Container Image

It confine (Limit/Group) the application code, its runtime, and all of its dependencies in a pre-defined format.

Container Runtime

With container runtimes like **Docker**, **runC**, **containerd**, or **rkt** we can use those pre-packaged images, to create one or more containers. We use that image to create an isolated executable environment, also known as container.

Container Orchestrator

All of these runtimes (docker, runC, rkt, etc.) 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.** This controller/management unitis generally refer to as a **container orchestrator.**

**Container orchestrators** are the tools which group hosts together to form a cluster, and help us fulfil the below requirements.

* Fault Tolerant
* On Demand Scale
* Accessible from Outside world
* Update/Rollback without downtime

Number of container orchestrators available are,

1. Docker Swarm
2. Kubernetes
3. Mesos Marathon
4. Amazon ECS
5. Hashicorp Nomad
6. Redhat Openshift
7. Azure Kubernetes Service (AKS)

**Kubernetes Features**

Some of its fully supported features are,

* Automatic Binpacking (Scheduling)

Kubernetes automatically schedules the containers based on resource usage and constraints.

* Self-healing

Kubernetes automatically replaces and reschedules the containers from failed nodes. It also kills and restarts the containers that 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 can discover the services (Kubernetes Service Resource) automatically, and load balance requests between containers of given Service.

* Automated Rollouts and Rollbacks

Kubernetes can roll out and roll back new versions/configurations of an Application, without introducing any downtime.

* Secret and Configuration Management

Kubernetes can manage secret/configuration details for an application using Secret resource.

* Storage Orchestration

With Kubernetes and its plugins, we can automatically mount local, external, and storage solutions to the containers in a seamless manner

* Batch Execution

Besides long running jobs, Kubernetes also supports batch execution.

**Kubernetes Architecture**

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

1. **One or more Master Node**

The master node is responsible for managing Kubernetes cluster, and it is the entry point for all administrative tasks. For fault-tolerance purposes, there can be more than One Master node in the Cluster.

If we have more than one Master node, it would be in a HA (High Availability) mode, and only one of them will be Leader performing all the operations. The rest of the Master nodes would be Follower.

A Master node has the following components:

* + **API Server**

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 ETCD.

* + **Scheduler**

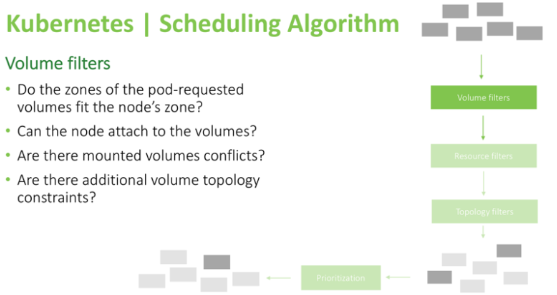
The Scheduler schedules the work to different worker nodes. The Scheduler has resource usage information for each worker node. Before scheduling work, the Scheduler also takes into account the Quality of Service requirements, data locality, affinity, anti-affinity, user/operator constraints etc.

The scheduler’s decision to start a new pod goes through these three stages:

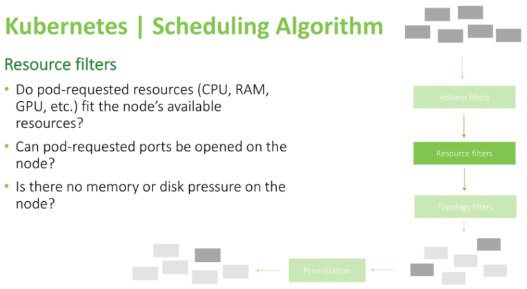
* + - Node filtering

The Scheduler will check which nodes are compatible with running this workload. It does so by running all nodes through a set of filters and removing those, which are not compatible from consideration. The following Filter are used,

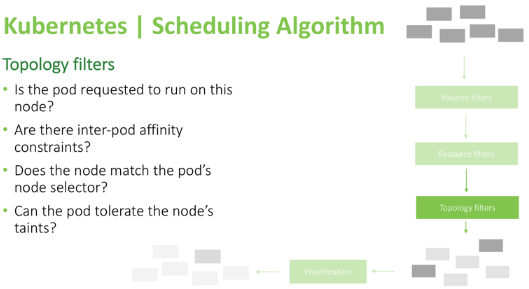
1) Volume filters



2) Resource filters (CPU, Memory, Port)



3) Affinity Selector (Node Affinity, Inter Pod Affinity, and Anti Affinity)



* + - Node priority calculation

After node filtering, the scheduler calculates a score for each node, and the highest scoring node will run that pod.

* + - Actual scheduling operation
  + **Controller Manager**

The Controller Manager manages different non-terminating Control Loops. This control loops 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 API server.

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.

* + **etcd**

**etcd** is a distributed key-value store which is used to store the cluster state. It can be part of the Kubernetes Master, or, it can be configure externally in which case master nodes would connect to it.

1. **One or more Worker Node**

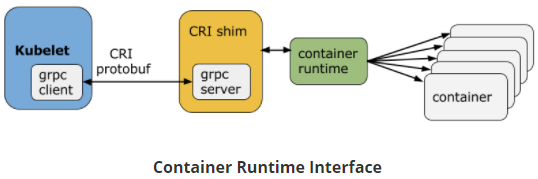
A **worker node** is a machine (VM, physical server, etc.) which runs the applications as Pods and is control by the master node. Pods is logical collection of one or more containers that are always schedule together.

A Worker node has the following components:

* + **Kubelet**

The Kubelet is an agent that runs on each Worker node and communicates with the Master node. Kubelet receives the Pod definition via various means (primarily via API server), and runs the container associated with the Pod.

It also make sure that the containers, which are part of Pods are Healthy at all times. The Kubelet connects to Container Runtime using Container Runtime Interface (CRI).



As shown above, the kubelet (grpc client) connects to the CRI shim (grpc server) to perform container and image operations. CRI implements 2 services,

* + - ImageService

The **ImageService** is responsible for all the image-related operations.

* + - RuntimeService

The **RuntimeService** is responsible for all the Pod and container-related operations.

* + **Kube-Proxy**

Instead of connecting directly to Pods to access the Applications, we use logical construct (resource) called a **Service.** A Service groups related Pods, when accessed, load balances to them.

Kube-Proxy is the Network proxy that 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 be reachable.

* + Container Runtime

To run and manager a container’s lifecycle, we need a Container Runtime on the worker node. Some of the known container runtime are,

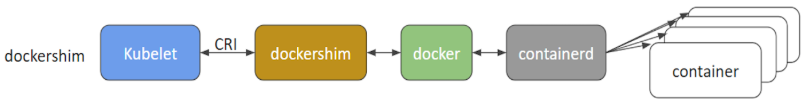
* + - Docker
    - Containerd
    - Rkt
    - Lxd

1. **Distributed Key-Value store like ETCD.**

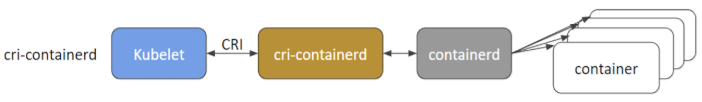
In Kubernetes, besides storing the cluster state, etcd is also use to store configuration details such as subnets, ConfigMaps, Secrets, etc. At any given time, one of the nodes in the group will be the master, and the rest of them will be the followers.

**Container Runtime Interface (CRI) Shims**

* Dockershim



* Containershim



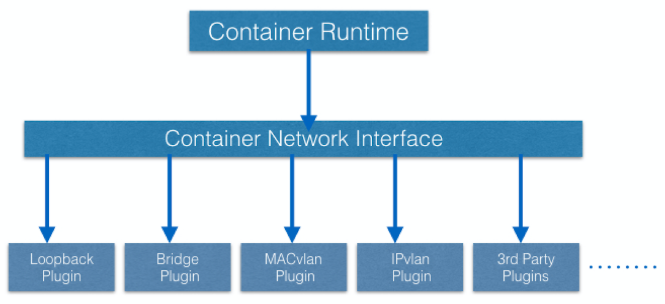
**Container Networking**

In Kubernetes, each Pod gets a unique IP address. For Container Networking, there are

Two (2) specifications,

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

Kubernetes used CNI to assign IP address to each Pod.



The Container Runtime (Docker, rkt, etc) offloads the IP assignment to CNI, which connect to underlying configured Plugins, like Bridge, MACvlan, IPvlan, 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 POD**

All the Container Runtime generally creates 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 OS.

Inside a POD, containers share the Network Namespaces so they can reach to each other via localhost.

**Kubernetes Installation**

The four major installation types are briefly presented below:

* All in One Single Node Installation
* Single Node ETCD, Single Master Node, Multi Worker Node
* Single Node ETCD, Multi Master Node, Multi Worker Node
* Multi Node ETCD, Multi Master Node, Multi Worker Node

Some of the known Installers are,

* Minikube
* Kubeadm
* KubeSpray

With [KubeSpray](https://github.com/kubernetes-incubator/kubespray), 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.

* Kops

With [Kops](https://github.com/kubernetes/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.

**Accessing Kubernetes Cluster**

Kubernetes Cluster can be accessed in 3 different ways,

* Command Line Interface (Kubectl)
* Graphical User Interface (Kubernetes Dashboard)
* API

[Kubectl](https://kubernetes.io/docs/user-guide/kubectl/) is the **Command Line Interface (CLI)** tool to manage the Kubernetes cluster resources and applications.

**Application Programming Interface (API)** of kubernetes can be divided into 3 independent groups:

1. Core API (**/api/v1**)

This group includes objects such as Pods, Services, Nodes, etc.

1. Named Group API

This Group includes objects in **/apis/$NAME/$VERSION** format. These also contains objects of different stability level,

* + Alpha Level - Objects here may be drop anytime without notice.
  + Beta Level - It is well tested, may change in subsequent beta or stable release.
  + Stable Level - It appears in next released software.

1. System Wide API

This group consist of system wide API endpoints like /apis, /healthz, /metrics, /logs, /ui, /version, etc.

Use below 2 methods to access the APIs using curl command.

1. Get the Token to access the Secured API server running on HTTPS.

$ TOKEN=$(kubectl describe secret |awk '/^token/ {print $2}')

$ curl https://localhost:6443 --header "Authorization: Bearer $TOKEN" -k

1. Use Kubectl Proxy command.

$ kubectl proxy &

$ curl http://localhost:8001

**Kubectl Configuration File**

To connect to the Kubernetes cluster, kubectl needs the master node endpoint and the credentials to connect to it. By default, a configuration file **config**, inside the **.kube** directory, which resides in the user's home directory. i.e **~/.kube/config (Default Location).**

You can set your own path to read Configuration file with “**KUBECONFIG**” environment variable.

$ export KUBECONFIG=~/.kube/config:~/.kube/kubconfig2

Configuration file can be view with below command as well.

$ kubectl config view

**Kubernetes Building Blocks (Objects)**

With each object, we declare our intent or desired state using the **spec** field. The Kubernetes system manages the **status** field for objects, in which 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.

The API request to create the object must have the **spec** field, as well as other details, in a JSON format.

* With the **apiVersion**field, we mention the API endpoint on the API server, which we want to connect.
* With the **kind**field, we mention the object type like Pod, Service, Namespace, Deployment, etc.
* With the **metadata**field, we attach the basic information to objects, like the name, labels.
* With **spec**, we define the desired state of the Object.
* The **status** field is add by the Kubernetes system.

1. Pods

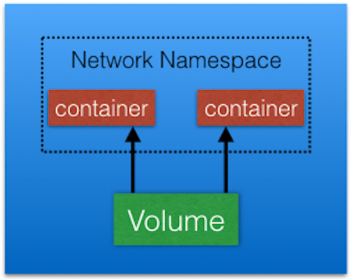
A Pod is logical collection of one or more containers, which:

- Are schedule together on the same host.

- Share the same network namespace.

- Mount the same external storage (Volumes)

**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 these controllers are Deployments, ReplicaSets, ReplicationControllers, etc.



1. Labels

Labels are Key-Value pairs that are attached to any Kubernetes Objects e.g Pods, Deployment, Service, Replication Controller etc. Labels do not provide uniqueness to objects one or more objects can have same labels.

1. Label Selectors

Kubernetes support 2 types of Selectors:

1. Equality Based Selectors

It allows filtering of Objects based on Label Keys and Values. We can use the **=**, **==**, or **!=** operators. E.g with **env==dev** we are selecting objects where **env** Label is set to **dev**.

1. Set Based Selectors

It allow filtering of Objects based on set of Values. We can use the **in**, **notin**, or **exist** operators. E.g, with **env** **in (dev,qa)** we are selecting objects where **env** Label is set to **dev** or **qa.**

1. ReplicationControllers

A ReplicationController (rc) is a controller that is **part of master nodes Controller Manager**. It make sure the specified number of replicas for Pod is running at any given point in time. If the Actual Pod are higher than desired, **rc** will kill the extra Pod. In case, Actual Pod are lesser than desired, **rc** will create more Pods to match the desired count.

ReplicationController only supports Equality Based Selectors.

