Kubernetes

101378029139

LimitRange is for managing constraints at a pod and container level within the project. An individual Pod or Container that requests resources outside of these LimitRange constraints will be rejected, whereas a ResourceQuota only applies to all of the namespace/project's objects in aggregate.

##### [What is Kubernetes?](https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/)

Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

[**The Kubernetes API**](https://kubernetes.io/docs/concepts/overview/kubernetes-api/)

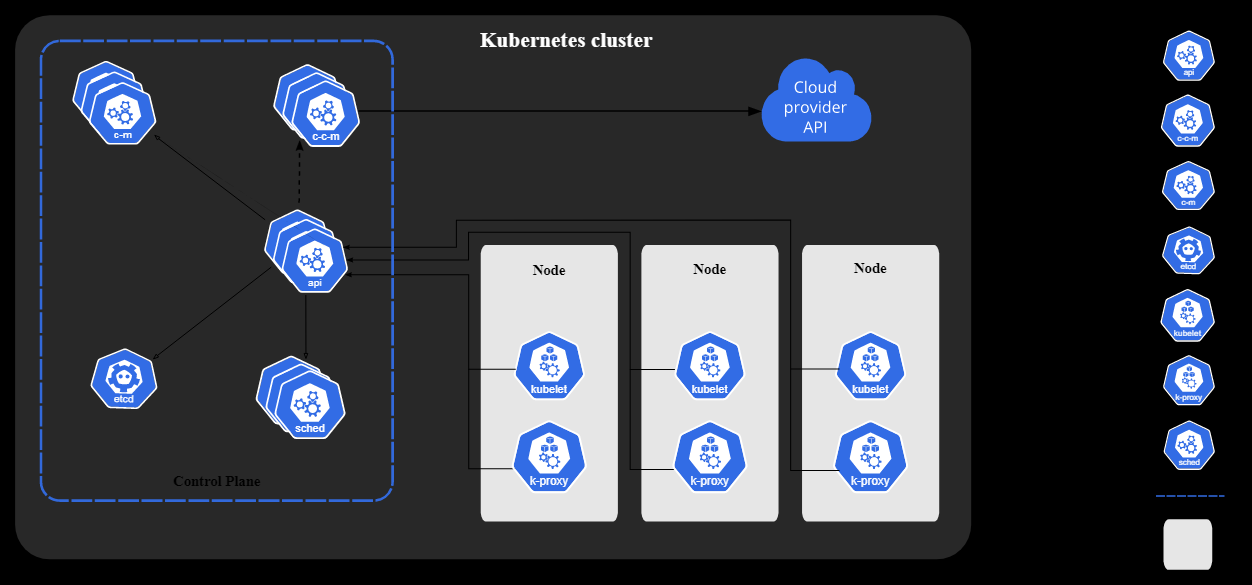
The Kubernetes API lets you query and manipulate the state of objects in Kubernetes. The core of Kubernetes' control plane is the API server and the HTTP API that it exposes. Users, the different parts of your cluster, and external components all communicate with one another through the API server.

**Kubernetes Components**

When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.



**Control Plane Components**

kube-apiserver - The API server is how the underlying Kubernetes APIs are exposed. This component provides the interaction for management tools, such as kubectl or the Kubernetes dashboard.

etcd - To maintain the state of your Kubernetes cluster and configuration, the highly available etcd is a key value store within Kubernetes.

kube-scheduler - Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

**Service**

Pod has a unique dynamic ip address.

Service is a system not processor. It provides the connectivity means exposes our app pod other app or outside the cluster.

**Service** enables network access to a set of Pods in Kubernetes. Services select Pods based on their labels. When a network request is made to the service, it selects all Pods in the cluster matching the service's selector, chooses one of them, and forwards the network request to it. In this lab you can explore different types of services.

Kubernetes Service Types allow you to specify what kind of Service you want. The default is Cluster IP.

* **Cluster IP:** Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default Service Type.
* [**NodePort**](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport)**:** Exposes the Service on each Node's IP at a static port (the NodePort). A Cluster IP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* [**LoadBalancer**](https://kubernetes.io/docs/concepts/services-networking/service/#loadbalancer)**:** Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.
* [**ExternalName**](https://kubernetes.io/docs/concepts/services-networking/service/#externalname)**:** Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record

**What is Kubernetes Ingress?**

Kubernetes Ingress is an API object that provides routing rules to manage external users' access to the services in a Kubernetes cluster, typically via HTTPS/HTTP.

With Ingress, you can easily set up rules for routing traffic without creating a bunch of Load Balancers or exposing each service on the node. This makes it the best option to use in production environments.

**What is the Ingress Controller?**

If Kubernetes Ingress is the API object that provides routing rules to manage external access to services, Ingress Controller is the actual implementation of the Ingress API. The Ingress Controller is usually a load balancer for routing external traffic to your Kubernetes cluster and is responsible for L4-L7 Network Services.

