the availability.
- **Self-healing**
  Kubernetes automatically replaces and reschedules the containers from failed nodes. It also kills and restarts the containers which do not respond to health checks, based on existing rules/policy.
- **Horizontal scaling**
  Kubernetes can automatically scale applications based on resource usage like CPU and memory. In some cases, it also supports dynamic scaling based on customer metrics.
- **Service discovery and Load balancing**
  Kubernetes groups sets of containers and refers to them via a Domain Name System (DNS). This DNS is also called a Kubernetes **service**. Kubernetes can discover these services automatically, and load-balance requests between containers of a given service.

- **Automated rollouts and rollbacks**
  Kubernetes can roll out and roll back new versions/configurations of an application, without introducing any downtime.
- **Secrets and configuration management**
  Kubernetes can manage secrets and configuration details for an application without re-building the respective images. With secrets, we can share confidential information to our application without exposing it to the stack configuration, like on GitHub.
- **Storage orchestration**
  With Kubernetes and its plugins, we can automatically mount local, external, and storage solutions to the containers in a seamless manner, based on software-defined storage (SDS).

- **Batch execution**
  Besides long running jobs, Kubernetes also supports batch execution.

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

User of Kubernetes :

Pearson

- Box
- eBay
- Wikimedia
- Huawei
- Haufe Group
- BlackRock
- BlaBlaCar
- And many more.

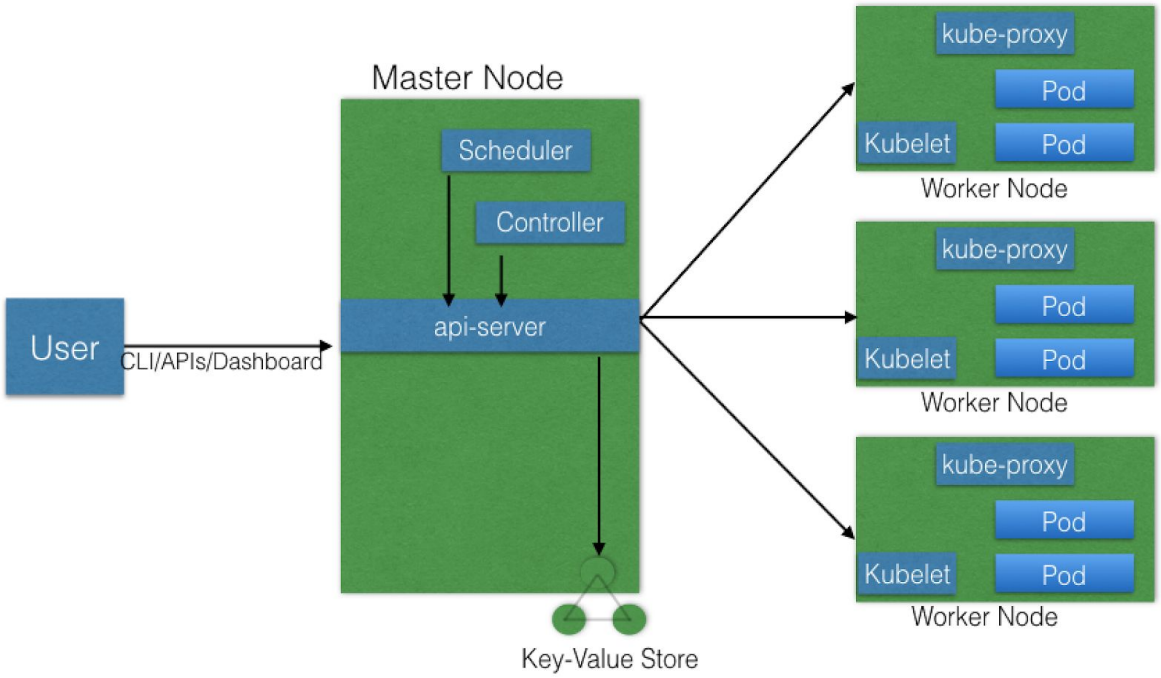
Kubernetes Architecture :

Introduction :

In this chapter, we will explore the **Kubernetes** architecture, the different components of the **master** and **worker nodes**, the cluster state management with **etcd** and the network setup requirements. We will also talk about the network specification called **Container Network Interface (CNI)**, which is used by Kubernetes.

At a very high level, Kubernetes has the following main components:

- One or more **master nodes**
- One or more **worker nodes**
- Distributed key-value store, like **etcd**.



Kubernetes Architecture

Kubernetes Master component:

A master node has the following components:

- API server
- Scheduler
- Controller manager
- etcd.

1. API Server :

All the administrative tasks are performed via the **API server** within the master node. A user/operator sends REST commands to the API server, which then validates and processes the requests. After executing the requests, the resulting state of the cluster is stored in the distributed key-value store.

2. Scheduler :

As the name suggests, the **scheduler** schedules the work to different worker nodes. The scheduler has the resource usage information for each worker node. It also knows about the constraints that users/operators may have set, such as scheduling work on a node that has the label `disk==ssd` set. Before scheduling the work, the scheduler also takes into

account the quality of the service requirements, data locality, affinity, anti-affinity, etc. The scheduler schedules the work in terms of Pods and Services.

3. Controller Manager :-

The **controller manager** manages different non-terminating control loops, which regulate the state of the Kubernetes cluster. Each one of these control loops knows about the desired state of the objects it manages, and watches their current state through the API server. In a control loop, if the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the current state is the same as the desired state.

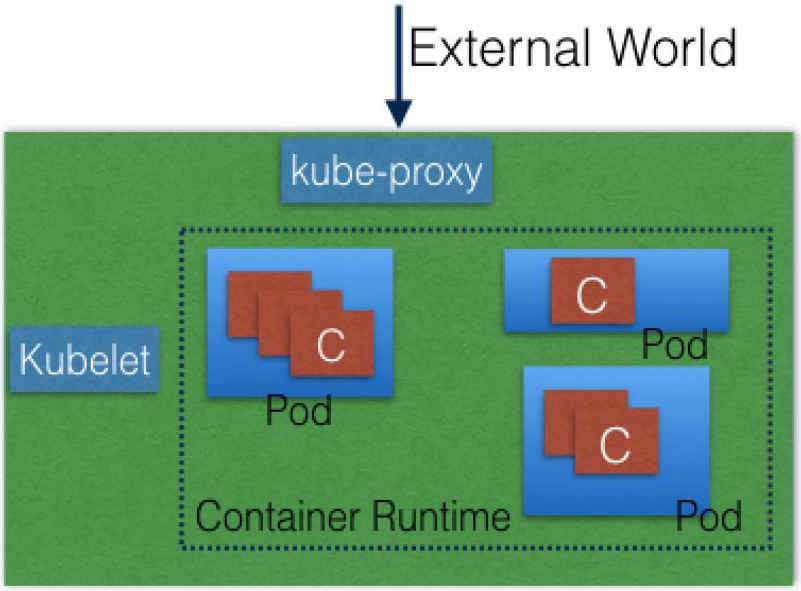
4. Etcd

Cluster state saved in key-value pair

5. worker node:

A **worker node** is a machine (VM, physical server, etc.) which runs the applications using Pods and is controlled by the master node. Pods are scheduled on the worker nodes, which have the necessary tools to run and connect them. A Pod is the scheduling unit in Kubernetes. It is a logical collection of one or more containers which are always scheduled together. We will explore them further in later chapters.

A **worker node** is a machine (VM, physical server, etc.) which runs the applications using Pods and is controlled by the master node. Pods are scheduled on the worker nodes, which have the necessary tools to run and connect them. A Pod is the scheduling unit in Kubernetes. It is a logical collection of one or more containers which are always scheduled together. We will explore them further in later chapters.



Kubernetes Worker Node

Worker Node Component :

A worker node has the following components:

- Container runtime
- kubelet
- kube-proxy.

1. Container runtime

To run and manage a container's lifecycle, we need a **container runtime** on the worker node. Some examples of container runtimes are:

- containerd
- rkt
- lxd.

Sometimes, Docker is also referred to as a container runtime, but to be precise, Docker is a platform which uses **containerd** as a container runtime.

2. Kubelet :

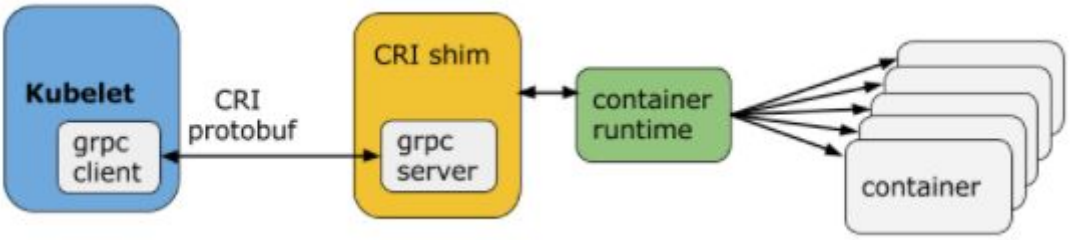
The **kubelet** is an agent which runs on each worker node and communicates with the master node. It receives the Pod definition via various means (primarily, through the API

server), and runs the containers associated with the Pod. It also makes sure that the containers which are part of the Pods are healthy at all times.

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

The **kubelet** is an agent which runs on each worker node and communicates with the master node. It receives the Pod definition via various means (primarily, through the API server), and runs the containers associated with the Pod. It also makes sure that the containers which are part of the Pods are healthy at all times.

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



Container Runtime Interface

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

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

Worker Node Components: kubelet: CRI shims

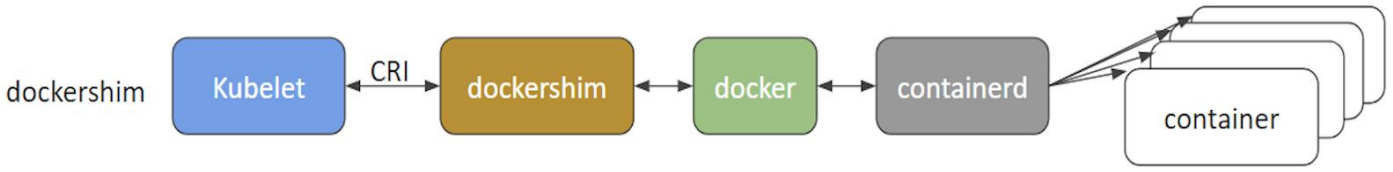
Below you will find some examples of CRI shims:

- **dockershim**
  With dockershim, containers are created using Docker installed on the

worker nodes. Internally, Docker uses containerd to create and manage containers

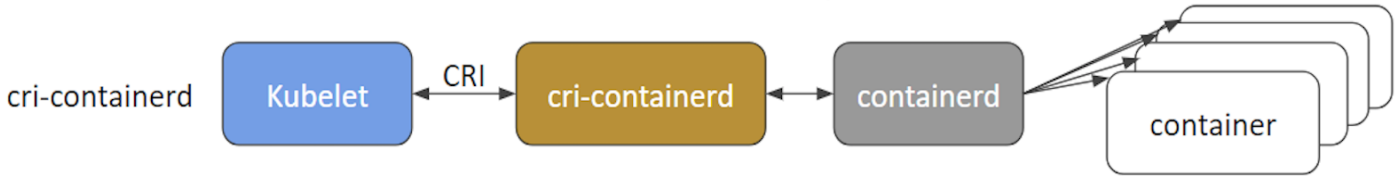
Below you will find some examples of CRI shims:

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

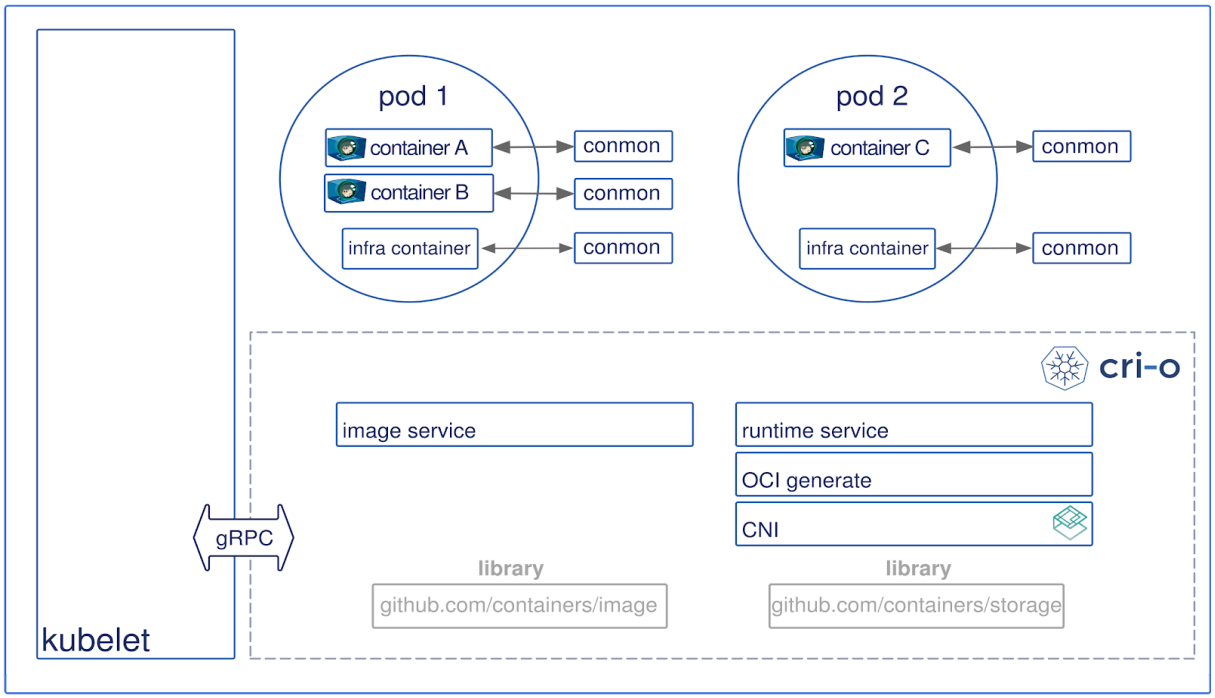


dockershim

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



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



kube-proxy :