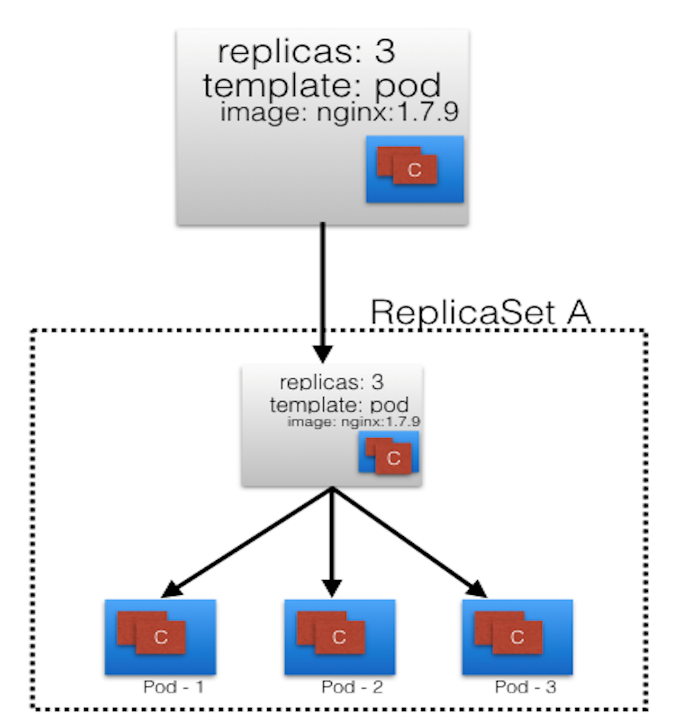
1. ReplicaSet

A ReplicaSet is the next generation ReplicationController. ReplicaSet supports both Equality Based, and Set Based Label selectors. It also make sure desired Pods Count are matching with the actual Pods count.

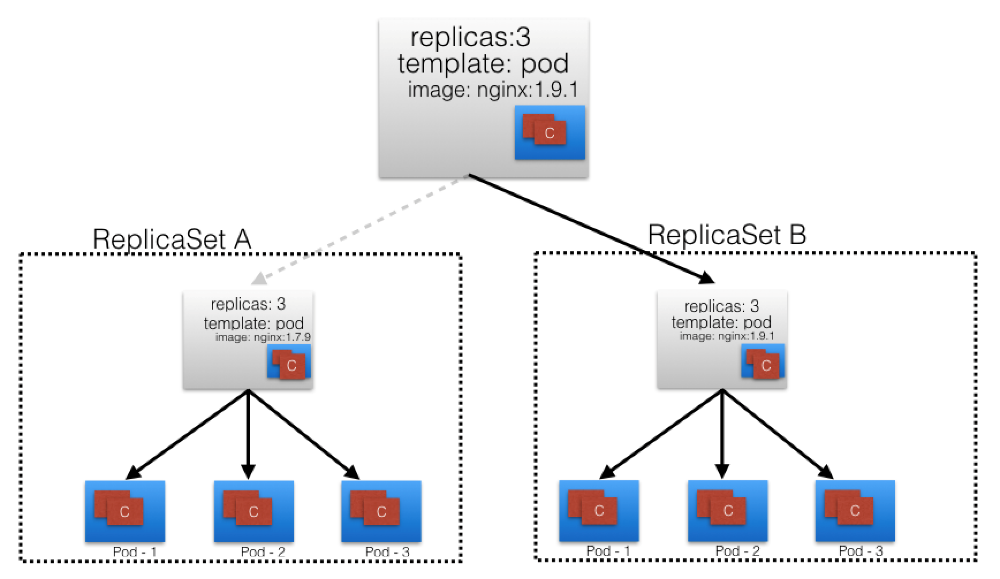
ReplicaSet can be use independently, but they are mostly use by Deployment to to orchestrate the Pod creation, deletion, or updates.

1. Deployments

The Deployment Controller is part of master nodes Controller Manager, and it also make sure the current state always match the desired state.



**Deployment Rollout** take place when you modify the Pod template of the deployment. Say suppose you have Deployment which runs Nginx container of 1.7.9 version, now if you plan to run new version sat 1.9.1, the deployment will kick in the Rollout that will create new ReplicaSet and new Pods with new version.



Deployment also provide us the flexibility to **Rollback** to previously known state.

1. Namespaces

Namespaces are way to divide the Cluster Resources between multiple Users/Teams/Projects. The name of the Object/Resources created inside Namespace are unique.

Generally, Kubernetes creates **default**, **kube-system**, and **kube-public** namespaces.

* + The **kube-system** Namespace contains objects created by the Kubernetes System.
  + The **default** Namespace contains objects that are not part of any other Namespace.
  + The **kube-public** Namespace is readable by all users (including those not authenticated).

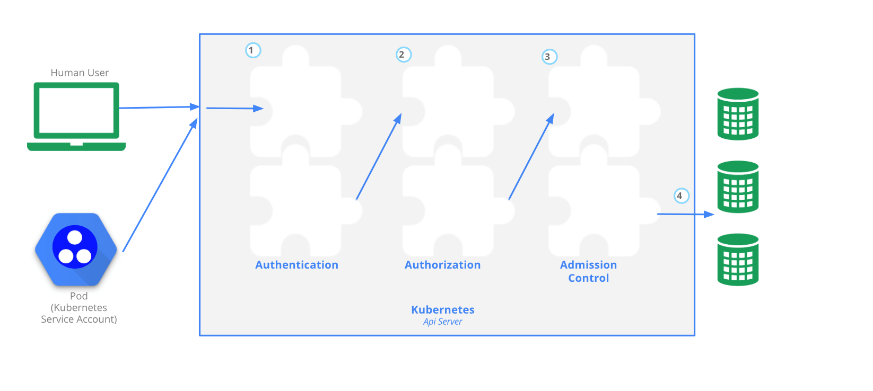
Using **ResourceQuota**, we can restrict the resource utilization within Namespace.

**Authentication, Authorization, and Admission Control**

To access or manage the Objects/Resources in Kubernetes cluster, we need to have access to the API endpoint on the API server. Each access request goes through the following stages,

1. Authentication
2. Authorization
3. Admission Control

Software modules that can modify or reject the requests based on some additional checks. E.g Quota



**Authentication:** Kubernetes does not have object called user, nor it stores usernames or other details to its object store (etcd). Kubernetes has two kinds of users:

* Normal Users

They are manage outside Kubernetes cluster via independent services like User/Client Certificates, a file listing Username/Passwords, etc.

* Service Accounts

The Service Account users are managed by Kubernetes. Most of the Service Account users are created automatically via the API server, but they can be created manually.

The Service Account users are bind to the Namespace. It also has the credentials required to communicate with the API server as Secrets (Token).

For Authentication, Kubernetes uses different modules: We can enable Multiple Authenticators.

* Client Certificates

To enable Client Certificate authentication, we need to reference a file containing one or more Certificate Authorities (CA) by passing the “—client-ca-file” option in API server config.

* Static Token File

We can pass a file containing pre-defined Bearers Tokens with the “—token-auth-file” options in API server config. These Tokens would last indefinitely, and cannot be changed without restarting API server.

* Bootstrap Token

It is mostly use for bootstrapping a new Kubernetes Cluster. This Feature is in Alpha status.

* Static Password File

It is similar to Static Token File. It can be enable by passing “—basic-auth-file” options in API server config.

* Service Account Tokens

This module is automatically enable that uses signed Bearer Tokens to verify the requests. These tokens are attach to Pods using **ServiceAccount Admission Controller**.

* OpenID Connect Tokens
* Webhook Token Authentications

With Webhook-based authentication, verification of bearer tokens can be offloaded to a remote service.

* Keystone Password
* Authenticating Proxy

**Authorization:** After a successful authentication, those API requests get authorized by Kubernetes using various Authorization modules. More than one module can be configure for one Kubernetes cluster, and each module is check in sequence.

If any authorizer approves or denies a request, then that decision is return immediately.

Below are various Authorization Modules,

* Node Authorizer

Node Authorization is special purpose authorization mode, which specifically authorizes API requests from the Kubelet.

* Attribute Based Access Control Authorizer
* Role Based Access Control Authorizer

With RBAC we can regulate the access to resources based on the roles of individual users. In Kubernetes, we can have different roles (object) that can be attach to (subjects) like users, groups, and service accounts, etc.

While creating the roles, we restrict resource access by specific operations, such as **create**, **get**, **update**, **patch**, etc. These operations are refer to as **verbs**.

In RBAC, we can create 2 types of roles:

1. Role

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

kind: Role

apiVersion: rbac.authorization.k8s.io/v1

metadata:

namespace: lfs158

name: pod-reader

rules:

- apiGroups: [""] **# "" indicates the core API group**

resources: ["pods"] **# Object in above API Group**

verbs: ["get", "watch", "list"] **# Operation**

1. ClusterRole

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

Once the role is created, we can bind users with RoleBinding. There are 2 kinds of RoleBinding.

1. RoleBinding

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

kind: RoleBinding

apiVersion: rbac.authorization.k8s.io/v1

metadata:

name: pod-read-access

namespace: lfs158

subjects:

- kind: User

name: bhavesh

apiGroup: rbac.authorization.k8s.io

**roleRef:**

**kind: Role**

**name: pod-reader**

**apiGroup: rbac.authorization.k8s.io**

1. ClusterRoleBinding

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

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.

* 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.

**Admission Control** is use to specify granular access control policies, which include allowing privileged containers, checking on resource quota, etc. Admission control come into effect only after API requests are Authenticated and Authorized.

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

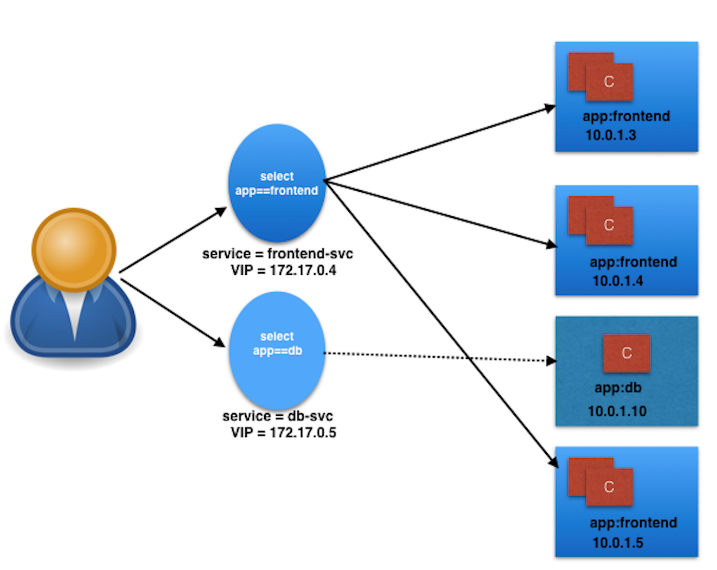
**Services**

To access the Container Application, user/client needs to connect to the Pods. As Pods are ephemeral in nature, resources like IP addresses cannot be static. Pods die abruptly or rescheduled to other Node.

To overcome this situation, Kubernetes provides a higher-level abstraction called Service.

Service logically groups Pods via **Labels** and **Selectors**. By default, each Service gets an IP address that is routable only inside the Cluster.

If the target port is not define explicitly, then traffic will be forward to Pods on the port on which the Service receives traffic.



**Service Discovery**

As Service are the Primary mode of communication between the Container Application in Kubernetes.

Kubernetes supports two methods of discovering Service.

1. 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 Service**.

For example, if we have Service called **http-svc**, which exposes port 8090, and its ClusterIP is 172.17.0.6, then if we create new POD say nginx, these new pod will have below environment variables:

HTTP\_SVC\_SERVICE\_HOST=172.17.0.6

HTTP\_SVC\_SERVICE\_PORT=8090

HTTP\_SVC\_PORT=tcp://172.17.0.6:8090

HTTP\_SVC\_PORT\_6379\_TCP=tcp://172.17.0.6:8090

HTTP\_SVC\_PORT\_6379\_TCP\_PROTO=tcp

HTTP\_SVC\_PORT\_6379\_TCP\_PORT=8090

HTTP\_SVC\_PORT\_6379\_TCP\_ADDR=172.17.0.6

Note: Please note Services created after Pod creation will not be inserted as Environment variables.

1. DNS

Kubernetes has an Add-On for DNS (kube-dns, core-dns), which creates DNS record for each new Service created within Kubernetes Cluster.

For example, if we add a Service **redis-master** in **my-ns** Namespace, then all the Pods in the same Namespace can reach to the 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**.

**ServiceType**

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

* Is only accessible within the Cluster (ClusterIP)
* Is accessible from within the Cluster and the External world (NodePort, LoadBalancer)
* Maps to an external entity which resides outside the Cluster ()

1. ClusterIP

It is the default ServiceType. That IP address is use for communicating with the Service and is accessible only within the cluster.

1. NodePort

With the NodePort ServiceType, in addition to creating a ClusterIP, a port from the range 30000-32767 is map to the respective Service.

By default, while exposing a NodePort, a random port is automatically selected by the Kubernetes Master from the port range **30000-32767**. If we do not 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.

For example, if the mapped NodePort is 32233 for the service http-svc, then if we connect to any worked node on port 32233, the node would redirect all the traffic to the assigned ClusterIP – 172.17.0.6

The **NodePort***ServiceType* is useful when we want to make our Services accessible from the external world.

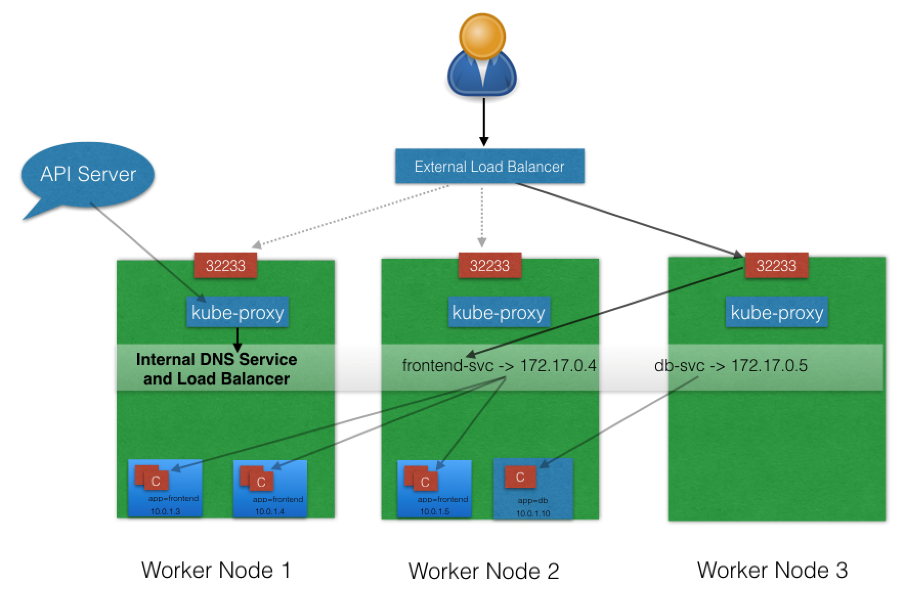
1. LoadBalancer

With the LoadBalancer ServiceType:

- NodePort and ClusterIP Services are create automatically, and the external LoadBalancer will route to them.

- The Service is expose at a static port on each worker node.

- The Service is expose externally using the underlying Cloud Providers Load Balancer.



The LoadBalancer *ServiceType* will only work if the underlying infrastructure supports the automatic creation of Load Balancers and have the respective support in Kubernetes.

1. ExternalName

ExternalName is special ServiceType, which has no Selectors and does not define any Endpoints.

When these Service is access within Cluster, it returns a CNAME record of an externally configured service. For example,

kind: Service

apiVersion: v1

metadata:

name: my-service

namespace: prod

spec:

type: ExternalName

externalName: my.database.example.com

When looking up the host my-service.prod.svc.cluster.local, the cluster DNS service will return a CNAME record with the value my.database.example.com.

**Probes**

1. **Liveness probe**

Liveness probe checks an Application health running inside container, and if for some reason, the Health check fails, it restarts the affected container automatically.

Liveness probe can be set by defining,

**Liveness Command**

Here, “/tmp/healthy” file is check every 5 seconds configured using “periodSeconds” parameter. The “initialDelaySeconds” parameter will requests the kubelet to wait for 3 seconds before doing First probe.

**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**

**Liveness HTTP Request**

Here, the Kubelet sends the HTTP GET request to the “**/healthz**” endpoint of the application on port 8080 within container. If that returns a failure, then Kubelet will restart the container.

**livenessProbe:**

**httpGet:**

**path: /healthz**

**port: 8080**

**httpHeaders:**

**- name: X-Custom-Header**

**value: Awesome**

**initialDelaySeconds: 3**

**periodSeconds: 3**

**TCP Liveness probe**

Here, 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**

1. **Readiness Probe**

Sometimes, applications have to meet certain conditions before they can serve traffic. These conditions may 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.

Liveness probe can be set by defining,

- Readiness HTTP Request

- TCP Readiness Probe

- Readiness Command

**readinessProbe:**

**exec:**

**command:**

**- cat**

**- /tmp/healthy**

**initialDelaySeconds: 5**

**periodSeconds:** 5

**ConfigMaps**

ConfigMaps allow us to decouple the configuration details from the Container Image. Using ConfigMaps, we can pass configuration details as key-value pair, which can be later consume by Pods, or any other system components, such as controllers. ConfigMaps can be created in 2 ways:

* From Literal Values
* From Files

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

**Secret**

With Secrets, we can share sensitive information like passwords, tokens, or keys in the form of key-value pairs, similar to ConfigMaps.

It is important to keep in mind that the Secret data is stored as **plain text inside etcd**.

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

**Ingress**

**An Ingress is collection of rules that allow inbound connection to reach the cluster Services.**

Ingress configures Layer 7 HTTP load balancer for Services and provides the following:

* TLS (Transport Layer Security)
* Name-based virtual Hosting
* Path-based Routing
* Custom Rules

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

**apiVersion: extensions/v1beta1**

**kind: Ingress**

**metadata:**

**name: cafe-ingress**

**spec:**

**tls:**

**- hosts:**

**- cafe.example.com**

**secretName: cafe-secret**

**rules:**

**- host: cafe.example.com**

**http:**

**paths:**

**- path: /tea**

**backend:**

**serviceName: tea-svc**

**servicePort: 80**

**- path: /coffee**

**backend:**

**serviceName: coffee-svc**

**servicePort: 80**

Ingress Controller

An Ingress Controller is an application, which watches API server for changes in the Ingress Resource mentioned above and update the Layer 7 Load Balancer accordingly.

Choose the ingress controller implementation that best fits your cluster, or implement a new ingress controller.

Kubernetes currently supports and maintains [GCE](https://git.k8s.io/ingress-gce/README.md) and [Nginx](https://git.k8s.io/ingress-nginx/README.md) controllers.

Type of Ingress

1. Single Service Ingress

Specifying a *default backend* with no rules.

Example:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test-ingress

spec:

backend:

serviceName: testsvc

servicePort: 80

1. Path Based Routing



Example:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test

annotations:

nginx.ingress.kubernetes.io/rewrite-target: /

spec:

rules:

- host: foo.bar.com

http:

paths:

- path: /foo

backend:

serviceName: s1

servicePort: 80

- path: /bar

backend:

serviceName: s2

servicePort: 80

1. Name based Virtual Hosting

Name-based virtual hosts use multiple host names for the same IP address.



Example:

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: test

spec:

rules:

- host: foo.bar.com

http:

paths:

- backend:

serviceName: s1

servicePort: 80

- host: bar.foo.com

http:

paths:

- backend:

serviceName: s2

servicePort: 80

1. TLS (Transport Layer Security)

Example:

apiVersion: v1

data:

tls.crt: <base64 encoded cert>

tls.key: <base64 encoded key>

kind: Secret

metadata:

name: testsecret

namespace: default

type: Opaque

---

apiVersion: extensions/v1beta1

kind: Ingress

metadata:

name: no-rules-map

spec:

tls:

- secretName: testsecret

backend:

serviceName: s1

servicePort: 80

**Annotations**

You can use either labels or annotations to attach metadata to Kubernetes objects. Labels can be use to select objects and to find collections of objects that satisfy certain conditions.

In contrast, annotations are not use to identify and select objects. Annotations can be use to:

* Store build/release IDs, PR numbers, git branch, etc.
* Phone/pager numbers of people responsible, or directory entries specifying where such information can be found
* Pointers to logging, monitoring, analytics, audit repositories, debugging tools, etc.

**ResourceQuota**

This object provides constraints that limit aggregate resource consumption per Namespace.

We can have the following types of quotas per Namespace:

* Compute Resource Quota  
  We can limit the total sum of compute resources (CPU, memory, etc.) that can be requested in a given Namespace.
* Storage Resource Quota  
  We can limit the total sum of storage resources (PersistentVolumeClaim, requests.storage, etc.) that can be requested.
* Object Count Quota  
  We can restrict the number of objects of a given type (pods, ConfigMaps, PersistentVolumeClaim, ReplicationControllers, Services, Secrets, etc.).

Users create resources (pods, services, etc.) in the namespace, and the quota system tracks usage to ensure it does not exceed hard resource limits defined in a ResourceQuota.

- If creating or updating a resource violates a quota constraint, the request will fail with HTTP status code 403 FORBIDDEN

- If quota is enabled in a namespace for compute resources like cpu and memory, users must specify requests or limits for those values; otherwise, the quota system may reject pod creation.

**Taints and Tolerations**

Taints and Tolerations work together to ensure that Pods are not schedule onto inappropriate Nodes.

Taints

Taints are apply to the Nodes, this marks that the Node should not accept any Pods that do not Tolerate the Taints. You can add the Taints to the Node using below command.

$ kubectl taint nodes host key=value:NoSchedule

The Taint has the Key “key”, Value “value”, and taint effect “NoSchedule”. This means that no pod will be able to schedule onto host unless it has a matching toleration.

To remove the taint added by the command above, you can run:

$ kubectl taint node host key:NoSchedule-

Following are the built-in Effect for the Taints.

* NoSchedule

Does not schedule Pod without matching Tolerations.

* PreferNoSchedule

Its softer version of NoSchedule effect, where it prefer not to schedule Pods without matching Tolerations.

* NoExecute

Evicts the Pods that do not have matching Tolerations.

Toleration

Tolerations are applied to the Pods, and allow the Pods to schedule onto Nodes with matching Taints. Toleration generally has 4 parts. A Key, Value, and Operator, and Effect. Default Operator, if not specified is “Equal”. There is other Operator that is “Exists”.

You specify a toleration for a pod in the PodSpec.

tolerations:

- key: "key"

operator: "Equal"

value: "value"

effect: "NoSchedule"

**Affinity: Node Affinity and Pod Affinity**

Affinity is properties of PodSpec. Affinity comes in 3 flavours:

1. Node Affinity

Node Affinity attracts Pod to certain Nodes. As of now, there are two types of Node Affinity supported,

* **requiredDuringSchedulingIgnoredDuringExecution**
* **preferredDuringSchedulingIgnoredDuringExecution**.

You can think of them as “hard” and “soft” respectively, in the sense that the former specifies rules that *must* met for a pod to be schedule onto a node. While the latter specifies *preferences* that the scheduler will try to enforce but will not guarantee.

The “**IgnoredDuringExecution**” part of the names means that, similar to how nodeSelector works, if labels on a node change at runtime (after scheduling done) such that the affinity rules on a pod are no longer met, the pod will still continue to run on the node.

Here’s an example of a pod that uses node affinity:

apiVersion: v1

kind: Pod

metadata:

name: with-node-affinity

spec:

affinity:

nodeAffinity:

requiredDuringSchedulingIgnoredDuringExecution:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/e2e-az-name

operator: In

values:

- e2e-az1

- e2e-az2

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 1

preference:

matchExpressions:

- key: another-node-label-key

operator: In

values:

- another-node-label-value

containers:

- name: with-node-affinity

image: k8s.gcr.io/pause:2.0

This rule says the Pod can only be placed on a Node with a label whose key is **kubernetes.io/e2e-az-name** and value is either e2e-az1**or**e2e-az2.

In addition, in case scheduler found more than 1 node with above labels, nodes with label whose key is **another-node-label-key**and whose value is **another-node-label-value** should be preferred with weight 1 (weight can be between 1-100).

You can see the operator In being used in the example. The new node affinity syntax supports the following operators: **In**, **NotIn**, **Exists**, **DoesNotExist**, **Gt**, **Lt**. You can use **NotIn** and **DoesNotExist** to achieve **node anti-affinity behaviour**, or use node taints to repel pods from specific nodes.

* If you specify both **nodeSelector** and **nodeAffinity**, both must be satisfied for the pod to be schedule onto a candidate node.
* If you specify multiple **nodeSelectorTerms** associated with **nodeAffinity** types, then the pod can be schedule onto a node if one of the **nodeSelectorTerms** is satisfied.
* If you specify multiple **matchExpressions** associated with **nodeSelectorTerms**, then the pod can be schedule onto a node only if all **matchExpressions** can be satisfied.

**If you remove or change the label of the node where the pod is scheduled, the pod will not be remove.**

1. Pod Affinity

Pod Affinity attracts Pod to certain Pods. As of now, there are two types of Pod Affinity supported,

* **requiredDuringSchedulingIgnoredDuringExecution**
* **preferredDuringSchedulingIgnoredDuringExecution**.

Inter-pod affinity is specify as field **podAffinity** of field **affinity** in the PodSpec. Here’s an

example of a pod that uses Pod affinity:

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: with-pod-affinity**

**spec:**

**affinity:**

**podAffinity:**

**requiredDuringSchedulingIgnoredDuringExecution:**

**- labelSelector:**

**matchExpressions:**

**- key: security**

**operator: In**

**values:**

**- S1**

**topologyKey: failure-domain.beta.kubernetes.io/zone**

**containers:**

**- name: with-pod-affinity**

**image: k8s.gcr.io/pause:2.0**

The legal operators for pod affinity and anti-affinity are **In**, **NotIn**, **Exists**, **DoesNotExist.**

In addition to **labelSelector** and **topologyKey**, you can optionally specify a list **namespaces** of namespaces that the labelSelector should match against.

1. Pod Anti-Affinity

Pod affinity furthermore has its opposite **Pod** **Anti-Affinity**, which as you would expect repels (force back, away) a Pod from other Pods. As of now, there are two types of Pod Anit-Affinity supported,

* **requiredDuringSchedulingIgnoredDuringExecution**
* **preferredDuringSchedulingIgnoredDuringExecution**.

Inter-pod affinity is specify as field **podAntiAffinity** of field **affinity** in the PodSpec. Here’s an

example of a pod that uses Pod Anti-affinity:

**apiVersion: v1**

**kind: Pod**

**metadata:**

**name: with-pod-antiaffinity**

**spec:**

**affinity:**

**podAntiAffinity:**

**preferredDuringSchedulingIgnoredDuringExecution:**

**- weight: 100**

**podAffinityTerm:**

**labelSelector:**

**matchExpressions:**

**- key: security**

**operator: In**

**values:**

**- S2**

**topologyKey: kubernetes.io/hostname**

**containers:**

**- name: with-pod-affinity**

**image: k8s.gcr.io/pause:2.0**

The legal operators for pod affinity and anti-affinity are **In**, **NotIn**, **Exists**, **DoesNotExist.**

In addition to **labelSelector** and **topologyKey**, you can optionally specify a list **namespaces** of namespaces that the labelSelector should match against.