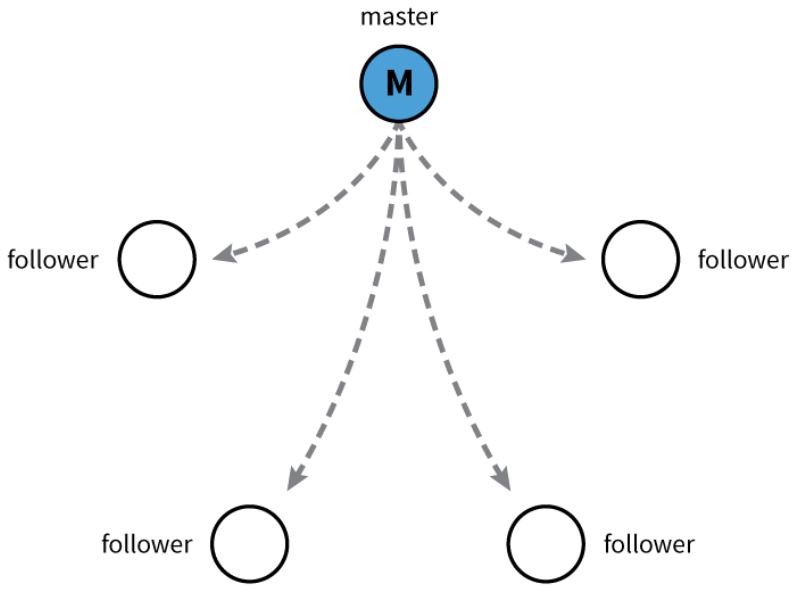
nstead of connecting directly to Pods to access the applications, we use a logical construct called a **Service** as a connection endpoint. A Service groups related Pods and, when accessed, load balances to them. We will talk more about Services in later chapters.

kube-proxy is the network proxy which runs on each worker node and listens to the API server for each Service endpoint creation/deletion. For each Service endpoint, kube-proxy sets up the routes so that it can reach to it.

Advanced component:

ETCD:-

As we mentioned earlier, Kubernetes uses **etcd** to store the cluster state. etcd is a distributed key-value store based on the Raft Consensus Algorithm. Raft allows a collection of machines to work as a coherent group that can survive the failures of some of its members. At any given time, one of the nodes in the group will be the master, and the rest of them will be the followers. Any node can be treated as a master.



Master and Followers

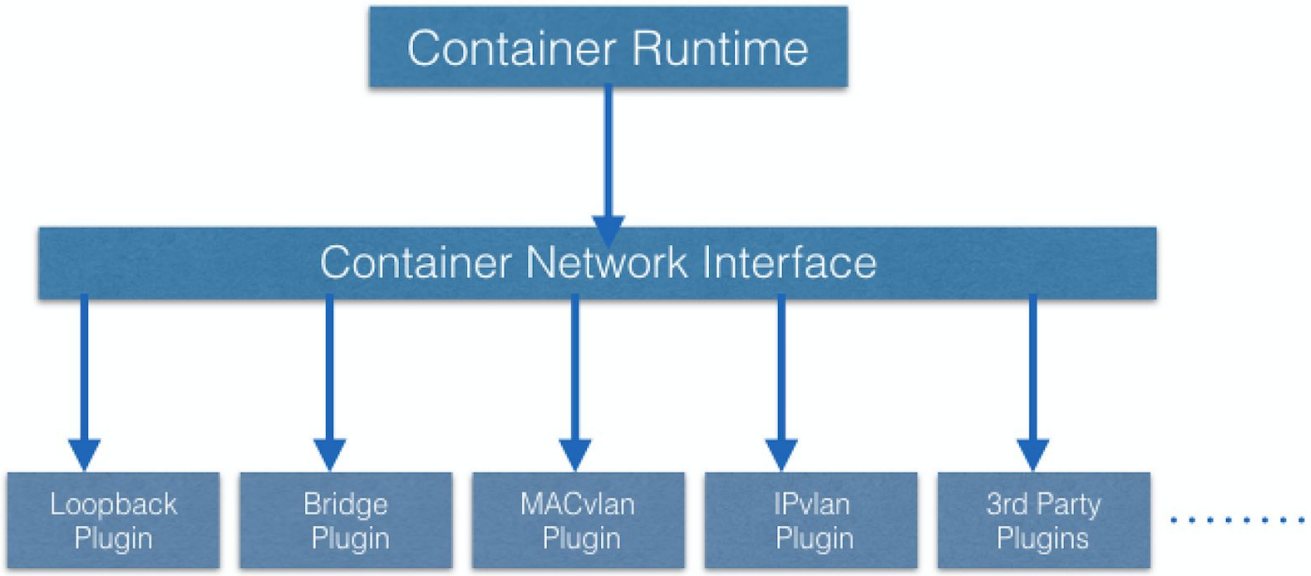
etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as subnets, ConfigMaps, Secrets, etc.

Assigning a Unique IP Address to Each Pod

In Kubernetes, each Pod gets a unique IP address. For container networking, there are two primary specifications:

- **Container Network Model (CNM)**, proposed by Docker
- **Container Network Interface (CNI)**, proposed by CoreOS.

Kubernetes uses CNI to assign the IP address to each Pod.



Container Network Interface (CNI)

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

Container-to-Container Communication Inside a Pod

With the help of the underlying host operating system, all of the container runtimes generally create an isolated network entity for each container that it starts. On Linux, that entity is referred to as a **network namespace**. These network namespaces can be shared across containers, or with the host operating system.

Inside a Pod, containers share the network namespaces, so that they can reach to each other via localhost.

Pod-to-Pod Communication Across Nodes

In a clustered environment, the Pods can be scheduled on any node. We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able

to reach any Pod. Kubernetes also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-to-Pod communication across hosts. We can achieve this via:

- Routable Pods and nodes, using the underlying physical infrastructure, like Google Kubernetes Engine
- Using Software Defined Networking, like Flannel, Weave, Calico, etc.

Communication Between the External World and Pods

By exposing our services to the external world with **kube-proxy**, we can access our applications from outside the cluster. We will have a complete chapter dedicated to this, so we will dive into this later.

Kubernetes setup :-

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

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

Kubernetes Installation Tools/Resources

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

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

Kubernetes Building Blocks:

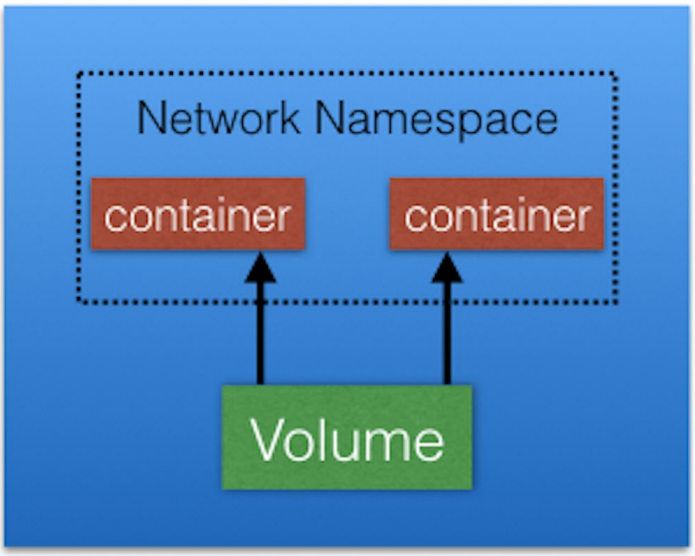
Introduction

In this chapter, we will explore the **Kubernetes object** model and discuss some of its building blocks, such as **Pods**, **ReplicaSets**, **Deployments**, **Namespaces**, etc. We will also discuss the role of **Labels** and **Selectors** when it comes to grouping objects together.

Pods

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

- Are scheduled together on the same host
- Share the same network namespace
- Mount the same external storage (volumes).



Pods

Pods are ephemeral in nature, and they do not have the capability to self-heal by themselves. That is why we use them with controllers, which can handle a Pod's replication, fault tolerance, self-heal, etc. Examples of controllers are Deployments, ReplicaSets, ReplicationControllers, etc. We attach the Pod's specification to other objects using Pods Templates

Labels

Labels are key-value pairs that can be attached to any Kubernetes objects (e.g. Pods). 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.

Labels

In the image above, we have used two Labels: **`app`** and **`env`**. Based on our requirements, we have given different values to our four Pods

Label Selectors

With Label Selectors, we can 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. With this type of selectors, we can use the `=`, `==`, or `!=` operators. For example, with `env==dev` we are selecting the objects where the `env` Label is set to `dev`.
- **Set-Based Selectors**
  Set-Based Selectors allow filtering of objects based on a set of values. With this type of Selectors, we can use the `in`, `notin`, and `exist` operators. For example, with `env in (dev,qa)`, we are selecting objects where the `env` Label is set to `dev` or `qa`.

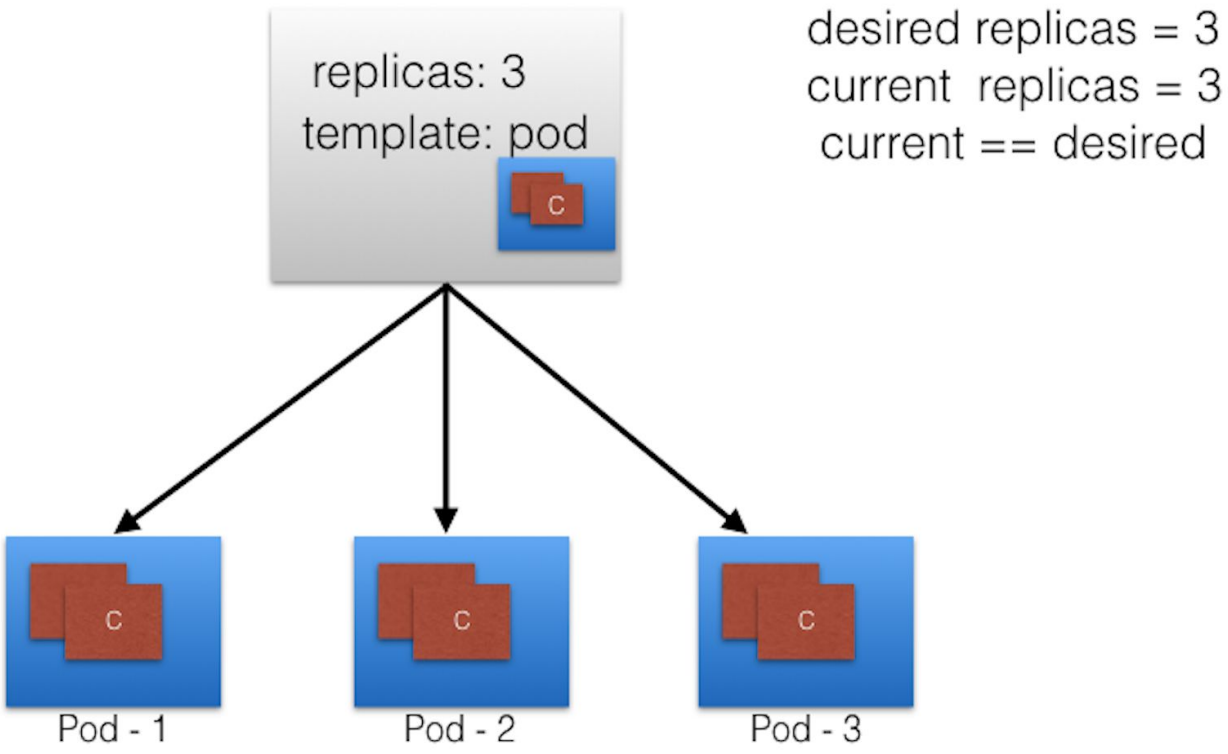
ReplicationControllers

A ReplicationController (rc) is a controller that is part of the master node's controller manager. It makes sure the specified number of replicas for a Pod is running at any given point in time. If there are more Pods than the desired count, the ReplicationController would kill the extra Pods, and, if there are less Pods, then the ReplicationController would create more Pods to match the desired count. Generally, we don't deploy a Pod independently, as it would not be able to re-start itself, if something goes wrong. We always use controllers like ReplicationController to create and manage Pods.

ReplicaSets I

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

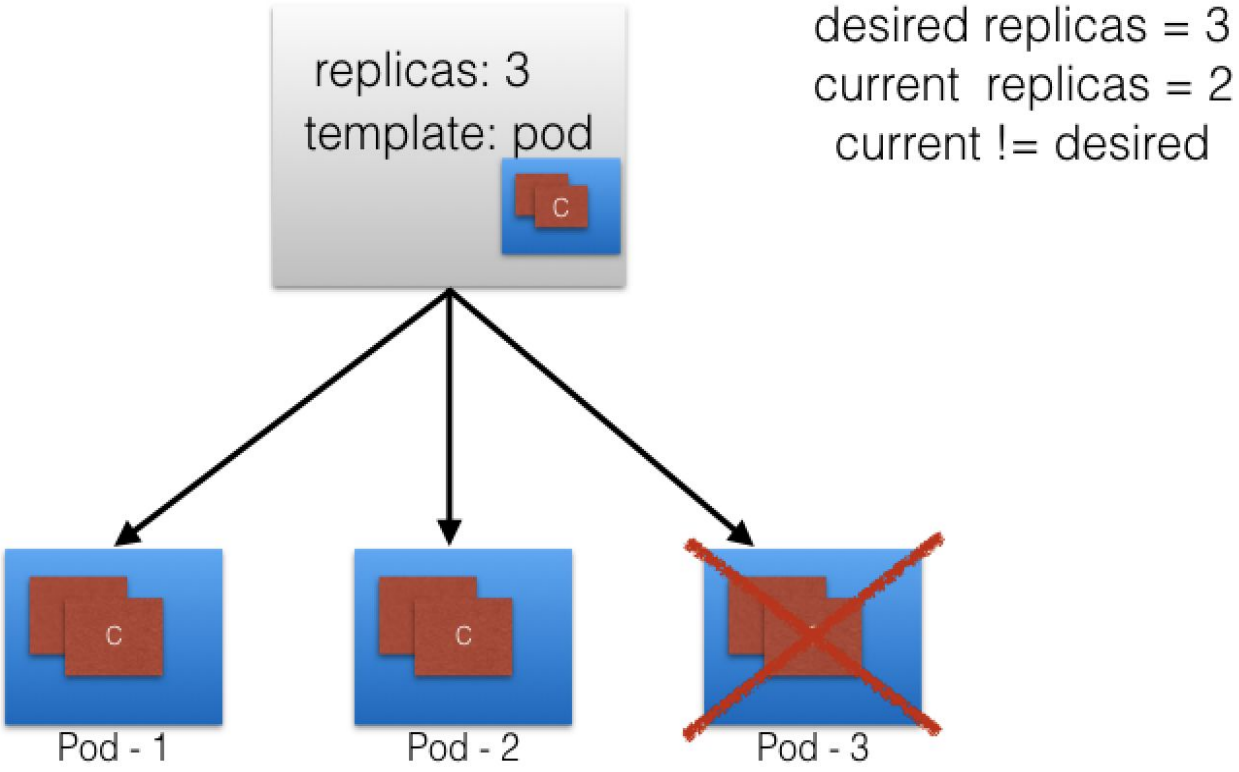
Next, you can see a graphical representation of a ReplicaSet, where we have set the replica count to 3 for a Pod.



ReplicaSet (Current State and Desired State Are the Same)

ReplicaSets II

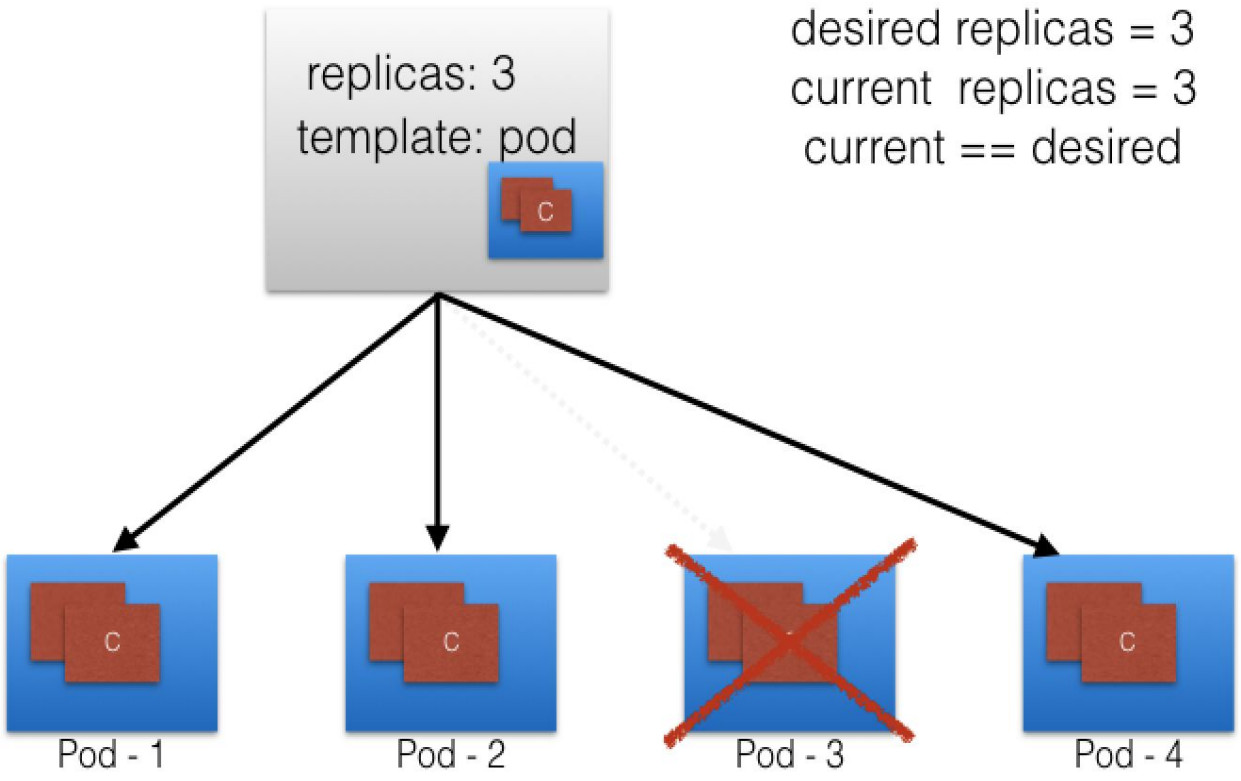
Now, let's suppose that one Pod dies, and our current state is not matching the desired state anymore.



ReplicaSet (Current State and Desired State Are Different)

ReplicaSets III

The ReplicaSet will detect that the current state is no longer matching the desired state. So, in our given scenario, the ReplicaSet will create one more Pod, thus ensuring that the current state matches the desired state.



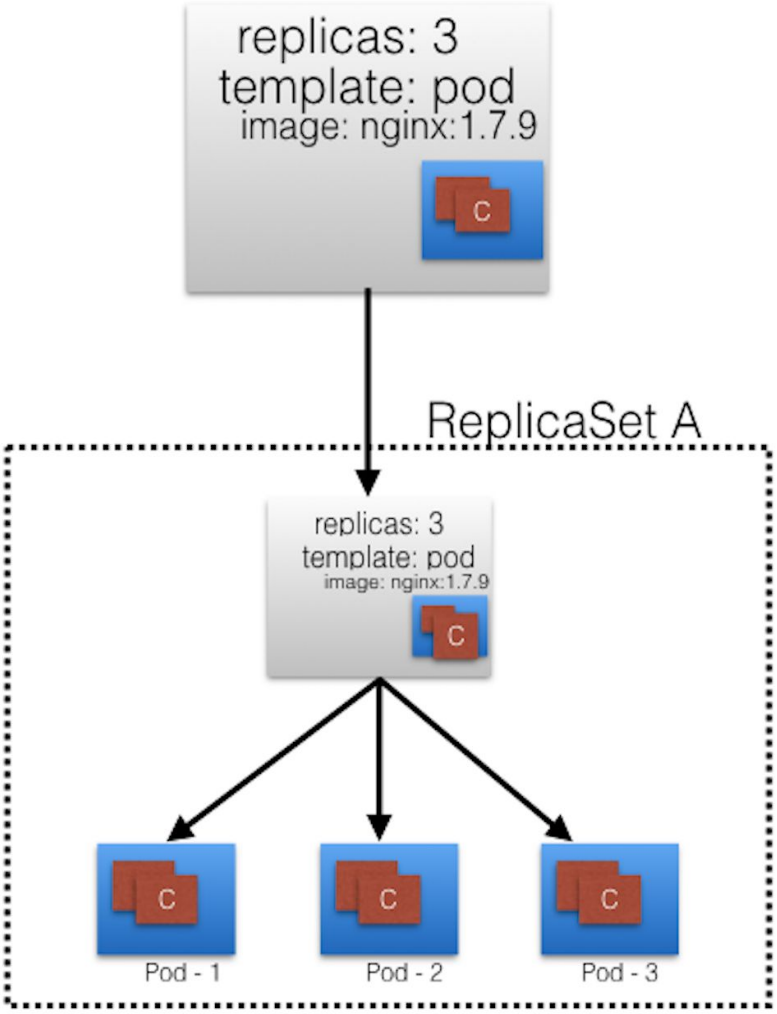
ReplicaSet (Creating a Pod to Match Current and Desired State)

ReplicaSets can be used independently, but they are mostly used by Deployments to orchestrate the Pod creation, deletion, and updates. A Deployment automatically creates the ReplicaSets, and we do not have to worry about managing them.

Deployments I

Deployment objects provide declarative updates to Pods and ReplicaSets. The DeploymentController is part of the master node's controller manager, and it makes sure that the current state always matches the desired state.

In the following example, we have a **Deployment** which creates a **`ReplicaSet A`**. **`ReplicaSet A`** then creates **3 Pods**. In each Pod, one of the containers uses the **`nginx:1.7.9`** image.

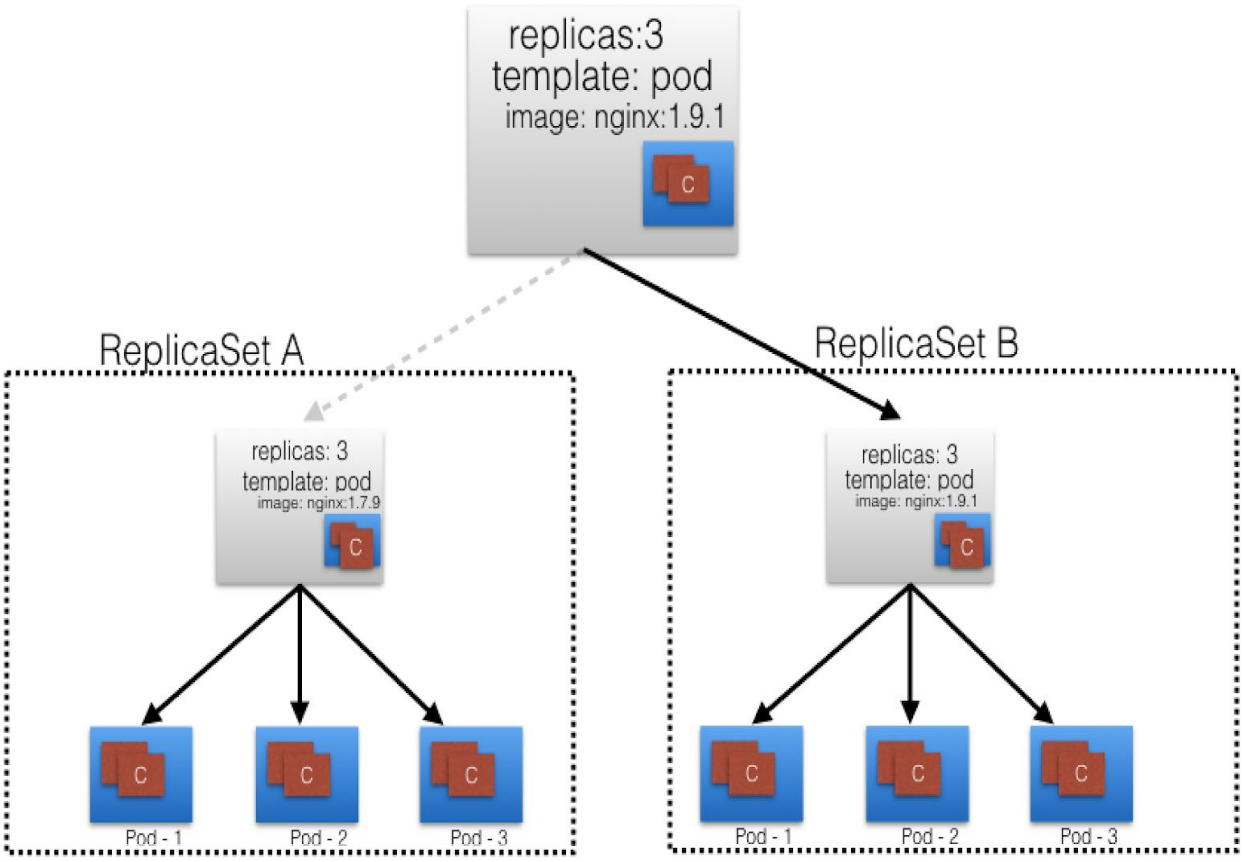


Deployment (ReplicaSet A Created)

Deployments II

Now, in the Deployment, we change the Pods Template and we update the image for the **`nginx`** container from **`nginx:1.7.9`** to **`nginx:1.9.1`**. As have modified the Pods Template, a new **`ReplicaSet B`** gets created. This process is referred to as a **Deployment rollout**.