**PodSecurityPolicy**

A PodSecurityPolicy is Cluster level resource.

The PodSecurityPolicy objects define set of conditions that Pod must obey, in order to be accepted into the System. PodSecurityPolicy are enforce by enabling the Admission Controller, but make sure you have added the Policies into the Cluster before enabling PodSecurityPolicy feature; otherwise, it **will prevent you from creating any Pods in cluster**.

|  |  |
| --- | --- |
| **Field Names** | **Control Aspect** |
| [privileged](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#privileged) | Running of privileged containers |
| [hostPID, hostIPC](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#host-namespaces) | Usage of the root namespaces |
| [hostNetwork, hostPorts](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#host-namespaces) | Usage of host networking and ports |
| [volumes](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#volumes-and-file-systems) | Usage of volume types |
| [allowedHostPaths](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#volumes-and-file-systems) | Usage of the host filesystem |
| [allowedFlexVolumes](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#flexvolume-drivers) | White list of FlexVolume drivers |
| [fsGroup](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#volumes-and-file-systems) | Allocating an FSGroup that owns the pod’s volumes |
| [readOnlyRootFilesystem](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#volumes-and-file-systems) | Requiring the use of a read only root file system |
| [runAsUser, supplementalGroups](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#users-and-groups) | The user and group IDs of the container |
| [allowPrivilegeEscalation, defaultAllowPrivilegeEscalation](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#privilege-escalation) | Restricting escalation to root privileges |
| [defaultAddCapabilities, requiredDropCapabilities, allowedCapabilities](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#capabilities) | Linux capabilities |
| [seLinux](https://v1-10.docs.kubernetes.io/docs/concepts/policy/pod-security-policy/#selinux) | The SELinux context of the container |

When a PodSecurityPolicy resource is created, it does nothing. In order to use it, the requesting user or ServiceAccount must be authorize. Cluster admin can authorize the User/ServiceAccount by allowing the **use** verb on the policy.

Most Kubernetes pods are not created directly by users. Instead, they are typically created indirectly as part of a Deployment, ReplicaSet, or other templated controller via the controller manager. Granting the controller access to the policy would grant access for all pods created by that the controller, so the preferred method for authorizing policies is to grant access to the Pods ServiceAccount.

A security context defines privilege and access control settings for a Pod or Container. Security context settings include:

* Discretionary Access Control (DAC): Permission to access an object, like a file, is base on user ID (UID) and group ID (GID).
* Security Enhanced Linux (SELinux): Objects are assigned security labels.
* Running as privileged or unprivileged.
* Linux Capabilities: Give a process some privileges, but not all the privileges of the root user.
* AppArmor: Use program profiles to restrict the capabilities of individual programs.
* Seccomp: Filter a process’s system calls.
* AllowPrivilegeEscalation: Controls whether a process can gain more privileges than its parent process. AllowPrivilegeEscalation is true always when the container is:

1) Run as Privileged OR

2) Has CAP\_SYS\_ADMIN

Security settings that you specify for a Container apply only to the individual Container, and they override settings made at the Pod level when there is overlap.

**Policy Authorization via RBAC**

RBAC is a standard Kubernetes authorization mode, and can easily be use to authorize use of policies.

**kind: ClusterRole**

**apiVersion: rbac.authorization.k8s.io/v1**

**metadata:**

**name: <role name*>***

***rules:***

***- apiGroups: ['policy']***

***resources: ['podsecuritypolicies']***

***verbs: ['use']***

***resourceNames:***

***- <list of policies to authorize>***

Then the (Cluster) Role is bound to the authorized user(s):

**kind: ClusterRoleBinding**

**apiVersion: rbac.authorization.k8s.io/v1**

**metadata:**

**name: <binding name*>***

***roleRef:***

***kind: ClusterRole***

***name: <role name>***

***apiGroup: rbac.authorization.k8s.io***

***subjects:***

***# Authorize specific service accounts:***

***- kind: ServiceAccount***

***name: <authorized service account name>***

***namespace: <authorized pod namespace>***

***# Authorize specific users (not recommended):***

***- kind: User***

***apiGroup: rbac.authorization.k8s.io***

***name: <authorized user name>***

In case you use **RoleBinding**, it will only grant usage for pods being run in the same namespace as the binding. You can also grant access to System Group that allow all users to create Pod.

**Note:**

The Kube Controller Manager must run against the Secured API server (https), otherwise all requests would bypass the Authentication and Authorization module and all **PodSecurityPolicy** would be allowed.

Policy Order

When multiple policies are available, the PodSecurityPolicy controller selects policies in the following order:

1. If any policies successfully validate the pod without altering it, they are used.
2. If it is a pod creation request, then the first valid policy in alphabetical order is used.

**Pod Lifecycle**

1. Pod Phase

The phase of a Pod is a simple, high-level summary of where the Pod is in its lifecycle. Below are different phase,

* **Pending**

The Pod has been accepted by the Kubernetes system. The time before being Scheduled or time spent downloading Container Images over the Network.

* **Running**

The Pod has been bound to the Node, and all the Containers have been created. At least one container is still running, or is in the process of Starting or Restarting.

* **Succeeded**

All Containers in the Pod have terminated in success, and will not be restarted.

* **Failed**

All Containers in the Pod have terminated, and at least one Container has terminated in failure.

* **Unknown**

For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Pod.

1. Pod Conditions

A Pod has PodStatus, which has array of PodConditions through which Pod has or has not passed. Below are the PodConditions,

* **Initialized**

The init containers in the Pod have started successfully

* **PodScheduled**

The Pod has been scheduled

* **Ready**

The Pod is able to service requests and all Containers in Pod are currently ready.

1. Container Probes

A **Probe** is diagnostic performed by the Kubelet on a Container. To perform diagnostic, the kubelet calls a **Handler** implemented by the Container.

There are 3 types of Handler,

* **ExecAction**

Executes a command inside the Container. The diagnostic is consider successful if the command exits with status code of zero.

* **TCPSocketAction**

Performs TCP check against the Containers IP address on specific Port. The diagnostic is consider successful if the Port is Open.

* **HTTPGetAction**

Performs an HTTP Get request against the IP address on specific Port and Path. The diagnostic is consider successful if the response has status code greater than or equal to 200 and less than 400.

1. **livenessProbe** and **readinessProbe**

* **livenessProbe**

Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its [restart policy](https://v1-10.docs.kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy).

If a Container does not provide a liveness probe, the default state is **Success**.

* **readinessProbe**

Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod’s IP address from the endpoints of all Services that match the Pod.

The default state of readiness before the initial delay is **Failure**.

If a Container does not provide a readiness probe, the default state is **Success**.

1. Container Restart Policy

* **Always**

Means that the container will be restarted even if it exited with a zero exit code (i.e. successfully). This is useful when you don't care why the container exited, you just want to make sure that it is always running (e.g. a web server). This is the default.

* **OnFailure**

Means that the container will only be restarted if it exited with a non-zero exit code (i.e. something went wrong). This is useful when you want accomplish a certain task with the pod, and ensure that it completes successfully - if it doesn't it will be restarted until it does.

* **Never**

Means that the container will not be restarted regardless of why it exited.

**ETCD**

ETCD stores its configuration into a data directory specified by the data-dir configuration parameter. Configuration is stored in the write ahead log (WAL) and includes: the local member ID, cluster ID, and initial cluster configuration. The write ahead log and snapshot files are used during member operation and to recover after a restart.

If a member’s data directory is ever lost or corrupted then the user should remove the etcd member from the cluster using etcdctl tool.

A user should avoid restarting an etcd member with a data directory from an out-of-date backup.

The data directory has two sub-directories in it:

1) wal: write ahead log files are stored here.

2) snap: log snapshots are stored here.

Each ETCD cluster has unique initial-cluster-token, all members that are part of cluster share the same token.

**Flannel Network**

Kubernetes networking model requires:

- All containers can communicate with all other containers without NAT

- All nodes can communicate with all containers (and vice-versa) without NAT

- The IP that a container sees itself as is the same IP that others see it as

Flannel runs a small, single binary agent called **flanneld** on each host, and is responsible for allocating a subnet lease to each host out of a larger, preconfigured address space. Flannel uses either the Kubernetes API or etcd directly to store the network configuration, the allocated subnets, and any auxiliary data (such as the host's public IP). Packets are forwarded using one of several backend mechanisms like,

* VXLAN (Recommended)

Use in-kernel VXLAN to encapsulate the packets.

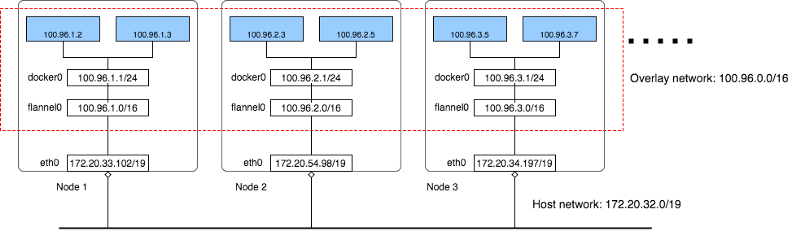
Type and options:

* Type (string): vxlan
* VNI (number): VXLAN Identifier (VNI) to be use. Defaults to 1.
* Port (number): UDP port to use for sending encapsulated packets. Defaults to kernel default, currently **8472**.
* GBP (Boolean): Enable VXLAN Group Based Policy. Defaults to false.
* DirectRouting (Boolean): Enable direct routes (like host-gw) when the hosts are on the same subnet. VXLAN will only be use to encapsulate packets to hosts on different subnets. Defaults to false.
* UDP (Old Kernel that does not support VXLAN)

Use UDP only for debugging if your network and kernel prevent you from using VXLAN.

Type and options:

* Type (string): udp
* Port (number): UDP port to use for sending encapsulated packets. Defaults to 8285.
* In each host, **flannel** runs a daemon process called **flanneld**, which creates some route rules in kernel’s route table.
* Flanneld daemon also creates a TUN device (Software Interface) called **flannel0.** This TUN is software interface implemented in Linux Kernel. It can pass raw IP packet between the User program and the kernel.
* Flannel stores the subnet to host mapping information into the etcd datastore.



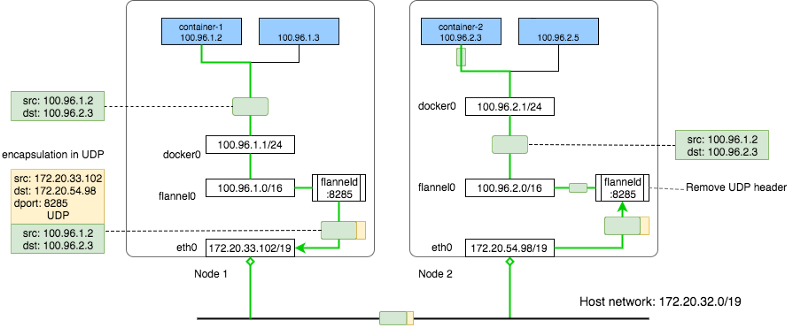
* All Host VMs are part of Host Network subnet 172.20.32.0/19 that are connected over LAN.
* Flannel has created another network 100.96.0.0/16 across all Kubernetes Node, which can hold upto 65536 addresses.
* Inside each Kubernetes Node, flannel assigned a 100.96.x.0/24 network to all pods in this host, it can hold upto 2⁸(256) addresses. The Docker Bridge will use this network to assign IP to Container.

1. Container Communication within same Node

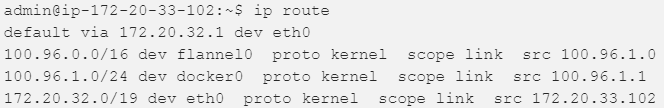
The Container inside same Node can communicate to other Container on same host using bridge **docker0**, which is simple.

1. Container Communication across Node

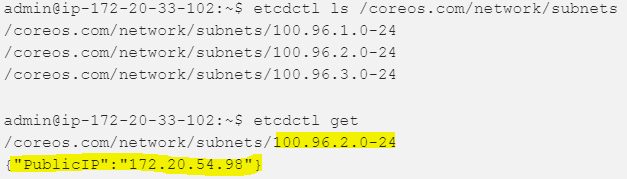
Scenario: The container-1 in Node 1 that has the IP address 100.96.1.2 wants to connect to the container-2 in Node 2 with the IP address 100.96.2.3.



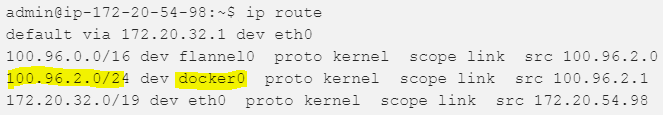
1. Container-1 creates an IP packet with src: 100.96.1.2 -> dst: 100.96.2.3, the packet will be send to gateway that is **docker0** bridge.
2. Bridge docker0 will refer to the Kernel Route table (created by Flanneld). As we can see, the packet’s destination address 100.96.2.3 falls in the bigger overlay network 100.96.0.0/16, so it matches the second rule, now kernel knows it should send the packet to **flannel0**.



1. When an IP packet arrives to the kernel via **flannel0** (TUN device), kernel will send the packet directly to the process which created this device which is the **flanneld** daemon process.
2. Now flanneld process queries etcd to know each subnet belongs to which host, and compares the destination ip address with all subnets key stored in etcd. In our case, the address 100.96.2.3 will match the subnet 100.96.2.0-24, and as we see the value stored in this keys says the Node ip is 172.20.54.98.



1. Flanneld then wraps the original IP packet into a UDP packet, with it’s own host’s ip as source address, and the target host’s IP as destination address. In each host, the flanneld process will listen on a default **UDP port 8285**. So it just need to set the UDP packet’s destination port to 8285, and send it on the wire.
2. After the UDP packet arrives at the **destination host**, the kernel’s IP stack will send the packet to the flanneld process because that’s the user process who listens on UDP port 8285.
3. Then flanneld will get the payload of the UDP packet, which is the original IP packet generated by the originate container, it simply write this packet to the TUN device **flannel0**. That in turn will be forwarded to Kernel.
4. The kernel will refer to the Route Table and will decide where this packet should go. The destination address of the IP packet is 100.96.2.3, kernel will take the most precise match, which is the third rule. The packet will be send to **docker0** device.



**Calico Network**

Calico leverages the routing and iptable firewall capabilities native to the Linux kernel. All traffic to and from individual containers, virtual machines, and hosts traverses these in-kernel rules before being routed to its destination.

* **Calicoctl**:

Allows you to achieve advanced policies and networking from a simple, command-line interface.

* **Orchestrator Plugins:**

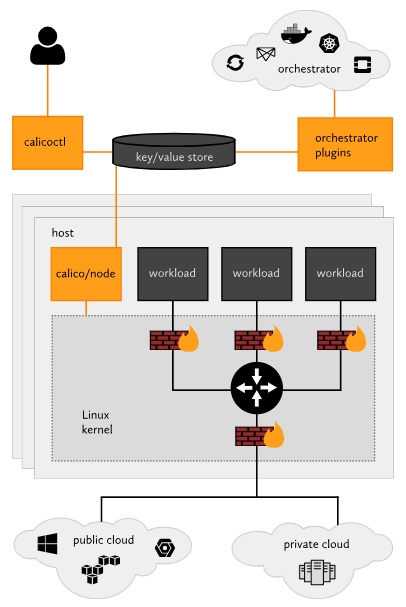
It provide close integration and synchronization with a variety of popular orchestrators.

* **Key/value store:**

It holds Calico’s policy and network configuration state.

* **Calico/node:**

It runs on each host, reads relevant policy and network configuration information from the key/value store, and implements it in the Linux kernel.



Components

Calico setup has 4 components

* **ETCD**

Etcd is the backend data store for all the information Calico needs. The default location for the calico keys in ETCD is /calico.

* **BIRD**

BIRD is a BGP routing daemon which runs on every host. Calico makes uses of BGP to propagate routes between hosts. It’s included in the calico/node container.

* **Confd**

Confd is a simple configuration management tool. It reads values from etcd and writes them to files on disk. Essentially, confd is what writes the BIRD configuration for Calico.

* **calico-felix**

It has multiple responsibilities:

- It writes the routing table of the operating system.

- It manipulates IPtables on the host.

**Service Account**

You must pass a service account private key file to the token controller in the controller-manager by using the **--service-account-private-key-file** option. The private key will be used to sign generated service account tokens. Similarly, you must pass the corresponding public key to the kube-apiserver using the **--service-account-key-file** option. The public key will be used to verify the tokens during authentication.

Kubernetes API Server to access the Kubelet API on each worker node. Access to the Kubelet API is required for retrieving metrics, logs, and executing commands in pods.

The Kubernetes API Server authenticates to the Kubelet as the kubernetes user using the client certificate as defined by the --kubelet-client-certificate flag.

Node authorization is a special-purpose authorization mode that specifically authorizes API requests made by kubelets.

**Deployment**

A Deployment runs multiple replicas of your application and automatically replaces any instances that fail or become unresponsive. A deployment is a supervisor for pods and replica sets, giving you fine-grained control over how and when a new pod version is **rolleout** as well as **rollback** to a previous state.

When you inspect the Deployments in your cluster, the below fields are displayed:



Here,

* NAME:

Lists the names of the Deployments in the cluster.

* DESIRED:

Displays the desired number of *replicas* of the application, which you define when you create the Deployment. This is the *desired state*.

* CURRENT:

Displays how many replicas are currently running.

* UP-TO-DATE:

Displays the number of replicas that have been updated to achieve the desired state.

* AVAILABLE:

Displays how many replicas of the application are available to your users.

* AGE:

Displays the amount of time that the application has been running.

To see ReplicaSet (rs) created by Deployment, run kubectl get rs



Notice, the name of the ReplicaSet is always formatted as [DEPLOYMENT-NAME]-[POD-TEMPLATE-HASH-VALUE]. The hash value is generated automatically when the Deployment is created.

Pod-Template-hash Label

The **pod-template-hash** label is added by the Deployment controller to every ReplicaSet that a Deployment creates.

This label ensures that child ReplicaSet of Deployment do not overlap. Hash value is generated by hashing the PodTemplate of the ReplicaSet. This Hash value is added as Label to the ReplicaSet selector, Pod template labels.

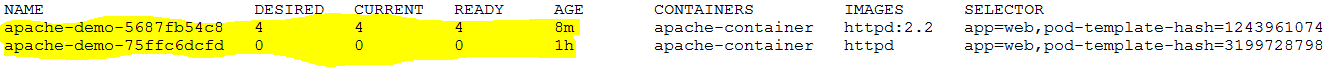
Updating Deployment

Suppose we want to update the apache pod to use 2.2 version instead of latest. We can either run,

$ kubectl set image deployment/<DEPLOYMENT\_NAME> <CONTAINER\_NAME>=<IMAGE\_NAME>:<IMAGE\_VERSION>

$ kubectl set image deployment/apache-demo apache-container=httpd:2.2

We can run kubectl get replicaset to see that the Deployment updated the Pods by creating a new ReplicaSet and scaling it up to 4 replicas, as well as scaling down the old ReplicaSet to 0 replicas.

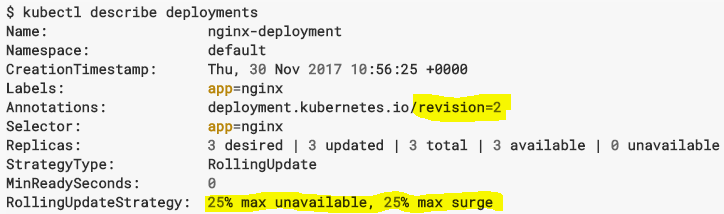


Eventually, the new ReplicaSet will be scaled to desired replica and all old ReplicaSets will be scaled to 0.

Deployment Strategy

Deployment can ensure that only a certain number of Pods may be down while they are being updated. By default, it ensures that at least 25% of desired number of Pods are down for update (25% max unavailable).

Deployment can also ensure that only a certain number of Pods may be created above the desired number of Pods. By default, it ensures that at most 25% more than the desired number of Pods are up (25% max surge).



Rolling Back Deployment

Sometimes you may want to rollback a Deployment; for example, when the Deployment is not stable, such as crash looping.(You can change revision history limit).

**Note:** The new Deployment revision is created if and only if the Deployment’s pod template (.spec.template) is changed. For example if you update the labels or container images of the template. Other updates, such as scaling the Deployment, do not create a Deployment revision.

You can perform rolling back to,

1. Previous version

$ kubectl rollout undo deployment apache-demo

1. To a Specific version

$ kubectl rollout undo deployment apache-demo --to-revision=1

Revision history can be view as,

$ kubectl rollout history deployment apache-demo

Pause and Resume Deployment

The Deployment can be Pause and Resume in the middle using below commands,

$ kubectl rollout pause deployment apache-demo

$ kubectl rollout resume deployment apache-demo

Revision History Limit

You can set .spec.revisionHistoryLimit field in a Deployment to specify how many old ReplicaSets for this Deployment you want to retain. The rest will be garbage-collected in the background. By default, it is 10.

Failed Deployment

Your Deployment may get stuck trying to deploy its newest ReplicaSet without ever completing. This can occur due to some of the following factors:

* Insufficient quota
* Readiness probe failures
* Image pull errors
* Insufficient permissions
* Limit ranges
* Application runtime misconfiguration

Deployment Strategy

.spec.strategy specifies the strategy used to replace old Pods by new ones. .spec.strategy.type can be “Recreate” or “RollingUpdate” (default).

1. Recreate

All existing Pods are kill before new ones are created.

1. RollingUpdate

The Deployment updates Pods in a rolling update fashion. You can specify maxUnavailable and maxSurge to control the rolling update process.

**DaemonSet**

A DaemonSet ensures that **all (or some) Nodes** run a copy of a Pod. As nodes are added to the cluster, Pods are added to them. As nodes are remove from the cluster, those Pods are garbage collected.

Some typical uses of a DaemonSet are:

* Running a cluster storage daemon, such as glusterd, ceph, on each node.
* Running a logs collection daemon on every node, such as fluentd or logstash.
* Running a node monitoring daemon on every node, such as Prometheus Node Exporter, collectd, Datadog agent, etc.

A Pod Template in a DaemonSet must have a RestartPolicy equal to Always, or be unspecified, which defaults to Always.

Running Pods on Only Some Nodes

If you specify a .spec.template.spec.nodeSelector, then the DaemonSet controller will create Pods on nodes that match that node selector. Likewise, if you specify a .spec.template.spec.affinity, then DaemonSet controller will create Pods on nodes that match that node affinity. If you do not specify either nodeSelector or nodeAffinity, then the DaemonSet controller will create Pods on all nodes.

(**Note**: If there are taint on node, pod template should have toleration to be able to schedule on those nodes.)

How DaemonSet Pods are Schedule

Normally, the Kubernetes Scheduler selects the machine that a Pod should runs on. However, Pods created by the DaemonSet controller have the machine already selected (.spec.nodeName is specified when the Pod is created, so it is ignored by the scheduler).

Note:

1. The DaemonSet controller does not respect the unschedulable field of a node (set via “kubectl cordon <node\_name>”.
2. The DaemonSet controller can make Pods even when the scheduler has not started, which can help cluster bootstrap.

Communicating with DaemonSet Pod

Some possible patterns for communicating with Pods in a DaemonSet are:

* Push:

Configure the Pods in the DaemonSet to send updates to another service, such as a stats database or server.

* NodeIP and Port

Pods in the DaemonSet can use a hostPort, so that the pods are reachable via the node IPs and specified port.

* DNS

Create a headless service with the same pod selector, and then discover DaemonSets using the endpoints resource or retrieve multiple A records from DNS.

**Volume**

First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a Pod it is often necessary to share files between those Containers.

At its core, a volume is just a directory, possibly with some data in it, which is accessible to the Containers in a Pod. Kubernetes supports many types of volumes, and a **Pod can use any number of them simultaneously**.

Volumes cannot mount onto other volumes or have hard links to other volumes. Each Container in the Pod must independently specify where to mount each volume.

Types of Volumes

Kubernetes supports several types of Volumes:

1. **awsElasticBlockStore**

An awsElasticBlockStore volume mounts an Amazon Web Services (AWS) EBS Volume into your Pod. There are some restrictions when using an awsElasticBlockStore volume:

- The nodes on which Pods are running must be AWS EC2 instances

- Those instances need to be in the same region and availability-zone as the EBS volume

- EBS only supports a single EC2 instance mounting a volume

Example:

apiVersion: v1

kind: Pod

metadata:

name: test-ebs

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-ebs

name: test-volume

volumes:

- name: test-volume

# This AWS EBS volume must already exist.

awsElasticBlockStore:

volumeID: <volume-id>

fsType: ext4

1. **azureDisk**
2. **azureFile**
3. **configMap**

The configMap resource provides a way to inject configuration data into Pods. The data stored in a ConfigMap object can be reference in a volume of type configMap and then consumed by containerized applications running in a Pod.

Example:

apiVersion: v1

kind: Pod

metadata:

name: configmap-pod

spec:

containers:

- name: test

image: busybox

volumeMounts:

- name: conifgmap-volume

mountPath: /etc/config

volumes:

- name: configmap-volume

configMap:

name: log-config

items:

- key: log\_level

path: log\_level

1. **emptyDir**

An emptyDir volume is created first when a Pod is assign to a Node, and exists as long as that Pod is running on that node. As the name says, it is initially empty.

Containers in the Pod can all read and write the same files in the emptyDir volume, though that volume can be mounted at the same or different paths in each Container.

When a Pod is remove from a node for any reason, the data in the emptyDir is deleted forever.

By default, emptyDir volumes are stored on whatever medium is backing the node, i.e SSD, network storage, etc.

However, you can set the emptyDir.medium field to "Memory" to tell Kubernetes to mount a tmpfs (RAM-backed filesystem) for you instead. While tmpfs is very fast, be aware that unlike disks, tmpfs is clear on node reboot and any files you write will count against your Container’s memory limit.

Example:

apiVersion: v1

kind: Pod

metadata:

name: emptyvol-pod

spec:

containers:

- name: test-container

image: k8s.gcr.io/test-webserver

volumeMounts:

- name: cache-volume

mountPath: /cache

volumes:

- name: cache-volume

emptyDir:

medium: Memory

1. **fc (fibre channel)**

An fc volume allows an existing fibre channel volume to be mounted in a Pod.

1. **gcePersistentDisk**

A gcePersistentDisk volume mounts a Google Compute Engine (GCE) Persistent Disk into your Pod. There are some restrictions when using a gcePersistentDisk:

- The nodes on which Pods are running must be GCE VMs

- Those VMs need to be in the same GCE project and zone as the PD

Persistent Disk (PDs) can only be mounted in read-write mode by a single consumer - no simultaneous writers allowed.