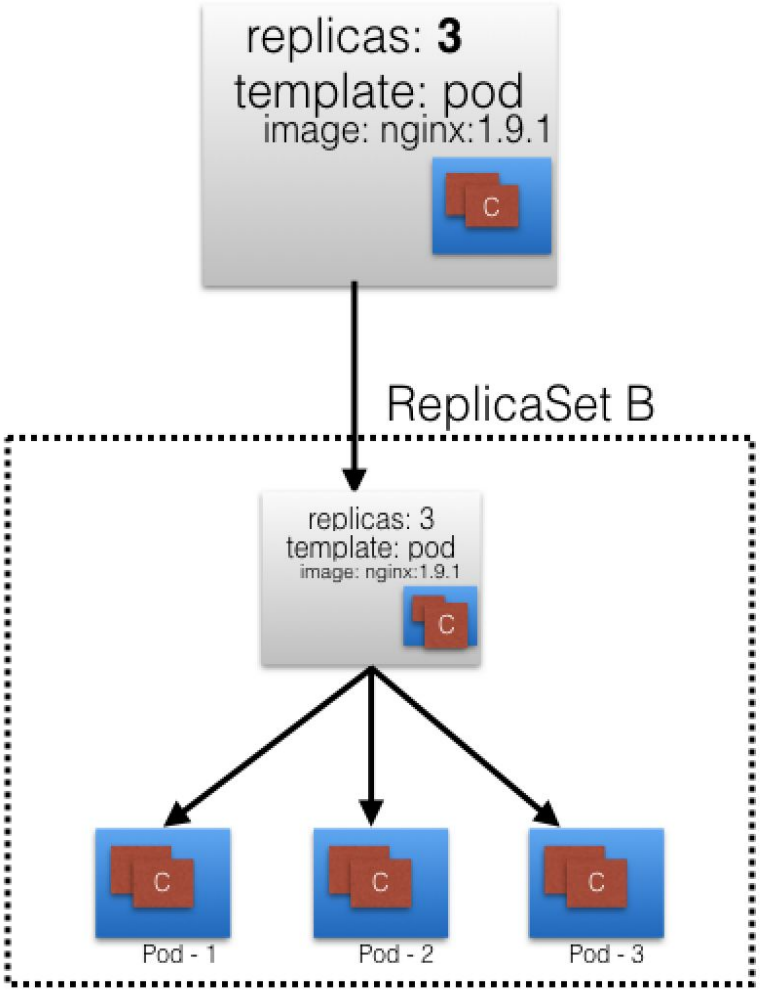
A rollout is only triggered when we update the Pods Template for a deployment. Operations like scaling the deployment do not trigger the deployment.



Deployment (ReplicaSet B Created)

Deployments III

Once **`ReplicaSet B`** is ready, the Deployment starts pointing to it.



Deployment Points to ReplicaSet B

On top of ReplicaSets, Deployments provide features like Deployment recording, with which, if something goes wrong, we can rollback to a previously known state.

Namespaces

If we have numerous users whom we would like to organize into teams/projects, we can partition the Kubernetes cluster into sub-clusters using Namespaces. The names of the resources/objects created inside a Namespace are unique, but not across Namespaces.

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

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

Generally, Kubernetes creates two default Namespaces: **`kube-system`** and **`default`.** The **`kube-system`** Namespace contains the objects created by the Kubernetes system. The **`default`** Namespace contains the objects which belong to any other Namespace. By default, we connect to the **`default`** Namespace. **`kube-public`** is a special Namespace, which is readable by all users and used for special purposes, like bootstrapping a cluster.

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

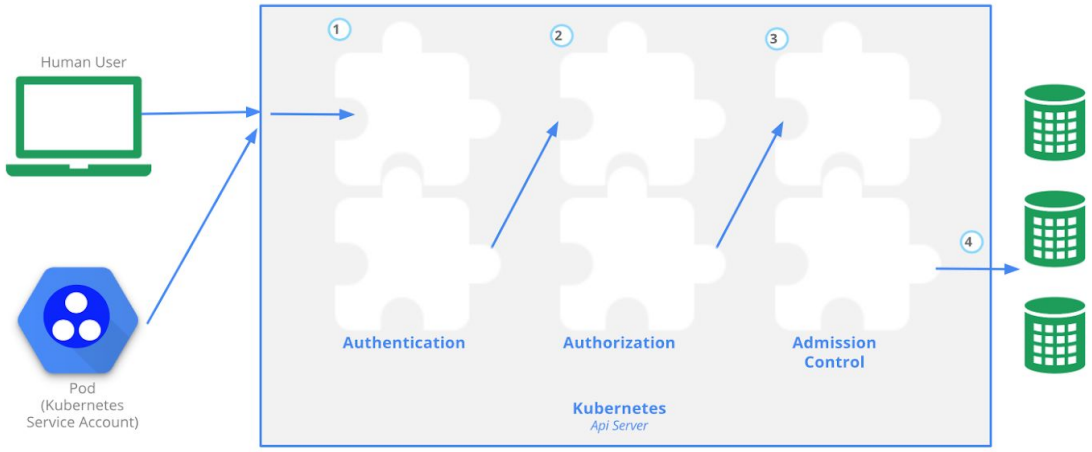
Authentication , Authorization and admission control

Authentication, Authorization, and Admission Control - Overview

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

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

The following image depicts the above stages:



Accessing the API

Authentication I

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

Kubernetes has two kinds of users:

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

If properly configured, Kubernetes can also support **anonymous requests**, along with requests from Normal Users and Service Accounts.

Authentication II

For authentication, Kubernetes uses different [authenticator modules]:

- **Client Certificates**
- To enable client certificate authentication, we need to reference a file containing one or more certificate authorities by passing the `--client-ca-file=SOMEFILE` option to the API server. The certificate authorities mentioned in the file would validate the client certificates presented to the API server. A demonstration video covering this topic is also available at the end of this chapter.
- **Static Token File**
- We can pass a file containing pre-defined bearer tokens with the `--token-auth-file=SOMEFILE` option to the API server. Currently, these tokens would last indefinitely, and they cannot be changed without restarting the API server.
-
- **Bootstrap Tokens**
- This feature is currently in an alpha status, and is mostly used for bootstrapping a new Kubernetes cluster.
- **Static Password File**
- It is similar to *Static Token File.* We can pass a file containing basic authentication details with the `--basic-auth-file=SOMEFILE` option. These credentials would last indefinitely, and passwords cannot be changed without restarting the API server.
-
- **Service Account Tokens**
- This is an automatically enabled authenticator that uses signed bearer tokens to verify the requests. These tokens get attached to Pods using the ServiceAccount Admission Controller, which allows in-cluster processes to talk to the API server.
- **OpenID Connect Tokens**
- OpenID Connect helps us connect with OAuth 2 providers, such as Azure Active Directory, Salesforce, Google, etc., to offload the authentication to external services.
-
- **Webhook Token Authentication**
- With Webhook-based authentication, verification of bearer tokens can be offloaded to a remote service.
- **Keystone Password**

- Keystone authentication can be enabled by passing the `--experimental-keystone-url=<AuthURL>` option to the API server, where `AuthURL` is the Keystone server endpoint.
-
- **Authenticating Proxy**
- If we want to program additional authentication logic, we can use an authenticating proxy.
-

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

Authorization I

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

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

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

Authorization II

Authorization modules (Part 1):

- **Node Authorizer**
- Node authorization is a special-purpose authorization mode which specifically authorizes API requests made by kubelets. It authorizes

the kubelet's read operations for services, endpoints, nodes, etc., and writes operations for nodes, pods, events, etc. For more details, please review the Kubernetes documentation.

- **Attribute-Based Access Control (ABAC) Authorizer**
- With the ABAC authorizer, Kubernetes grants access to API requests, which combine policies with attributes. In the following example, user *nkhare* can only read Pods in the Namespace `lfs158`.
-
- ```
  {
    "apiVersion": "abac.authorization.kubernetes.io/v1beta1",
   "kind": "Policy",
    "spec": {
     "user": "nkhare",
     "namespace": "lfs158",
       "resource": "pods",
       "readonly": true
     }
  }
  ```
-
- To enable the ABAC authorizer, we would need to start the API server with the `--authorization-mode=ABAC` option. We would also need to specify the authorization policy, like `--authorization-policy-file=PolicyFile.json`. For more details, please review the Kubernetes documentation.
- **Webhook Authorizer**
- With the Webhook authorizer, Kubernetes can offer authorization decisions to some third-party services, which would return *true* for successful authorization, and *false* for failure. In order to enable the Webhook authorizer, we need to start the API server with the `--authorization-webhook-config-file=SOME_FILENAME` option, where `SOME_FILENAME` is the configuration of the remote authorization service

Authorization III

Authorization modules (Part 2):

Role-Based Access Control (RBAC) Authorizer

In general, with RBAC we can regulate the access to resources based on the roles of individual users. In Kubernetes, we can have different roles that can be attached to subjects like users, service accounts, etc. While creating the roles, we restrict resource
  ```

access by specific operations, such as **`create`**, **`get`**, **`update`**, **`patch`**, etc. These operations are referred to as verbs.

In RBAC, we can create two kinds of roles:

Role

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

ClusterRole

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

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

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

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

There are two kinds of *RoleBindings*:

RoleBinding

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

Namespace resources defined in the ClusterRole within the RoleBinding's Namespace.

ClusterRoleBinding

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

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

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

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

To enable the RBAC authorizer, we would need to start the API server with the **`--authorization-mode=RBAC`** option. With the RBAC authorizer, we dynamically configure policies.

Admission Control

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

To use admission controls, we must start the Kubernetes API server with the `admission-control`, which takes a comma-delimited, ordered list of controller names, like in the following example:

```
--admission-control=NamespaceLifecycle,ResourceQuota,PodSecurityPolicy,DefaultStorageClass.
```

By default, Kubernetes comes with some built-in admission controllers

Services :-

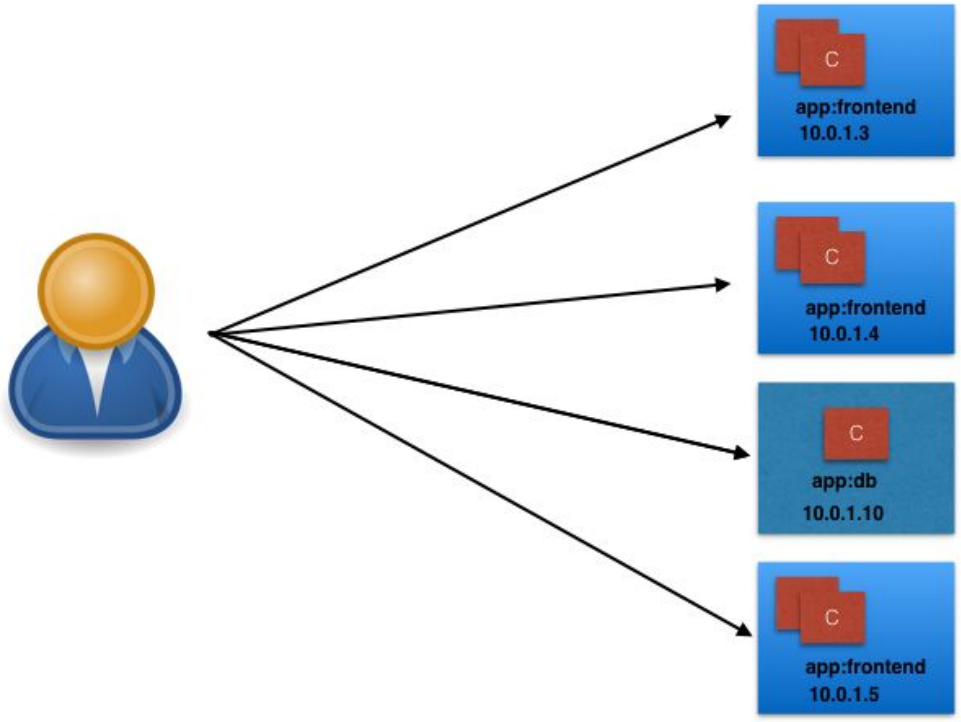
Introduction :

about **Services**, using which we can group Pods to provide common access points from the external world. We will learn about the **kube-proxy** daemon, which runs on each worker node to provide access to services.

Connecting Users to Pods

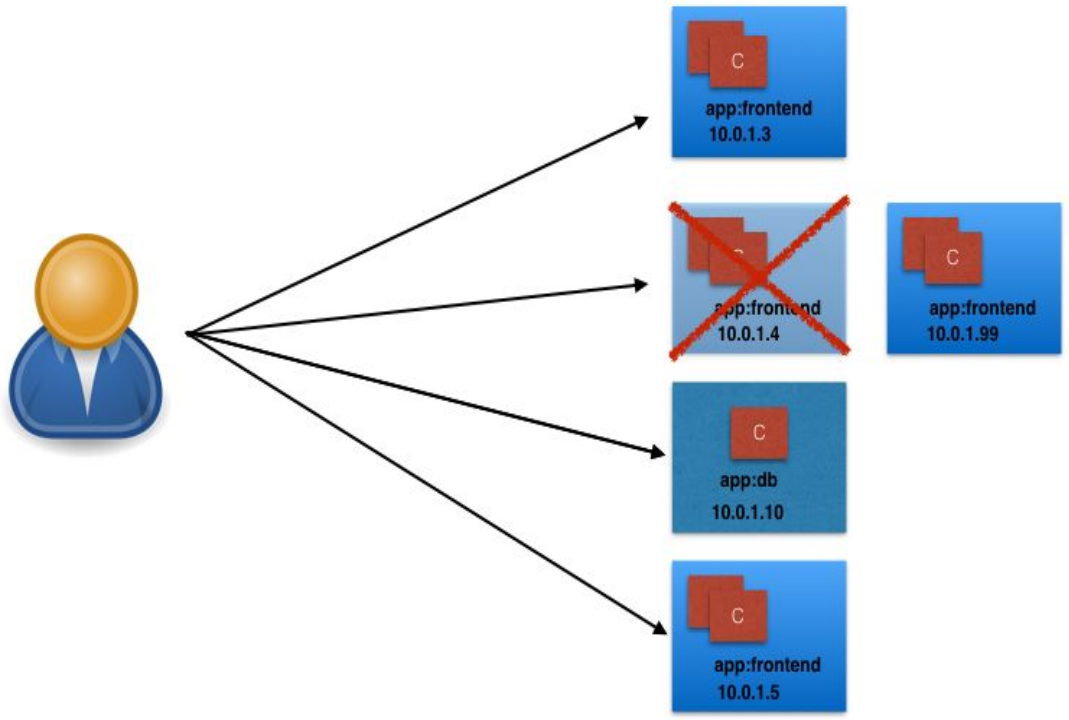
To access the application, a user/client needs to connect to the Pods. As Pods are ephemeral in nature, resources like IP addresses allocated to it cannot be static. Pods could die abruptly or be rescheduled based on existing requirements.

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



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

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

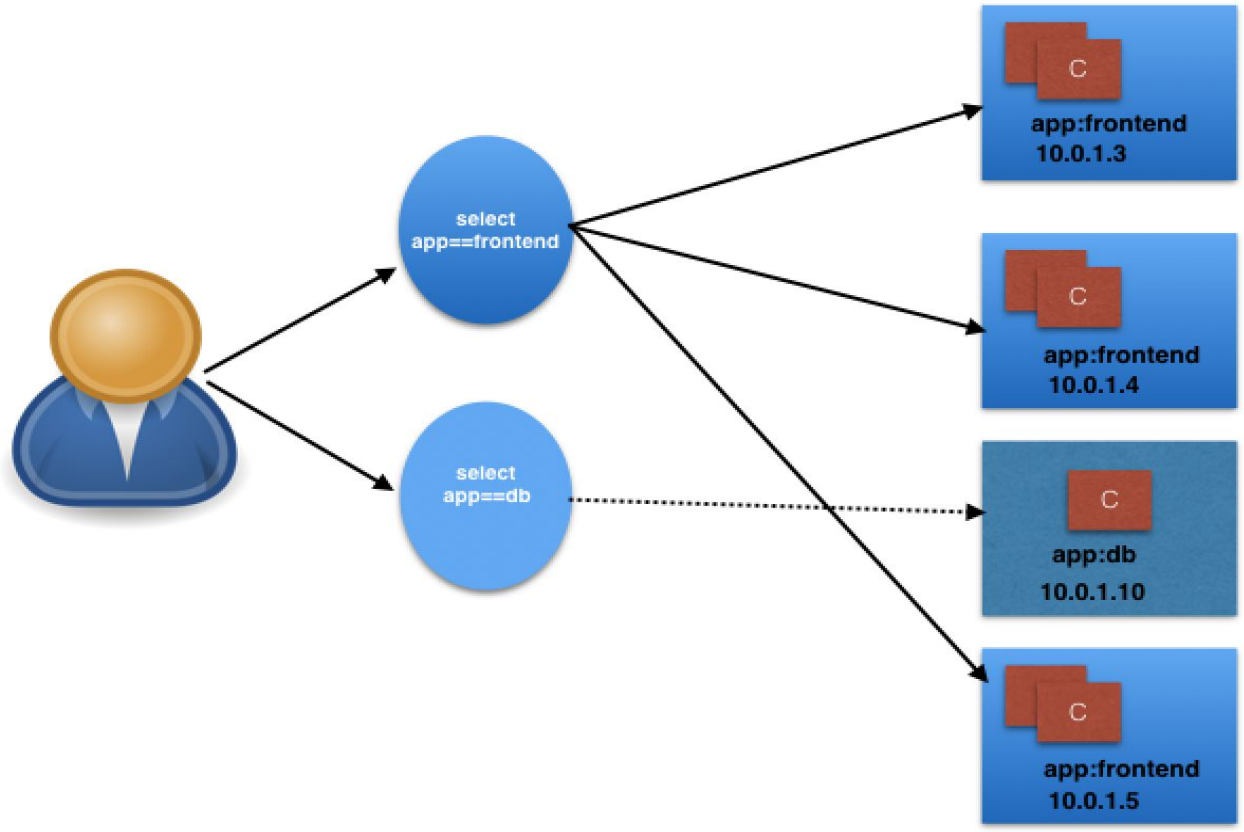


A New Pod Is Created After the Old One Dies Unexpectedly

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

Services

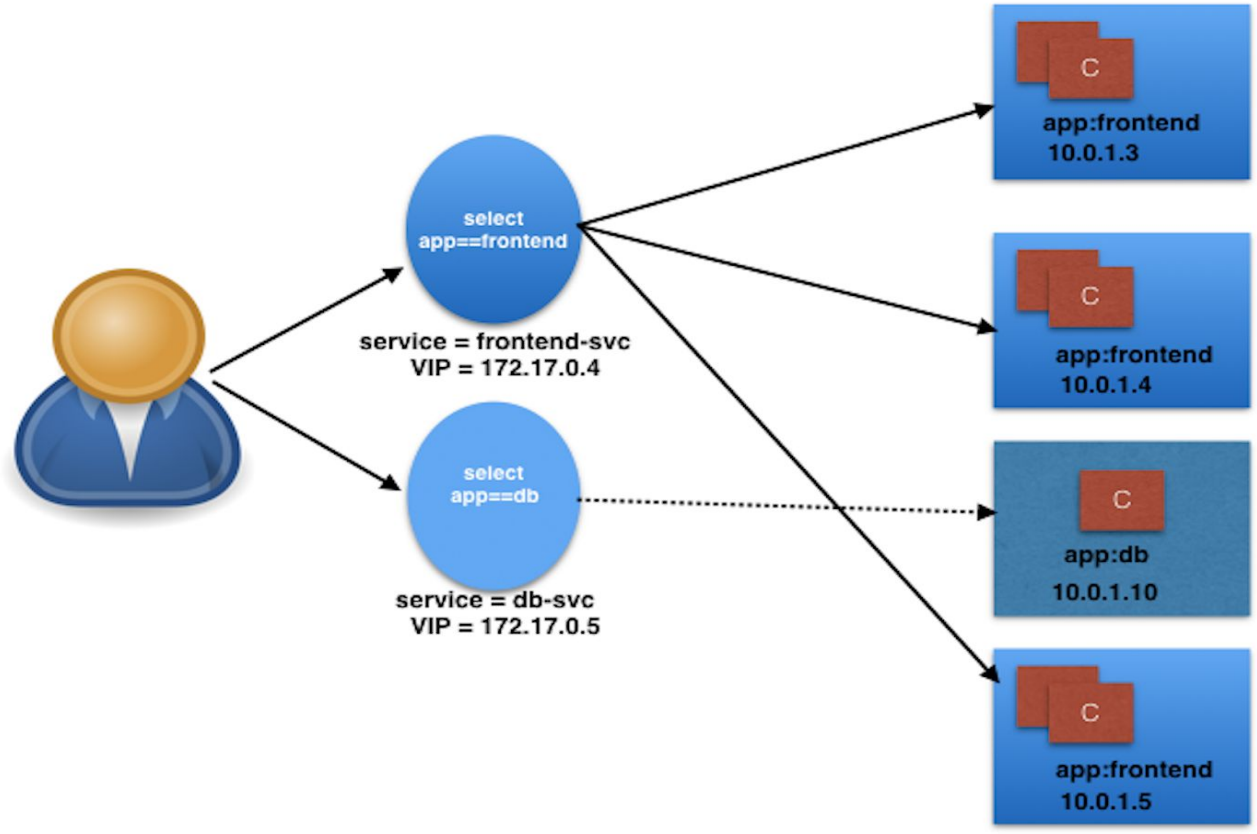
For example, in the following graphical representation we have used the `app` keyword as a Label, and `frontend` and `db` as values for different Pods.



Grouping of Pods using Labels and Selectors

Using selectors (**`app==frontend`** and **`app==db`**), we can group them into two logical groups: one with 3 Pods, and one with just one Pod.

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



Grouping of Pods using the Service object

Service Object Example

The following is an example of a Service object:

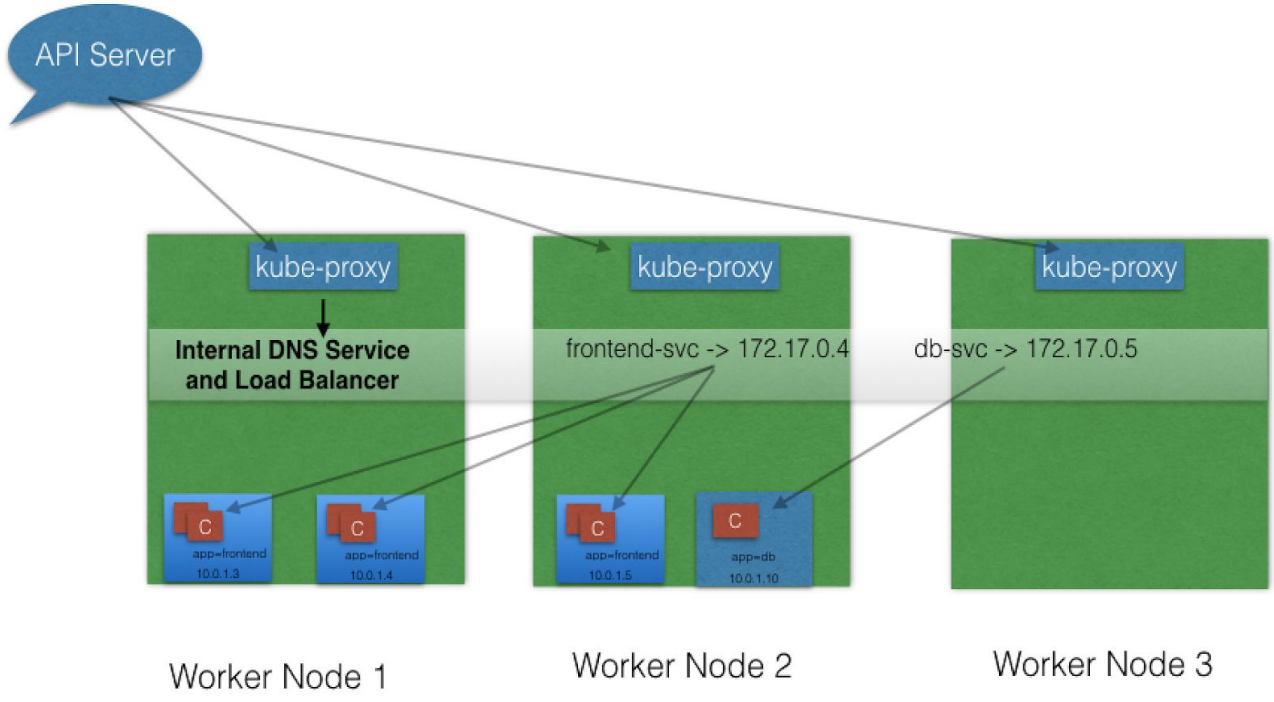
```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
```

```
  - protocol: TCP
    port: 80
    targetPort: 5000
```

In this example, we are creating a **`frontend-svc`** Service by selecting all the Pods that have the Label **`app`** set to the **`frontend`**. By default, each Service also gets an IP address, which is routable only inside the cluster. In our case, we have **`172.17.0.4`** and **`172.17.0.5`** IP addresses for our **`frontend-svc`** and **`db-svc`** Services, respectively. The IP address attached to each Service is also known as the ClusterIP for that Service.

kube-proxy

All of the worker nodes run a daemon called kube-proxy, which watches the API server on the master node for the addition and removal of Services and endpoints. For each new Service, on each node, **`kube-proxy`** configures the iptables rules to capture the traffic for its ClusterIP and forwards it to one of the endpoints. When the service is removed, **`kube-proxy`** removes the iptables rules on all nodes as well.



kube-proxy, Services, and Endpoints

Service Discovery

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

- **Environment Variables**
- As soon as the Pod starts on any worker node, the **`kubelet`** daemon running on that node adds a set of environment variables in the Pod for all active Services. For example, if we have an active Service called **`redis-master`**, which exposes port **`6379`**, and its ClusterIP is **`172.17.0.6`**, then, on a newly created Pod, we can see the following environment variables:
-
- **`REDIS_MASTER_SERVICE_HOST=172.17.0.6`**
- **`REDIS_MASTER_SERVICE_PORT=6379`**
- **`REDIS_MASTER_PORT=tcp://172.17.0.6:6379`**
- **`REDIS_MASTER_PORT_6379_TCP=tcp://172.17.0.6:6379`**
- **`REDIS_MASTER_PORT_6379_TCP_PROTO=tcp`**
- **`REDIS_MASTER_PORT_6379_TCP_PORT=6379`**
- **`REDIS_MASTER_PORT_6379_TCP_ADDR=172.17.0.6`**
-
- With this solution, we need to be careful while ordering our Services, as the Pods will not have the environment variables set for Services which are created after the Pods are created.
- **DNS**
- Kubernetes has an add-on for DNS, which creates a DNS record for each Service and its format is like **`my-svc.my-namespace.svc.cluster.local`**. Services within the same Namespace can reach to other Services with just their name. For example, if we add a Service **`redis-master`** in the **`my-ns`** Namespace, then all the Pods in the same Namespace can reach to the **`redis`** Service just by using its name, **`redis-master`**. Pods from other Namespaces can reach the Service by adding the respective Namespace as a suffix, like **`redis-master.my-ns`**.

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

ServiceType

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

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

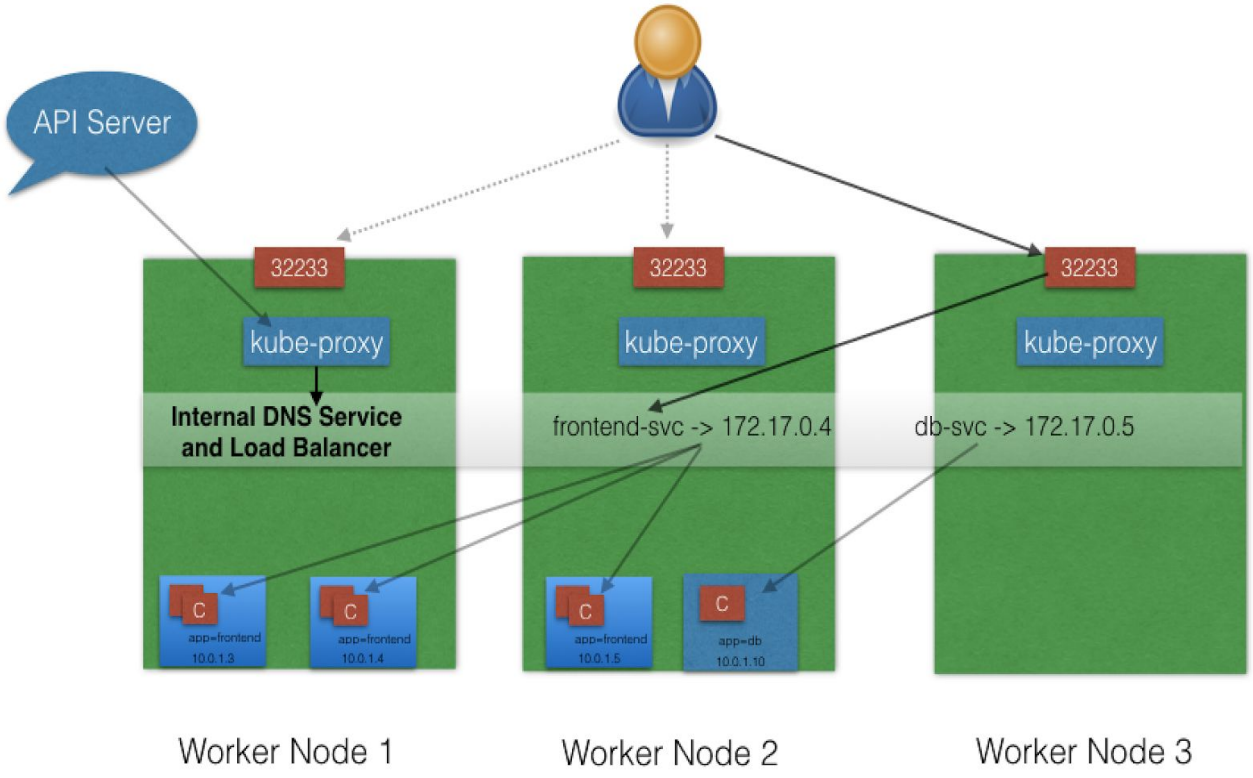
Access scope is decided by S*erviceType*, which can be mentioned when creating the Service.

ServiceType: ClusterIP and NodePort

ClusterIP is the default *ServiceType*. A Service gets its Virtual IP address using the ClusterIP. That IP address is used for communicating with the Service and is accessible only within the cluster.

With the **NodePort** *ServiceType*, in addition to creating a ClusterIP, a port from the 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.`**

By default, while exposing a NodePort, a random port is automatically selected by the Kubernetes Master from the port range **`30000-32767`**. If we don't want to assign a dynamic port value for NodePort, then, while creating the service, we can also give a port number from the earlier specific range.



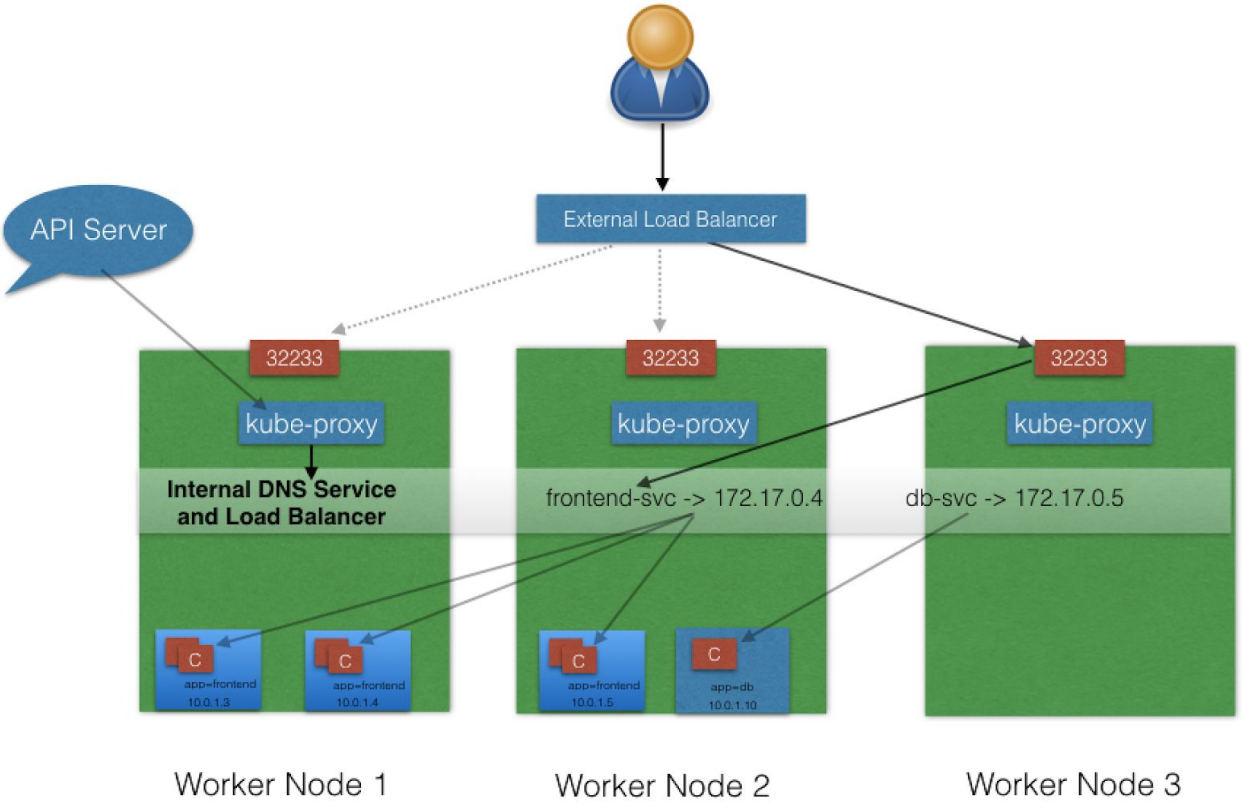
NodePort

The **`NodePort`** *ServiceType* is useful when we want to make our Services accessible from the external world. The end-user connects to the worker nodes on the specified port, which forwards the traffic to the applications running inside the cluster. To access the application from the external world, administrators can configure a reverse proxy outside the Kubernetes cluster and map the specific endpoint to the respective port on the worker nodes.

ServiceType: LoadBalancer

With the **LoadBalancer** *ServiceType*:

- NodePort and ClusterIP Services are automatically created, and the external load balancer will route to them
- The Services are exposed at a static port on each worker node
- The Service is exposed externally using the underlying cloud provider's load balancer feature.

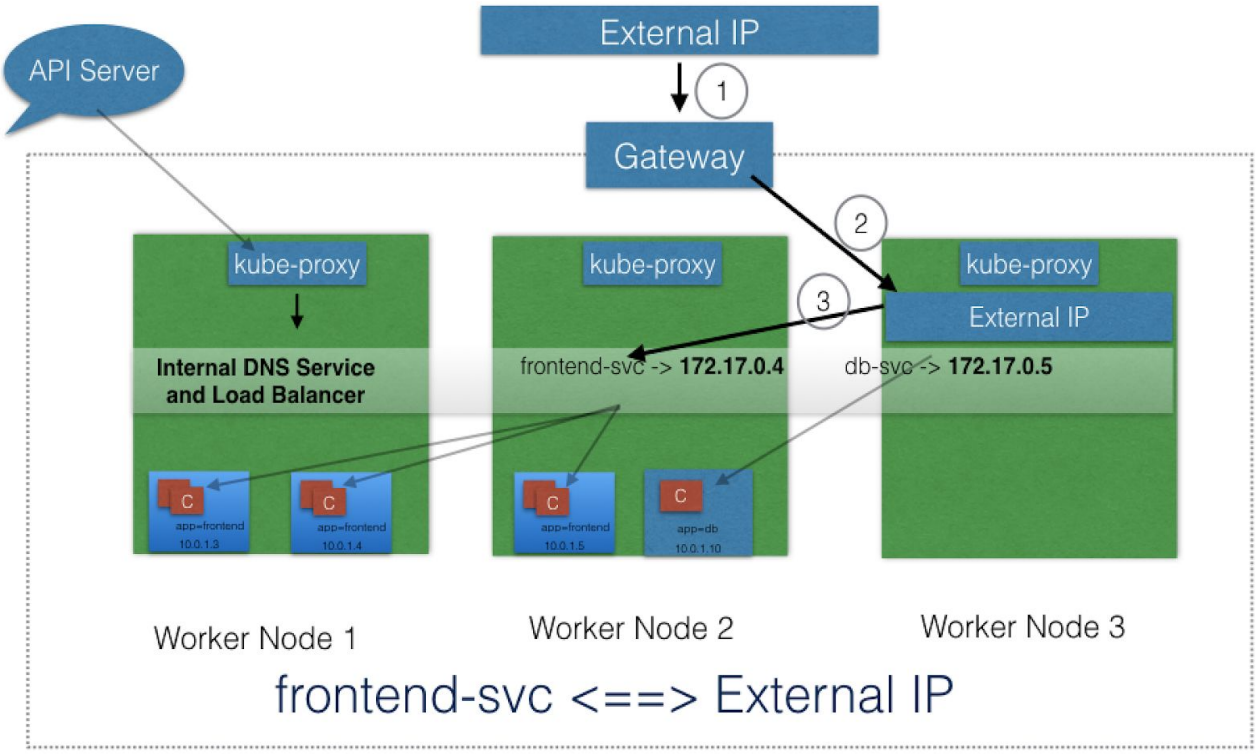


LoadBalancer

The LoadBalancer *ServiceType* will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes, as is the case with the Google Cloud Platform and AWS.

ServiceType: ExternalIP

A Service can be mapped to an **ExternalIP** address if it can route to one or more of the worker nodes. Traffic that is ingressed into the cluster with the ExternalIP (as destination IP) on the Service port, gets routed to one of the the Service endpoints.



ExternalIP

Please note that ExternalIPs are not managed by Kubernetes. The cluster administrators has configured the routing to map the ExternalIP address to one of the nodes.

ServiceType: ExternalName

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

The primary use case of this *ServiceType* is to make externally configured Services like `my-database.example.com` available inside the cluster, using just the name, like `my-database`, to other Services inside the same Namespace.

Deploy a standalone application :-

Introduction

In this chapter, we will learn how to deploy an application using **Graphical User Interface (GUI)** or **Command Line Interface (CLI)**. We will also expose an application with NodePort, and access it from the external world.

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

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

Liveness and Readiness Probes

While we are discussing application deployment, let's talk about **Liveness** and **Readiness Probes**. These probes are very important, because they allow the kubelet to control the health of the application running inside a Pod's container.

Liveness

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

Rather than doing it manually, we can use **Liveness Probe**. Liveness probe checks on an application's health, and, if for some reason, the health check fails, it restarts the affected container automatically.

Liveness Probes can be set by defining:

- Liveness command
- Liveness HTTP request

- TCP Liveness Probe.

Liveness Command

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