1. **glusterfs**

A glusterfs volume allows a Glusterfs (an open source networked filesystem) volume to be mounted into your Pod.

1. **hostPath**

A hostPath volume mounts a file or directory from the host node’s filesystem into your Pod. Some uses for a hostPath are:

- running a Container that needs access to Docker internals; use a hostPath of /var/lib/docker

- running cAdvisor in a Container; use a hostPath of /sys

In addition to the required path property, user can optionally specify a type for a hostPath volume. The supported values for field type are:

Some of caveat using hostPath volume,

- Pods with identical configuration (such as created from a podTemplate) may behave differently on different nodes due to different files on the nodes.

- when Kubernetes adds resource-aware scheduling, as is planned, it will not be able to account for resources used by a hostPath

- the files or directories created on the underlying hosts are only writable by root. You either need to run your process as root in a privileged Container or modify the file permissions on the host to be able to write to a hostPath volume

Example:

apiVersion: v1

kind: Pod

metadata:

name:

spec:

containers:

- name:

image:

volumeMounts:

- name:

mountPath:

volumes:

- name: test-volume

hostPath:

path: /pod-data

type: Directory

1. **iscsi**

An iscsi volume allows an existing iSCSI (SCSI over IP) volume to be mount into your Pod. A feature of iSCSI is that multiple consumers can mount it as read-only simultaneously. Unfortunately, iSCSI volumes can only be mounted by a single consumer in read-write mode - no simultaneous writers allowed.

1. **local**

A local volume represents a mounted local storage device such as a disk, partition or directory.

Local volumes can only be use as a statically created PersistentVolume. Dynamic provisioning is not supported yet.

Compared to hostPath volumes, local volumes can be use in a durable and portable manner without manually scheduling Pods to nodes relation.

However, local volumes are still subject to the availability of the underlying node and are not suitable for all applications. If a node becomes unhealthy, then the local volume will also become inaccessible, and a Pod using it will not be able to run.

Example:

apiVersion: v1

kind: PersistentVolume

metadata:

name: local-pv

spec:

capacity: 100Gi

# volumeMode field requires BlockVolume Alpha feature gate to be enabled.

volumeMode: Filesystem

accessModes:

- ReadWriteOnce

persistentVolumeReclaimPolicy: Delete

storageClassName: local-storage

local:

path: /mnt/disks/ssd1

nodeAffinity:

required:

nodeSelectorTerms:

- matchExpressions:

- key: kubernetes.io/hostname

operator: In

values:

- example-node

PersistentVolume nodeAffinity is required when using local volumes. PersistentVolume volumeMode can now be set to “Block” (instead of the default value “Filesystem”) to expose the local volume as a raw block device. The volumeMode field requires BlockVolume Alpha feature gate to be enabled.

1. **nfs**

An nfs volume allows an existing NFS (Network File System) share to be mounted into your Pod. Multiple consumer can mount NFS simultaneously.

1. **persistentVolumeClaim**

A persistentVolumeClaim volume is use to mount a PersistentVolume into a Pod.

1. **secret**

A secret volume is use to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by Pods without coupling to Kubernetes directly. Secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage.

1. **downwordAPI**

There are two ways to expose Pod and Container fields to a running Container:

* Environments Variables
* DownwardAPIVolumeFiles

Together, these two ways of exposing Pod and Container fields are call the Downward API.

The following information is available to containers through environment variables and downwardAPI volumes:

1. Information available via fieldRef:

* spec.nodeName - the node’s name
* status.hostIP - the node’s IP
* metadata.name - the pod’s name
* metadata.namespace - the pod’s namespace
* status.podIP - the pod’s IP address
* spec.serviceAccountName - the pod’s service account name
* metadata.uid - the pod’s UID
* metadata.labels - all of the pod’s labels
* metadata.labels['<KEY>'] - the value of the pod’s label <KEY> (for example, metadata.labels['mylabel']); available in Kubernetes 1.9+
* metadata.annotations - all of the pod’s annotations
* metadata.annotations['<KEY>'] - the value of the pod’s annotation <KEY> (for example, metadata.annotations['myannotation']); available in Kubernetes 1.9+

1. Information available via resourceFieldRef:

* A Container’s CPU limit
* A Container’s CPU request
* A Container’s memory limit
* A Container’s memory request

Example:

apiVersion: v1

kind: Pod

metadata:

name: kubernetes-downwardapi-volume-example

labels:

zone: us-est-coast

cluster: test-cluster1

rack: rack-22

annotations:

build: two

builder: john-doe

spec:

containers:

- name: client-container

image: k8s.gcr.io/busybox

command: ["sh", "-c"]

args:

- while true; do

if [[ -e /etc/podinfo/labels ]]; then

echo -en '\n\n'; cat /etc/podinfo/labels; fi;

if [[ -e /etc/podinfo/annotations ]]; then

echo -en '\n\n'; cat /etc/podinfo/annotations; fi;

sleep 5;

done;

volumeMounts:

- name: podinfo

mountPath: /etc/podinfo

readOnly: false

volumes:

- name: podinfo

downwardAPI:

items:

- path: "labels"

fieldRef:

fieldPath: metadata.labels

- path: "annotations"

fieldRef:

fieldPath: metadata.annotations

1. **Projected**

A projected volume maps several existing volume sources into the same directory. Currently, the following types of volume sources can be projected:

* secret
* downwardAPI
* configMap
* serviceAccountToken

All sources are required to be in the same namespace as the Pod.

Example:

apiVersion: v1

kind: Pod

metadata:

name: volume-test

spec:

containers:

- name: container-test

image: busybox

volumeMounts:

- name: all-in-one

mountPath: "/projected-volume"

readOnly: true

volumes:

- name: all-in-one

projected:

sources:

- secret:

name: mysecret

items:

- key: username

path: my-group/my-username

- downwardAPI:

items:

- path: "labels"

fieldRef:

fieldPath: metadata.labels

- path: "cpu\_limit"

resourceFieldRef:

containerName: container-test

resource: limits.cpu

- configMap:

name: myconfigmap

items:

- key: config

path: my-group/my-config

1. **vsphereVolume**

A vsphereVolume is use to mount a vSphere VMDK Volume into your Pod. The contents of a volume are preserve when it is unmounted. It supports both VMFS and VSAN datastore.

Example:

apiVersion: v1

kind: Pod

metadata:

name: test-vmdk

spec:

containers:

- image: k8s.gcr.io/test-webserver

name: test-container

volumeMounts:

- mountPath: /test-vmdk

name: test-volume

volumes:

- name: test-volume

# This VMDK volume must already exist.

vsphereVolume:

volumePath: "[DatastoreName] volumes/myDisk"

fsType: ext4

Using subPath

Sometimes, it is useful to share one volume for multiple uses in a single Pod. The volumeMounts.subPath property can be use to specify a sub-path inside the referenced volume.

Example:

apiVersion: v1

kind: Pod

metadata:

name: my-lamp-site

spec:

containers:

- name: mysql

image: mysql

env:

- name: MYSQL\_ROOT\_PASSWORD

value: "rootpasswd"

volumeMounts:

- mountPath: /var/lib/mysql

name: site-data

subPath: mysql

- name: php

image: php:7.0-apache

volumeMounts:

- mountPath: /var/www/html

name: site-data

subPath: html

volumes:

- name: site-data

persistentVolumeClaim:

claimName: my-lamp-site-data

Here is an example of a Pod with a LAMP stack (Linux Apache Mysql PHP) using a single, shared volume. The HTML contents are map to its **html** folder, and the databases will be stored in its **mysql** folder.

**Persistent Volume**

A **PersistentVolume (PV)** is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster just like a other resources like node is a cluster resource. PV have a lifecycle independent of any individual pod that uses the PV.

A **PersistentVolumeClaim (PVC)** is a request for storage by a user. PVCs consume PV resources. Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).

Cluster administrators need to be able to offer a variety of PersistentVolume (storage) that differ in more ways than just size and access modes.

Lifecycle of a PV (PersistentVolume) and PVC (PersistentVolumeClaim)

PVs are resources in the cluster. PVCs are requests for those resources. The interaction between PVs and PVCs follows this lifecycle:

1. Provisioning

There are two ways PVs may be provisioned: statically or dynamically.

- **Static**

A cluster administrator creates a number of PVs. They carry the details of the real storage, which is available for use by cluster users.

- **Dynamic**

When none of the static PVs the administrator created matches a user PersistentVolumeClaim, the cluster may try to dynamically provision a volume especially for the PVC.

This provisioning is based on **StorageClasses**: the PVC must request a storage class and the administrator must have created and configured that class in order for dynamic provisioning to occur.

To enable dynamic storage provisioning based on storage class, the cluster administrator needs to enable the DefaultStorageClass admission controller on the API server.

1. Binding

A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together. If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.

Claims will remain unbound indefinitely if a matching volume does not exist. For example, a cluster provisioned with many 50Gi PVs would not match a PVC requesting 100Gi. The PVC can be bound when a 100Gi PV is added to the cluster.

1. Using

Pods use claims as volumes. The cluster inspects the claim to find the bound volume and mounts that volume for a pod. For volumes that support multiple access modes, the user specifies which mode is desire when using their claim as a volume in a pod.

1. Reclaiming

When a user is done with their volume, they can delete the PVC objects from the API which allows reclamation of the resource. The reclaim policy for a PersistentVolume tells the cluster what to do with the volume after it has been released of its claim. Currently, volumes can either be,

* **Retain**

The Retain reclaim policy allows for manual reclamation of the resource. When the PersistentVolumeClaim is deleted, the PersistentVolume still exists and the volume is considered “released”.

However, it is not yet available for another claim because the previous claimant’s data remains on the volume.

An administrator can manually reclaim the volume with the following steps.

* 1. Delete the PersistentVolume. The associated storage asset in external infrastructure (such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume) still exists after the PV is deleted.
  2. Manually clean up the data on the associated storage asset accordingly.
  3. Manually delete the associated storage asset, **or** if you want to reuse the same storage asset, create a new PersistentVolume with the storage asset definition.
* **Recycle**

If supported by the underlying volume plugin, the Recycle reclaim policy performs a basic scrub (rm -rf /thevolume/\*) on the volume and makes it available again for a new claim.

* **Delete**

For volume plugins that support the Delete reclaim policy, deletion removes both the PersistentVolume object from Kubernetes, as well as the associated storage asset in the external infrastructure, such as an AWS EBS, GCE PD, Azure Disk, or Cinder volume.

Volumes that were dynamically provisioned inherit the reclaim policy of their StorageClass, which defaults to Delete.

Persistent Volumes Claims Expansion

Administrator can allow expanding persistent volume claims by setting ExpandPersistentVolumes feature gate to true. Administrator should also enable PersistentVolumeClaimResize admission plugin to perform additional validations of volumes that can be resized.

Once PersistentVolumeClaimResize admission plug-in has been turned on, resizing will only be allowed for storage classes whose allowVolumeExpansion field is set to true.

Once both feature gate and the aforementioned admission plug-in are turned on, a user can request larger volume for their PersistentVolumeClaim by simply editing the claim and requesting a larger size. This in turn will trigger expansion of the volume that is backing the underlying PersistentVolume.

Following volume types support expanding Persistent volume claims:

* gcePersistentDisk
* awsElasticBlockStore
* Cinder
* glusterfs
* rbd

For expanding volumes containing a file system, file system resizing is only performed when a new Pod is started using the PersistentVolumeClaim in ReadWrite mode. In other words, if a volume being expanded is used in a pod or deployment, you will need to delete and recreate the pod for file system resizing to take place.

File system resizing is only supported for following file system types:

* XFS
* Ext3, Ext4

Persistent Volume (PV) Specification

1. **Capacity**

Generally, a PV will have a specific storage capacity. This is set using the PV’s capacity attribute.

Currently, storage size is the only resource that can be set or requested. Future attributes may include IOPS, throughput, etc.

1. **Volume Mode**

Valid values for volumeMode are “Filesystem” or “Block”. If left unspecified, volumeMode defaults to “Filesystem” internally.

1. **Access Mode**

The access modes are:

ReadWriteOnce – the volume can be mounted as read-write by a single node

ReadOnlyMany – the volume can be mounted read-only by many nodes

ReadWriteMany – the volume can be mounted as read-write by many nodes

For example, NFS can support multiple read/write clients, but a specific NFS PV might be exported on the server as read-only.

1. **Class**

A PV can have a class, which is specified by setting the storageClassName attribute to the name of a StorageClass.

A PV of a particular class can only be bound to PVCs requesting that class.

A PV with no storageClassName has no class and can only be bound to PVCs that request no particular class.

1. **Reclaim Policy**

Current reclaim policies are:

Retain – manual reclamation

Recycle – basic scrub (rm -rf /thevolume/\*)

Delete – associated storage asset such as AWS EBS, GCE PD, Azure Disk, or OpenStack Cinder volume is deleted.

Currently, only NFS and HostPath support recycling.

AWS EBS, GCE PD, Azure Disk, and Cinder volumes support deletion.

1. **Mount Option**

A Kubernetes administrator can specify additional mount options for when a Persistent Volume is mounted on a node.

Persistent Volume (PV) Phase

A volume will be in one of the following phases:

* Available – a free resource that is not yet bound to a claim
* Bound – the volume is bound to a claim
* Released – the claim has been deleted, but the resource is not yet reclaimed by the cluster
* Failed – the volume has failed its automatic reclamation

Persistent Volume Claim (PVC) Specification

1. **Access Mode**

Claims use the same conventions as volumes when requesting storage with specific access modes.