```
apiVersion: v1
kind: Pod
metadata:
  labels:
    test: liveness
  name: liveness-exec
spec:
  containers:
  - name: liveness
    image: k8s.gcr.io/busybox
    args:
    - /bin/sh
    - -c
    - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy;
sleep 600
    livenessProbe:
     exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 5
```

The existence of the **`/tmp/healthy`** file is configured to be checked every 5 seconds using the **`periodSeconds`** parameter. The **`initialDelaySeconds`** parameter requests the kubelet to wait for 3 seconds before doing the first probe. When running the command line argument to the container, we will first create the **`/tmp/healthy`** file, and

then we will remove it after 30 seconds. The deletion of the file would trigger a health failure, and our Pod would get restarted.

Liveness HTTP Request

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

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

TCP Liveness Probe

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

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

Readiness Probes

Sometimes, applications have to meet certain conditions before they can serve traffic. These conditions include ensuring that the depending service is ready, or acknowledging

that a large dataset needs to be loaded, etc. In such cases, we use **Readiness Probes** and wait for a certain condition to occur. Only then, the application can serve traffic.

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

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

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

Kubernetes Volume management :

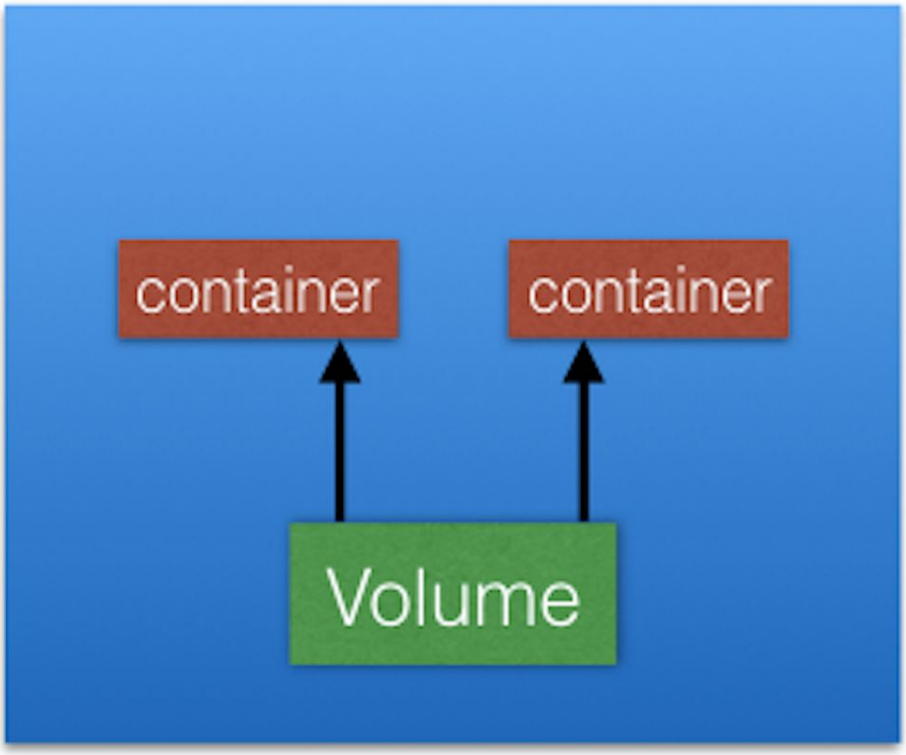
Introduction

To back a Pod with a persistent storage, Kubernetes uses **Volumes.** In this chapter, we will learn about Volumes and their types. We will also talk about **PersistentVolume** and **PersistentVolumeClaim** objects, which help us attach storage Volumes to Pods.

Volumes

As we know, containers, which create the 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 state, which means that it will not have any of the old data.

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



Volumes

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

Volume Types

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

- **`emptyDir`**
- An **`empty`** Volume is created for the Pod as soon as it is scheduled on the worker node. The Volume's life is tightly coupled with the Pod. If the Pod dies, the content of **`emptyDir`** is deleted forever.
- **`hostPath`**

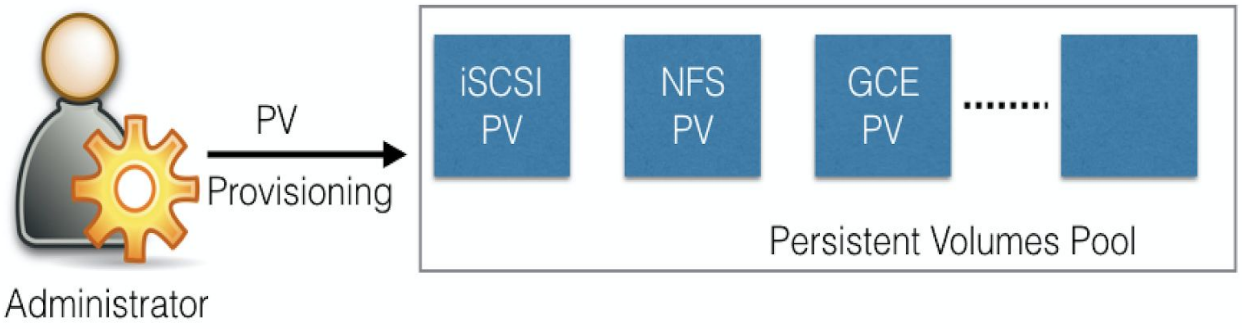
- With the **`hostPath`** Volume Type, we can share a directory from the host to the Pod. If the Pod dies, the content of the Volume is still available on the host.
- **`gcePersistentDisk`**
- With the **`gcePersistentDisk`** Volume Type, we can mount a Google Compute Engine (GCE) persistent disk into a Pod.
- **`awsElasticBlockStore`**
- With the **`awsElasticBlockStore`** Volume Type, we can mount an AWS EBS Volume into a Pod.
- **`nfs`**
- With nfs, we can mount an NFS share into a Pod.
- **`iscsi`**
- With iscsi, we can mount an iSCSI share into a Pod.
- **`secret`**
- With the **`secret`** Volume Type, we can pass sensitive information, such as passwords, to Pods. We will take a look at an example in a later chapter.
- **`persistentVolumeClaim`**
- We can attach a PersistentVolume to a Pod using a **`persistentVolumeClaim`**.

PersistentVolumes

In a typical IT environment, storage is managed by the storage/system administrators. The end user will just get instructions to use the storage, but does not have to worry about 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 storage. To manage the Volume, it uses the PersistentVolume API resource type, and to consume it, it uses the PersistentVolumeClaim API resource type.

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



PersistentVolume

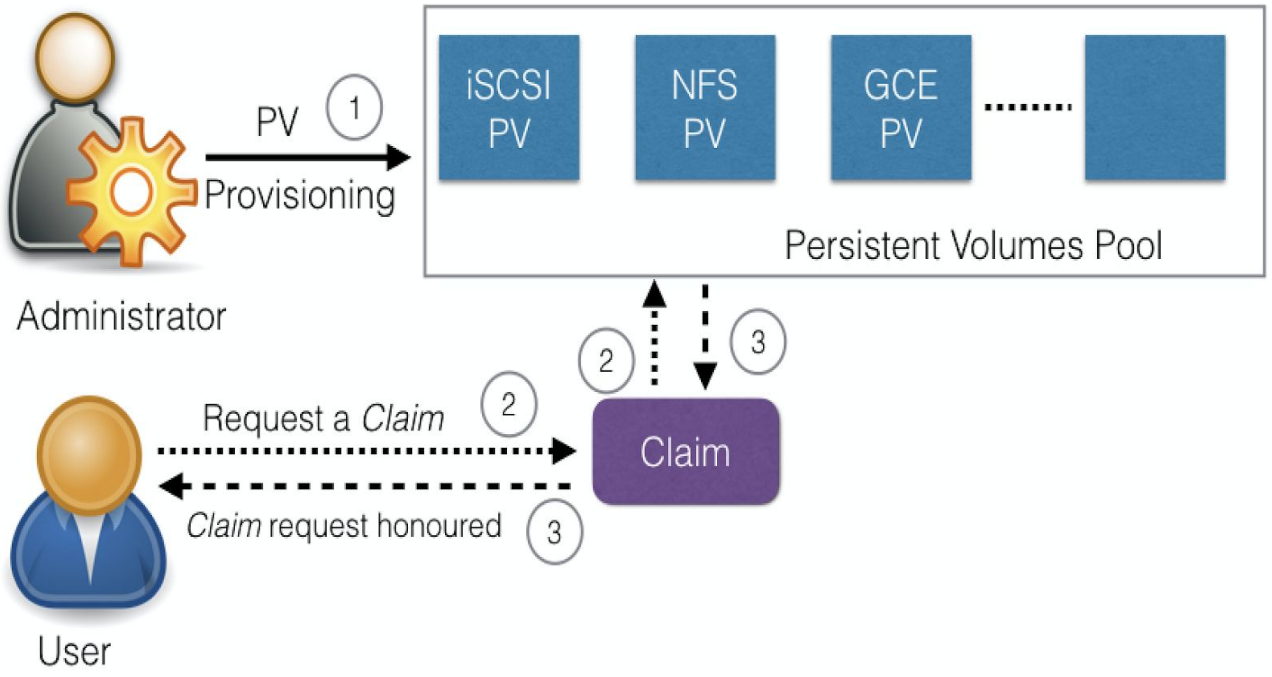
PersistentVolumes can be dynamically provisioned based on the StorageClass resource. A StorageClass contains pre-defined provisioners and parameters to create a PersistentVolume. Using PersistentVolumeClaims, a user sends the request for dynamic PV creation, which gets wired to the StorageClass resource.

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

- GCEPersistentDisk
- AWSElasticBlockStore
- AzureFile
- NFS
- iSCSI.

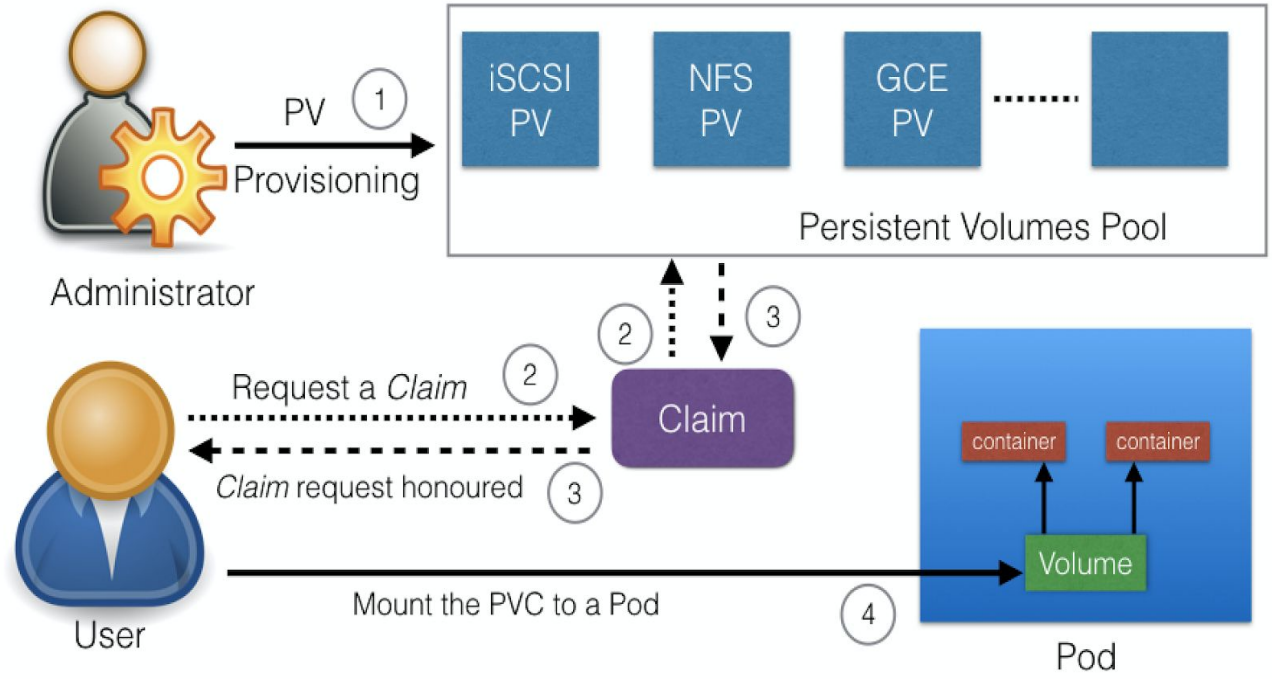
PersistentVolumeClaims

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. Users request for PersistentVolume resources based on size, access modes, etc. Once a suitable PersistentVolume is found, it is bound to a PersistentVolumeClaim.



PersistentVolumeClaim

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



PersistentVolumeClaim Used In a Pod

Once a user finishes its work, the attached PersistentVolumes can be released. The underlying PersistentVolumes can then be reclaimed and recycled for future usage.

Container Storage Interface (CSI)

At the time this course was created, container orchestrators like Kubernetes, Mesos, Docker or Cloud Foundry had their own way of managing external storage using Volumes.

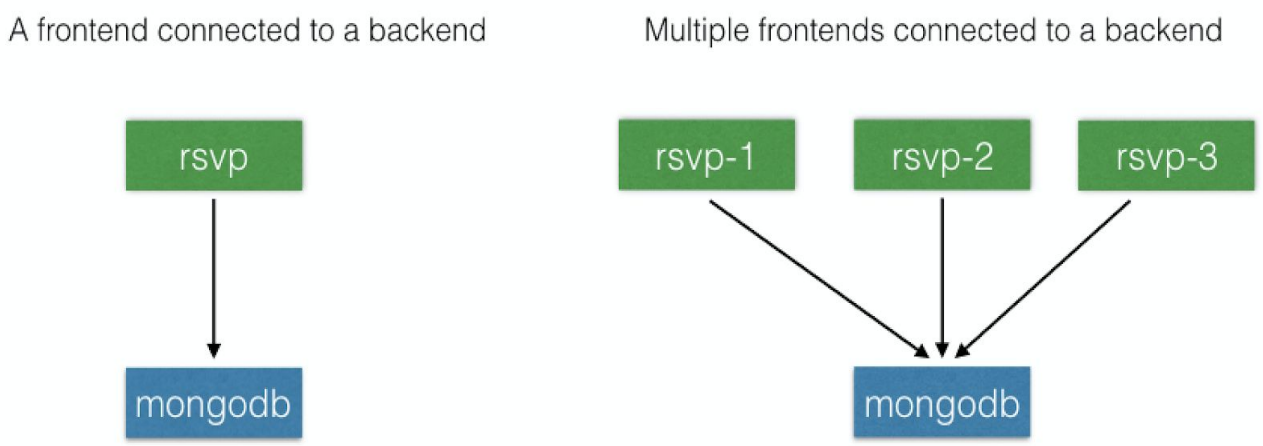
For storage vendors, it is difficult to manage different Volume plugins for different orchestrators. Storage vendors and community members from different orchestrators are working together to standardize the Volume interface; a volume plugin built using a standardized CSI would work on different container orchestrators. You can find CSI specifications here.

Kubernetes 1.9 added alpha support for CSI, which makes installing new CSI-compliant Volume plugins very easy. With CSI, third-party storage providers can develop solutions without the need to add them into the core Kubernetes codebase.

Multi tier application hosting :-

RSVP Application

We will be using a sample **RSVP application**. Using this application, users can register for an event by providing their username and email ID. Once a user registers, his/her name and email appears in a table. The application consists of a backend database and a frontend. For the backend, we will be using a MongoDB database, and for the frontend, we have a Python Flask-based application.



RSVP Application

The application's code is available here. In the frontend code (**`rsvp.py`**), we will look for the **`MONGODB_HOST`** environment variable for the database endpoint, and, if it is set, we will connect to it on port **`27017`**.

```
MONGODB_HOST=os.environ.get('MONGODB_HOST', 'localhost')
client = MongoCLient(MONGODB_HOST, 27017)
```

After deploying the application with one backend and one frontend, we will scale the frontend to explore the scaling feature of Kubernetes.

Next, we will deploy the MongoDB database - for this, we will need to create a Deployment and a Service for MongoDB.

Create the Deployment for MongoDB

Create an **`rsvp-db.yaml`** file with the following content:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rsvp-db
  labels:
    appdb: rsvpdb
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      appdb: rsvpdb
  template:
    metadata:
      labels:
        appdb: rsvpdb
    spec:
      containers:
      - name: rsvp-db
        image: mongo:3.3
        ports:
        - containerPort: 27017
```

and run the following command to create the **rsvp-db** Deployment:

```
$ kubectl create -f rsvp-db.yaml
deployment "rsvp-db" created
```

Create the Service for MongoDB

To create a **mongodb** Service for the backend, create an **rsvp-db-service.yaml** file with the following content:

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb
  labels:
    app: rsvpdb
```

```
spec:
  ports:
  - port: 27017
    protocol: TCP
  selector:
    appdb: rsvpdb
```

and run the following command to create a Service named **mongodb** to access the backend:

```
$ kubectl create -f rsvp-db-service.yaml
service "mongodb" created
```

As we did not specify any *ServiceType*, **mongodb** will have the default ClusterIP *ServiceType*. This means that the **mongodb** Service will not be accessible from the external world.

ConfigMap and Secrets :-

Introduction

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

ConfigMaps

ConfigMaps allow us to decouple the configuration details from the container image. Using ConfigMaps, we can pass configuration details as key-value pairs, which can be later consumed by Pods, or any other system components, such as controllers. We can create ConfigMaps in two ways:

- From literal values
- From files.

Create a ConfigMap from Literal Values and Get Its Details

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

Create the ConfigMap

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

Get the ConfigMap Details for `my-config`

```
$ kubectl get configmaps my-config -o yaml
apiVersion: v1
data:
  key1: value1
  key2: value2
kind: ConfigMap
metadata:
  creationTimestamp: 2017-05-31T07:21:55Z
  name: my-config
  namespace: default
  resourceVersion: "241345"
```

```
  selfLink:
/api/v1/namespaces/default/configmaps/my-config
  uid: d35f0a3d-45d1-11e7-9e62-080027a46057
```

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

Use ConfigMap Inside Pods

As an Environment Variable

We can get the values of the given key as environment variables inside a Pod. In the following example, while creating the Deployment, we are assigning values for environment variables from the `customer1` ConfigMap:

```
....
 containers:
      - name: rsvp-app
        image: teamcloudyuga/rsvpapp
        env:
        - name: MONGODB_HOST
          value: mongodb
        - name: TEXT1
          valueFrom:
            configMapKeyRef:
              name: customer1
              key: TEXT1
        - name: TEXT2
          valueFrom:
```

```
            configMapKeyRef:
              name: customer1
              key: TEXT2
        - name: COMPANY
          valueFrom:
            configMapKeyRef:
              name: customer1
              key: COMPANY
....
```

With the above, we will get the **`TEXT1`** environment variable set to **`Customer1_Company`**, **`TEXT2`** environment variable set to **`Welcomes You`**, and so on.

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 put down 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 situations such as the one we just mentioned, the Secret object can help. 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 system components, the Secret object is *referenced*, without exposing its content.

It is important to keep in mind that the Secret data is stored as plain text inside **etcd**. Administrators must limit the access to the API server and **etcd**.

Create the Secret with the 'kubectl create secret' Command

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

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

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

'get' and 'describe' the Secret

Analyzing the **`get`** and **`describe`** examples below, we can see that they do not reveal the content of the Secret. The type is listed as **`Opaque`**.

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

Create a Secret Manually

We can also create a Secret manually, using the YAML configuration file. With Secrets, each object data must be encoded using **`base64`**. If we want to have a configuration file for our Secret, we must first get the **`base64`** encoding for our password:

```
$ echo mysqlpassword | base64
```

```
bXlzcWxwYXNzd29yZAo=
```

and then use it in the configuration file:

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

Please note that **base64** encoding does not do any encryption, and anyone can easily decode it:

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

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

Use Secrets Inside Pods

We can get Secrets to be used by containers in a Pod by mounting them as data volumes, or by exposing them as environment variables.

Using Secrets as Environment Variables

As shown in the following example, we can reference a Secret and assign the value of its key as an environment variable (**WORDPRESS_DB_PASSWORD**):

```
.....
    spec:
      containers:
      - image: wordpress:4.7.3-apache
        name: wordpress
```

```
        env:
        - name: WORDPRESS_DB_HOST
          value: wordpress-mysql
        - name: WORDPRESS_DB_PASSWORD
          valueFrom:
            secretKeyRef:
              name: my-password
              key: password
.....
```

Using Secrets as Files from a Pod

We can also mount a Secret as a Volume inside a Pod. A file would be created for each key mentioned in the Secret, whose content would be the respective value. For more details

Ingress:-

Introduction

In *Chapter 9. Services*, we saw how we can access our deployed containerized application from the external world. Among the *ServiceTypes* mentioned in that chapter, NodePort and LoadBalancer are the most often used. For the LoadBalancer *ServiceType*, we need to have the support from the underlying infrastructure. Even after having the support, we may not want to use it for every Service, as LoadBalancer resources are limited and they can increase costs significantly. Managing the NodePort *ServiceType* can also be tricky at times, as we need to keep updating our proxy settings and keep track of the assigned ports. In this chapter, we will explore the **Ingress**, which is another method we can use to access our applications from the external world.

Ingress I

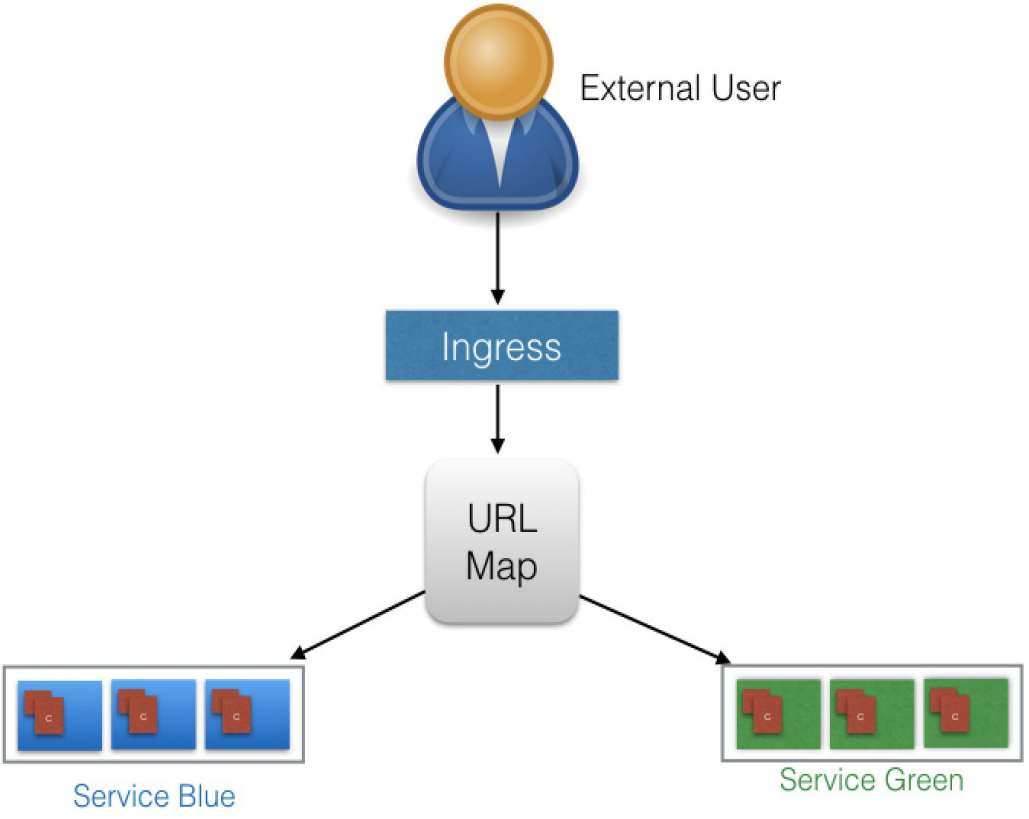
With Services, routing rules are attached to a given Service. They exist for as long as the Service exists. If we can somehow decouple the routing rules from the application, we can then update our application without worrying about its external access. This can be done using the Ingress resource.

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 load balancer for Services and provides the following:

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



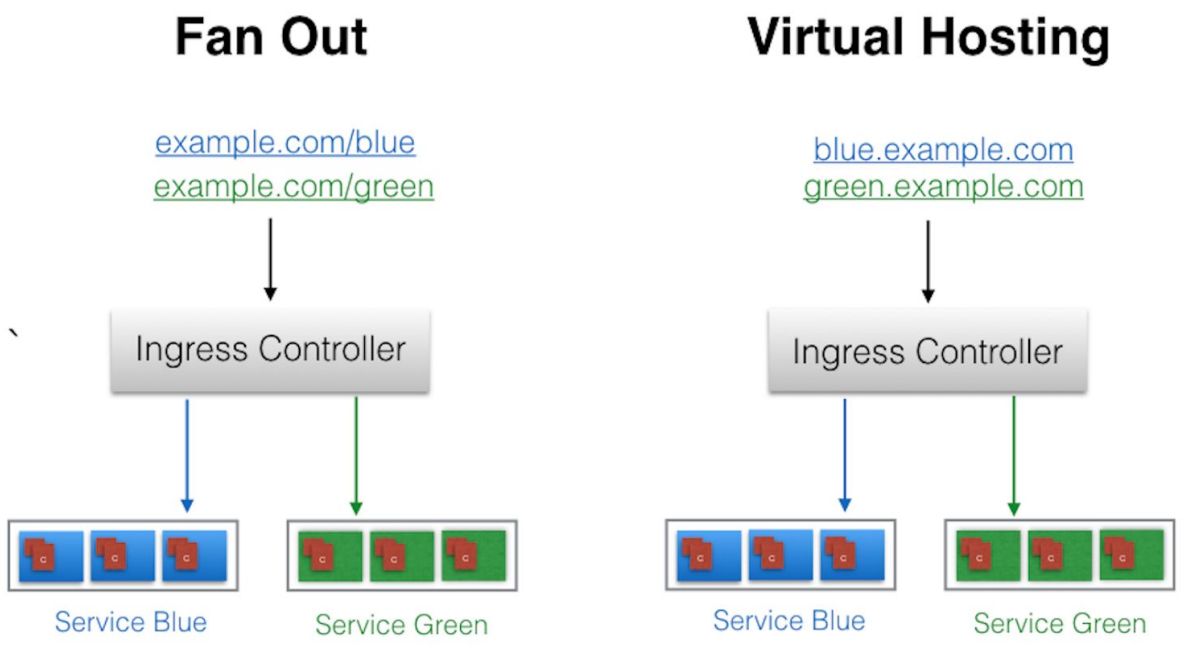
Ingress II

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

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: web-ingress
  namespace: default
spec:
  rules:
  - host: blue.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-blue-svc
          servicePort: 80
  - host: green.example.com
    http:
      paths:
      - backend:
          serviceName: webserver-green-svc
          servicePort: 80
```

According to the example we provided above, users 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. Here, we have seen an example of a Name-Based Virtual Hosting Ingress rule.

We can also have Fan Out Ingress rules, in which we send requests like **`example.com/blue`** and **`example.com/green`**, which would be forwarded to **`webserver-blue-svc`** and **`webserver-green-svc`**, respectively.



Ingress URL Mapping

Ingress Controller

An Ingress Controller is an application which watches the Master Node's API server for changes in the Ingress resources and updates the Layer 7 Load Balancer accordingly. Kubernetes has different Ingress Controllers, and, if needed, we can also build our own. GCE L7 Load Balancer and Nginx Ingress Controller are examples of Ingress Controllers.

Start the Ingress Controller with Minikube

Minikube v0.14.0 and above ships the Nginx Ingress Controller setup as an add-on. It can be easily enabled by running the following command:

```
$ minikube addons enable ingress
```

Deploy an Ingress Resource

Once the Ingress Controller is deployed, we can create an Ingress resource using the `kubectl create` command. For example, if we create a `webserver-ingress.yaml`

file with the content that we saw on the *Ingress II* page, then, we will use the following command to create an Ingress resource:

```
$ kubectl create -f webserver-ingress.yaml
```

Access Services Using Ingress

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

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

Once this is done, we can open **`blue.example.com`** and **`green.example.com`** on the browser and access the application.