1. **Volume Modes**

Claims use the same convention as Persistent Volume to indicates the consumption of the volume as either a filesystem or block device.

1. **Resources**

Claims, like pods, can request specific quantities of a resource. In this case, the request is for storage. The same resource model applies to both volumes and claims.

1. **Selector**

Claims can specify a label selector to further filter the set of volumes. Only the volumes whose labels match the selector can be bound to the claim.

1. **Class**

A claim can request a particular class by specifying the name of a StorageClass using the attribute storageClassName. Only PVs of the requested class, ones with the same storageClassName as the PVC, can be bound to the PVC.

A PVC with its storageClassName set equal to "" is always interpreted to be requesting a PV with no class, so it can only be bound to PVs with no class (no annotation or one set equal to "").

Claims As Volumes

Pods access storage by using the claim as a volume. Claims must exist in the same namespace as the pod using the claim.

Example:

kind: Pod

apiVersion: v1

metadata:

name: mypod

spec:

containers:

- name: myfrontend

image: dockerfile/nginx

volumeMounts:

- mountPath: "/var/www/html"

name: mypd

volumes:

- name: mypd

persistentVolumeClaim:

claimName: myclaim

**Storageclass**

A StorageClass provides a way for administrators to describe the “classes” (types) of storage they offer.

Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators.

Each StorageClass contains the fields **provisioner**, **parameters**, and **reclaimPolicy**, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned (create).

The name of a StorageClass object is significant, and is how users can request a particular class. **StorageClass objects cannot be updated once they are created.**

* **Provisioner**

Storage classes have a provisioner that determines what volume plugin is used for provisioning PVs. You are not restricted to specifying the “internal” provisioners listed here (whose names are prefixed with “kubernetes.io” and shipped alongside Kubernetes).

You can also run and specify external provisioners, which are independent programs that follow a specification defined by Kubernetes.

* **Reclaim Policy**

Persistent Volumes that are dynamically created by a storage class will have the reclaim policy specified in the reclaimPolicy field of the class, which can be either Delete or Retain. If no reclaimPolicy is specified when a StorageClass object is created, it will default to Delete.

* **Mount Options**

Persistent Volumes that are dynamically created by a storage class will have the mount options specified in the mountOptions field of the class.

If the volume plugin does not support mount options but mount options are specified, provisioning will fail.

**StatefulSets**

A StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. StatefulSets are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.

- Stable, persistent storage.

- Ordered, graceful provisioning and scaling.

- Ordered, graceful deletion and termination.

- Ordered, automated rolling updates.

Here, stable means persistence across Pod (re)scheduling.

Limitations

* The storage for a given Pod must either be provisioned by a PersistentVolume Provisioner based on the requested storage class, or pre-provisioned by an admin.
* Deleting and/or scaling a StatefulSet down will not delete the volumes associated with the StatefulSet.
* StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. You are responsible for creating this Service.

Example:

apiVersion: v1

kind: Service

metadata:

name: nginx

labels:

app: nginx

spec:

ports:

- port: 80

name: web

clusterIP: None

selector:

app: nginx

---

apiVersion: apps/v1

kind: StatefulSet

metadata:

name: web

spec:

selector:

matchLabels:

app: nginx # has to match .spec.template.metadata.labels

serviceName: "nginx"

replicas: 3 # by default is 1

template:

metadata:

labels:

app: nginx # has to match .spec.selector.matchLabels

spec:

terminationGracePeriodSeconds: 10

containers:

- name: nginx

image: k8s.gcr.io/nginx-slim:0.8

ports:

- containerPort: 80

name: web

volumeMounts:

- name: www

mountPath: /usr/share/nginx/html

volumeClaimTemplates:

- metadata:

name: www

spec:

accessModes: [ "ReadWriteOnce" ]

storageClassName: "my-storage-class"

resources:

requests:

storage: 1Gi

POD Identity

StatefulSet Pods have a unique identity that is comprised of an **ordinal**, a **stable network identity**, and **stable storage**. The identity sticks to the Pod, regardless of which node it’s (re)scheduled on.

**Ordinal**

Each Pod in the StatefulSet will be assigned an integer ordinal, from 0 up through N-1, that is unique over the Set.

**Stable Network ID**

Each Pod in a StatefulSet derives its hostname from the name of the StatefulSet and the ordinal of the Pod. The pattern for the constructed hostname is $(statefulset name)-$(ordinal). The example above will create three Pods named web-0, web-1, web-2.

The Service created above take form: $(service name).$(namespace).svc.cluster.local, where “cluster.local” is the cluster domain.

Also each Pod gets matching DNS subdomain, taking the form: $(podname).$(governing service domain), where the governing service is defined by the spec.serviceName field on the StatefulSet.

**Stable Storage**

In the nginx example above, each Pod will receive a single PersistentVolume with a StorageClass of my-storage-class and 1 Gib of provisioned storage. When a Pod is (re)scheduled onto a node, its volumeMounts mount the PersistentVolumes associated with its PersistentVolume Claims.

Provisioning and Scaling Guarantees

* For a StatefulSet with N replicas, when Pods are being deployed, they are created sequentially, in order from {0..N-1}.
* When Pods are being deleted, they are terminated in reverse order, from {N-1..0}.
* Before a scaling operation is applied to a Pod, all of its predecessors must be Running and Ready.
* Before a Pod is terminated, all of its successors must be completely shutdown.

Pod Management Policies

StatefulSet allows you to relax its ordering guarantees via its .spec.podManagementPolicy field.

* OrderedReady pod management is the default for StatefulSets.
* Parallel pod management tells the StatefulSet controller to launch or terminate all Pods in parallel, and to not wait for Pods to become Running and Ready or completely terminated prior to launching or terminating another Pod.

POD Update Strategies

StatefulSet’s .spec.updateStrategy field allows you to configure and disable automated rolling updates for containers, labels, resource request/limits, and annotations for the Pods in a StatefulSet.

* When a StatefulSet’s .spec.updateStrategy.type is set to **OnDelete**, the StatefulSet controller will not automatically update the Pods in a StatefulSet.

Users must manually delete Pods to cause the controller to create new Pods that reflect modifications made to a StatefulSet’s .spec.template.

* The **RollingUpdate** update strategy implements automated, rolling update for the Pods in a StatefulSet. It is the default strategy when .spec.updateStrategy is left unspecified.

When a StatefulSet’s .spec.updateStrategy.type is set to RollingUpdate, the StatefulSet controller will delete and recreate each Pod in the StatefulSet.

It will proceed in the same order as Pod termination (from the largest ordinal to the smallest), updating each Pod one at a time.

**Partitions**

The **RollingUpdate** update strategy can be partitioned, by specifying a .spec.updateStrategy.rollingUpdate.partition. If a partition is specified, all Pods with an ordinal that is greater than or equal to the partition will be updated when the StatefulSet’s .spec.template is updated. All Pods with an ordinal that is less than the partition will not be updated, and, even if they are deleted, they will be recreated at the previous version.

|  |  |
| --- | --- |
| **Deployment** | **StatefulSet** |
| Will roll out a [Replica Set](https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/) and can scale to facilitate more load. | Ordered, graceful deployment and scaling. |
| Rolling updates are done by updating the PodTemplateSpec of the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. | Ordered, automated rolling updates. |
| Rollback to an earlier Deployment. | No rollback, only deleting and scaling down. |
| Networking Service is separate. | StatefulSets currently require a Headless Service to be responsible for the network identity of the Pods. |

**Job**

A job creates one or more pods and ensures that a specified number of them **successfully terminate**. When a specified number of successful completions is reached, the job itself is complete.

The Job object will start a new Pod if the first pod fails or is deleted (for example due to a node hardware failure or a node reboot).

Normally, when you create a job object, you do not specify .spec.selector. The system defaulting logic adds this field when the job is created. It picks a selector value that will not overlap with any other jobs.

There are three main types of jobs:

1. Non-parallel Jobs

* Normally only one pod is started, unless the pod fails.
* Job is complete as soon as Pod terminates successfully.

For a Non-parallel job, you can leave both .spec.completions and .spec.parallelism unset. When both are unset, both are defaulted to 1.

Example:

apiVersion: batch/v1

kind: Job

metadata:

generateName: countdown-job-

spec:

template:

spec:

containers:

- name: busybox

image: busybox:1

command:

- sh

- -c

- "for i in 9 8 7 6 5 4 3 2 1 0; do echo $i; done"

imagePullPolicy: Always

restartPolicy: OnFailure

1. Parallel Jobs with a fixed completion count:

* Specify a non-zero positive value for .spec.completions.
* The job is complete when there is one successful pod for each value in the range 1 to .spec.completions.

For a Fixed Completion Count job, you should set .spec.completions to the number of completions needed. You can set .spec.parallelism, or leave it unset and it will default to 1.

Example:

apiVersion: batch/v1

kind: Job

metadata:

generateName: countdown-parallelism-

spec:

# Define number of Pods to Complete with Success before declaring Job successful.

completions: 10

# Define number of Pods to run in Parallel

parallelism: 2

template:

spec:

containers:

- name: busybox

image: busybox:1

imagePullPolicy: Always

command:

- /bin/sh

- -c

- "for i in 9 8 7 6 5 4 3 2 1 0; do echo $i; done"

restartPolicy: OnFailure

3. Parallel Jobs with a work queue

The pods must coordinate with themselves or an external service to determine what each should work on.

* Each pod is independently capable of determining whether all its peers are done, thus the entire Job is done.
* When any pod terminates with success, no new pods are created.
* Once any pod has exited with success, no other pod should still be doing any work or writing any output. They should all be in the process of exiting.

For a Work Queue Job, you must leave .spec.completions unset, and set .spec.parallelism to a non-negative integer.

Pod Backoff failure policy

There are situations where you want to fail a Job after some amount of retries due to a logical error in configuration etc. To do so, set .spec.backoffLimit to specify the number of retries before considering a Job as failed.

The back-off limit is set by default to 6.

Failed Pods associated with the Job are recreated by the Job controller with an exponential back-off delay (10s, 20s, 40s …) capped at six minutes.

The back-off count is reset if no new failed Pods appear before the Job’s next status check.

**Cronjob**

One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in Cron format. Cronjobs are useful for creating periodic and recurring tasks, like running backups or sending emails.

Creating Cronjob

apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: countdonw-cronjob

spec:

schedule: "\*/1 \* \* \* \*"

jobTemplate:

spec:

backoffLimit: 3

parallelism: 1

completions: 1

template:

spec:

containers:

- name: busybox

image: busybox:1

command:

- sh

- -c

- "for i in 5 4 3 2 1; do echo \"Countdown: $i\"; done"

restartPolicy: OnFailure

Cronjob Concurrency Policy

It specifies what action need to take for the multiple concurrent job created by this Cronjob. You can specify the policy with .spec.concurrencyPolicy option. Below options are allowed.

* Allow:

The Cronjob will allow running concurrently running Jobs.

* Forbid:

With this, Cronjob will not allow running concurrent Job; if it is time for new Job to run and the previous Job has not finished yet, the Cronjob will skip the new Job run.

* Replace:

If it is time for new Job to run and the previous Job has not finished yet, the Cronjob will replaces the current running job with New Job.

Jobs History Limits

The **.spec.successfulJobsHistoryLimit** and **.spec.failedJobsHistoryLimit** fields are optional. These fields specify how many completed and failed jobs should kept in history.

**NetworkPolicy**

NetworkPolicy resources use labels to select pods and define rules that specify what traffic is allow to the selected pods. Network policies are implemented by the network plugin, so you must be using a networking solution t supports NetworkPolicy.

Isolated and Non-isolated Pods

By default, pods are non-isolated; they accept traffic from any source.

Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy.

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: test-network-policy

namespace: default

spec:

podSelector:

matchLabels:

role: db

policyTypes:

- Ingress

- Egress

ingress:

- from:

- ipBlock:

cidr: 172.17.0.0/16

except:

- 172.17.1.0/24

- namespaceSelector:

matchLabels:

project: myproject

- podSelector:

matchLabels:

role: frontend

ports:

- protocol: TCP

port: 6379

egress:

- to:

- ipBlock:

cidr: 10.0.0.0/24

ports:

- protocol: TCP

port: 5978

Here,

**podSelector**:

Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies. The example policy selects pods with the label “role=db”. An empty podSelector selects all pods in the namespace.

**policyTypes**:

Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. If no policyTypes are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.

**ingress**:

Each NetworkPolicy may include a list of whitelist ingress rules. Each rule allows traffic which matches both the from and ports sections. The example policy contains a single rule, which matches traffic on a single port, from one of 3 sources, the first specified via an ipBlock, the second via a namespaceSelector and the third via a podSelector.

**egress**:

Each NetworkPolicy may include a list of whitelist egress rules. Each rule allows traffic which matches both the to and ports sections. The example policy contains a single rule, which matches traffic on a single port to any destination in 10.0.0.0/24.

So, the example NetworkPolicy:

1. Isolates “role=db” pods in the “default” namespace for both ingress and egress traffic (if they weren’t already isolated)
2. Allows connections to TCP port 6379 of “role=db” pods in the “default” namespace from any pod in the “default” namespace with the label “role=frontend”
3. Allows connections to TCP port 6379 of “role=db” pods in the “default” namespace from any pod in a namespace with the label “project=myproject”
4. Allows connections to TCP port 6379 of “role=db” pods in the “default” namespace from IP addresses that are in CIDR 172.17.0.0/16 and not in 172.17.1.0/24
5. allows connections from any pod in the “default” namespace with the label “role=db” to CIDR 10.0.0.0/24 on TCP port 5978

Default Policies

1. Default deny all ingress traffic

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Ingress

1. Default allow all ingress traffic

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-all

spec:

podSelector: {}

ingress:

- {}

1. Default deny all egress traffic

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Egress

1. Default allow all egress traffic

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: allow-all

spec:

podSelector: {}

egress:

- {}

policyTypes:

- Egress

1. Default deny all ingress and all egress traffic

Example:

apiVersion: networking.k8s.io/v1

kind: NetworkPolicy

metadata:

name: default-deny

spec:

podSelector: {}

policyTypes:

- Ingress

- Egress

**API Server AdmissionControl**

**Journalctl**

**Helm**

To deploy an application, we use different Kubernetes manifests, such as Deployments, Services, Volume Claims, Ingress, etc. Sometimes, it can be tiresome 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 refer to as Chart.

[Helm](https://github.com/kubernetes/helm) is a package manager (analogous to **yum** and **apt**) for Kubernetes, which can install/update/delete those Charts in the Kubernetes cluster. Helm has 2 components,

* A Client called **helm**, which runs on your user’s workstation.
* A Server called **tiller**, which run inside Kubernetes Cluster.

A client **helm** connect to the server **tiller** to manage the Charts.

**Node Resource Reservation**

Pods can consume all the available capacity on a node by default. Unless resources are set aside for these system daemons, pods and system daemons compete for resources and lead to resource starvation issues on the node.

Allocatable on a Kubernetes node is define as the amount of compute resources that are available for pods.

Kube Reserved

Kubelet Flag: --kube-reserved=[cpu=100m][,][memory=100Mi][,][ephemeral-storage=1Gi]

Kubelet Flag: --kube-reserved-cgroup=

System Reserved

Kubelet Flag: --system-reserved=[cpu=100mi][,][memory=100Mi][,][ephemeral-storage=1Gi]

Kubelet Flag: --system-reserved-cgroup=

Eviction Thresholds

Kubelet Flag: --eviction-hard=[memory.available<500Mi]

Example

Node has 32Gi of memory, 16 CPUs and 100Gi of Storage

--kube-reserved is set to cpu=1,memory=2Gi,ephemeral-storage=1Gi

--system-reserved is set to cpu=500m,memory=1Gi,ephemeral-storage=1Gi

--eviction-hard is set to memory.available<500Mi,nodefs.available<10%

**Static Pods**

Static Pods are directly manage by Kubelet daemon on specific Node. It does not have an associated ReplicationController, DaemonSet, Deployment, and ReplicaSet.

Kubelet Daemon itself watches the Static Pod and Restart in case it crashes. The Static Pods are always bound to One Kubelet daemon and always run on same node.

Kubelet automatically tries to create Mirror Pod on the Kubernetes API server for each Static Pod, this enable to view Static Pods on API server, but cannot Control this Static Pods.

Static Pods can be created in 2 ways by specifying below parameters to Kubelet.

* --pod-manifest-path=<the directory>
* --manifest-url=<URL>

Kubelet periodically scans the directory and creates/deletes static pods as yaml/json files appear/disappear there.

**Auditing**

Kube-apiserver performs auditing. Each request on each stage of its execution generates an event, which is then pre-processed according to a certain policy and written to a backend. Each request can be recorded with an associated “stage”. The known stages are:

* RequestReceived

The stage for events generated as soon as the audit handler receives the request.

* ResponseStarted

Once the response headers are sent, but before the response body is sent.

* ResponseComplete

The response body has been completed and no more bytes will be sent.

* Panic

Events generated when a panic occurred.

Audit Policy

Audit policy defines rules about what events should be recorded and what data they should include. When an event is processed, it’s compared against the list of rules in order. The first matching rule sets the “audit level” of the event. The known audit levels are:

* None

Do not log events that match this rule.

* Metadata

Log request metadata (requesting user, timestamp, resource, verb, etc.) but not request or response body.

* Request

Log event metadata and request body but not response body. This does not apply for non-resource requests.

* RequestResponse

Log event metadata, request and response bodies. This does not apply for non-resource requests.

You can pass a file with the policy to **kube-apiserver** using the --audit-policy-file flag.

Example:

apiVersion: audit.k8s.io/v1beta1 # This is required.

kind: Policy

# Don't generate audit events for all requests in RequestReceived stage.

omitStages:

- "RequestReceived"

rules:

# Log pod changes at RequestResponse level

- level: RequestResponse

resources:

- group: ""

# Resource "pods" doesn't match requests to any subresource of pods,

# which is consistent with the RBAC policy.

resources: ["pods"]

# Log "pods/log", "pods/status" at Metadata level

- level: Metadata

resources:

- group: ""

resources: ["pods/log", "pods/status"]

# Don't log requests to a configmap called "controller-leader"

- level: None

resources:

- group: ""

resources: ["configmaps"]

resourceNames: ["controller-leader"]

# Log all other resources in core and extensions at the Request level.

- level: Request

resources:

- group: "" # core API group

- group: "extensions" # Version of group should NOT be included.

Audit Backend

Audit backends persist audit events to an external storage. Kube-apiserver out of the box provides two backends:

* **Log backend**, which writes events to a disk
* **Webhook backend**, which sends events to an external API

Log backend writes audit events to a file in JSON format. You can configure log audit backend using the following kube-apiserver flags:

--audit-log-path specifies the log file path that log backend uses to write audit events.

--audit-log-maxage defined the maximum number of days to retain old audit log files

--audit-log-maxbackup defines the maximum number of audit log files to retain

--audit-log-maxsize defines the maximum size in megabytes of the audit log file before it gets rotated

**Container Termination Policy**

Kubernetes retrieves termination messages from the termination message file specified in the “terminationMessagePath” field of a Container, which as a default value of /dev/termination-log.

Moreover, users can set the “terminationMessagePolicy” field to,

1. File

Means the termination messages are retrieve only from the termination message file.

1. FallbackToLogsOnError

Use the last chunk of container log output if the termination message file is empty and the container exited with an error. The log output is limited to 2048 bytes or 80 lines, whichever is smaller.

**Monitoring Compute, Storage, and Network Resources**

**Heapster** is a cluster-wide aggregator of monitoring and event data. The Heapster pod discovers all nodes in the cluster and queries usage information from the nodes’ Kubelets, the on-machine Kubernetes agent. The Kubelet itself fetches the data from cAdvisor. This data is then push to a configurable backend for storage (Influx) and visualization (Grafana).

Grafana 🡨 Influx 🡨 Hepaster 🡨 Kubelet 🡨 cAdvisor

**cAdvisor** is an open source container resource usage and performance analysis agent. In Kubernetes, cAdvisor is integrated into the Kubelet binary. cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics.

**InfluxDB** exposes an easy to use API to write and fetch time series data. Heapster is setup to use this storage backend by default on most Kubernetes clusters.

**Grafana** container serves UI that provides an easy to configure dashboard interface.

**POD Preset**

A POD Preset is allowing you to inject additional information into the POD specification at the time of Pod creation. You can use a PodPreset object to inject information like secrets, volume mounts, and environment variables etc into pods at creation time.

How It Works

1. Retrieve all PodPreset available for use.
2. Check if the label selectors of any PodPreset matches the labels on the pod being created.
3. Attempt to merge the various resources defined by the PodPreset into the Pod being created.
4. On error, throw an event documenting the merge error on the pod, and create the pod without any injected resources from the PodPreset.
5. Annotate the resulting modified Pod spec to indicate that a PodPreset has modified it. The annotation is of the form,

podpreset.admission.kubernetes.io/podpreset-<pod-preset name>: "<resource version>".

Disable Preset for a Specific POD

Add below Annotation in the Pod Spec,

podpreset.admission.kubernetes.io/exclude: “true”

**Kubectl Patch**

The kubectl patch command has a type parameter that you can set to one of these values:

* json
* merge

With a JSON merge patch, if you want to update a list, you have to specify the entire new list. Moreover, the new list completely replaces the existing list.

* strategic

With a strategic merge patch, a list is either replaced or merged depending on its patch strategy.

Example,

$ cat patch.yaml

spec:

template:

spec:

containers:

- name: patch-demo-ctr-3

image: gcr.io/google-samples/node-hello:1.0

$ kubectl patch deployment patch-demo --type merge --patch "$(cat patch.yaml)"

**Delete Deployment/Statefulset without Pod being Deleted**

You can use “--cascade=false” option while deleting the Deployment or StatefulSets object.

$ kubectl delete deployment httpd-dp --cascade=false

**Horizontal POD Autoscaling**

The Horizontal Pod Autoscaler is implemented as a control loop, with a period controlled by the controller manager’s --horizontal-pod-autoscaler-sync-period flag (with a default value of 30 seconds).

During each period, the controller manager queries the resource utilization against the metrics server. The HorizontalPodAutoscaler normally fetches metrics from a series of aggregated APIs (metrics.k8s.io, custom.metrics.k8s.io, and external.metrics.k8s.io).

**Fetching metrics from Heapster is deprecated as of Kubernetes 1.11.**

The current stable version, which only includes support for CPU autoscaling, can be found in the autoscaling/v1 API version.

The beta version, which includes support for scaling on memory and custom metrics, can be found in autoscaling/v2beta1.

Horizontal Autoscaling Delay

--horizontal-pod-autoscaler-downscale-delay: The value for this option is a duration that specifies how long the autoscaler has to wait before another downscale operation can be performed after the current one has completed. The default value is 5 minutes (5m0s)

--horizontal-pod-autoscaler-upscale-delay: The value for this option is a duration that specifies how long the autoscaler has to wait before another upscale operation can be performed after the current one has completed. The default value is 3 minutes (3m0s).

**Pod Disruption**

The Disruptions can broadly divided into 2 parts,

1. Involuntary Disruptions

* Hardware Failure of the Physical Machine backing the Node
* Deletion of VM backing node by Cluster Administrator
* Kernel Panic of the Node
* Pod eviction due to resource constraint

1. Voluntary Disruptions

* Deleting the Deployment or other Controller that manages the Pod
* Updating the Deployment Pod’s Template causing restart of Pod
* Directly deleting Pod

Protecting an Application with Voluntary Disruption

An application owner can create a PodDisruptionBudget (PDB) object for each application. A PDB limits the number of simultaneous disruption of Pods in replicated application.

For Example, a Quorum based application would like to ensure that the number of replicas running is never fall down below the number needed by Quorum.

PDB cannot prevent Involuntary Disruptions.

Specifying PodDisruptionBudget

A PodDisruptionBudget has 3 fields,

* .spec.selector

A label selector to specify the set of Pods to which it applies

* .spec.minAvailable

This can be either Absolute number or a Percentage.

* .spec.maxUnavailable

This can be either Absolute number or a Percentage.

You can only specify one of minAvailable or maxUnavailable in a single PodDisruptionBudget.

Example,

apiVersion: policy/v1beta1

kind: PodDisruptionBudget

metadata:

name: zk-pdb

spec:

minAvailable: 2

selector:

matchLabels:

app: zookeeper

**Troubleshooting**

1. A POD cannot reach itself via Service IP

🡪 Make sure you have hairpin mode set to either “hairping-veth” or “promiscuous-bridge” in Kubelet configuration.

1. Is Kube-Proxy writing iptable Rules?

🡪Verify it via below command in different mode,

1. Userspace

$ iptables-save | grep -i ‘<ip-address>’

1. IPtables

$ iptables-save | grep -i ‘<ip-address>’

1. IPVS

$ ipvsadm –ln

**CKA Exam Questions to Examiner**

1. Should I be able to install any Packages on the Cluster Nodes? Actually, I want to install “bash-completion” package.

$ source <(kubectl completion bash)

**Notes:**

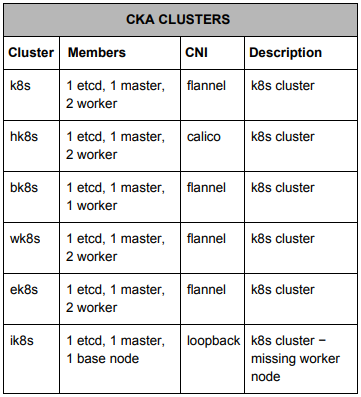
1. Deployment does not have Restart Policy “Always” it only supports “OnFailure”, and “Never” policy.

Notes:

1. Exam servers will be Ubuntu 16.
2. Website allowed to accessed during exam is <http://kubernetes.io>
3. Weights of the questions during exam,

* Application Lifecycle Management **8%**
* Installation, Configuration & Validation **12%**
* Core Concepts **19%**
* Networking **11%**
* Scheduling **5%**
* Security **12%**
* Cluster Maintenance **11%**
* Logging / Monitoring **5%**
* Storage **7%**
* Troubleshooting **10%**

1. Exam Cluster



1. Must have Chrome Browser
2. Keep Passport with You for Identification
3. Portal URL <https://training.cncf.io/portal>
4. “Linux Foundation” as Exam Sponsor and “CKA” as Exam
5. Systemd Creation and Management
6. Used terminal multiplexer (e.g, tmux)
7. Work on vi editor and its shortcuts.
8. Browser Based Simulator

<https://github.com/liftoff/GateOne/downloads>

1. CKA Curriculum

<https://github.com/walidshaari/Kubernetes-Certified-Administrator/blob/master/README.md>

1. Kubernetes Hard Way

<https://github.com/kelseyhightower/kubernetes-the-hard-way/blob/master/README.md>

1. Kubernetes Cheat Sheet

<https://github.com/nkuba/k8s-admin-helper>

1. Exam Simulator

<https://github.com/smartbit/cka-practice-environment>