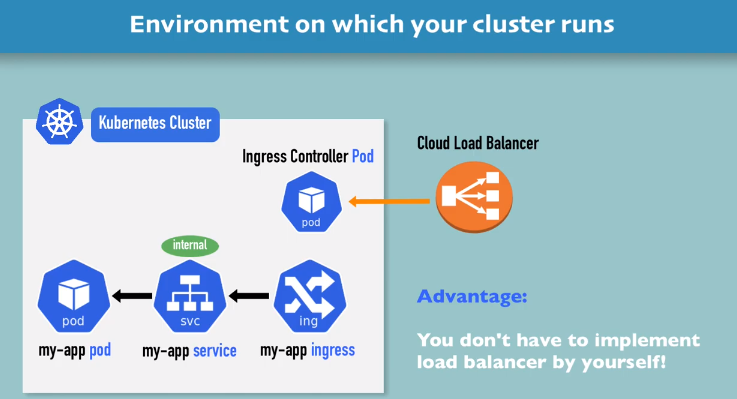
-Evaluates all the rules

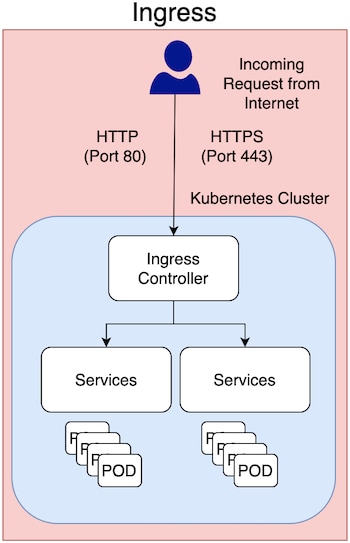
-Manages redirections

-Entry point to cluster

-Many third party implementations

-K8s ingress controller





**Replication Controllers:**

* A replication controller (RC) is a supervisor for long-running pods. An RC will launch a specified number of pods called replicas and makes sure that they keep running, for example when a node fails or something inside of a pod, that is, in one of its containers goes wrong.
* If there are too many pods, the ReplicationController terminates the extra pods.
* Unlike manually created pods, the pods maintained by a ReplicationController are automatically replaced if they fail, are deleted, or are terminated.

**Namespace:** A namespace is used to categorize objects in a cluster.

Namespaces can be used to segregate your cluster for multiple dev teams for discrete purposes (e.g. dev, staging, and prod).

There are three different namespaces:

1.default

2. kube-public

3. kube-system

you can verify by using the following command to show the default namespaces in the cluster.

$kubectl get ns

**Create the Namespace**

$kubectl create ns devteam

**By Using Yaml file**

apiVersion: v1

kind: Namespace

metadata:

name: devteam-namespace

$kubectl create -f <path/devteam-namespace.yaml>

**list out the all namespaces in a cluster**

$kubectl get ns

**Delete the Namespace**

**$**kubectl delete ns devteam-namespace

**default:** By default a Kubernetes cluster will instantiate a default namespace when provisioning the cluster to hold the default set of Pods, Services, and Deployments used by the cluster. If you do not specify namespace name Kuberentes resources will be deployed into default namespace.

**kube-system:** the 'kube-system' namespace is for objects generated by the Kubernetes. Usually this has pods (containerized applications) like kube-dns, kube-proxy, kubernetes-dashboard. The following command will show all the pods in a kube-system namespace. It is recommended not to disturb any of the setup in this namespace.

**kube-public:** the kube-public namespace is automatically created with the cluster (just like default and kube-system). This content is accessible for public unauthenticated users and is mainly used for system containers.

**Pod:**

A Pod is a collection of application containers running on the same node.

Pods, not containers, are the smallest deployable artifacts in a Kubernetes cluster, which means all of the containers will be deployed on the same node (usually there is only one container per pod).

Pods are faster to start, can self heal, are easy to scale & update when compared to virtual machines ("VMs").

Usually, containers are tasked with one job (e.g. as a http server, as a database, as a proxy).

**Yaml file**

apiVersion: v1

kind: pod

metadata:

name: http-service (pod name)

namespace: devteam

labels:

run: my-nginx

spec:

containers:

* name: my-nginx (Container name)

image: nginx

ports:

- containerPort: 80

$Kubectl create -f <path/devteam-pod.yaml>

* The pod manifest will be submitted to the Kubernetes API server.
* The Kubernetes system will then schedule that pod to run on a healthy node in the cluster, where it will be monitored by the kubelet daemon process.

**List out the pods running in a namespace**

**$**kubectl -n devteam get pods

**Describe the Pod Details**

**$**kubectl -n devteam get po http-service -o wide

**$**kubectl -n devteam get po http-service

**Running Commands in a Container with exec**

**$**kubectl -n devteam exec -it http-service bash

**Logs of the pod**

**$**kubectl -n devteam logs -f http-service

**Edit the Pod**

**$**kubectl -n devteam edit po http-service

**Delete the Pod**

**$**kubectl -n devteam delete po http-service

**ConfigMaps**

ConfigMaps are used to provide configuration information (like environment variables) for containarized workloads, it is a Kubernetes object that defines small filesystem.

apiVersion: v1

kind: ConfigMap

metadata:

name: test-cm

data:

# Configuration values can be set as key-value properties

database: mongodb

database\_uri: mongodb://localhost:27017

# Or set as complete file contents (even JSON!)

keys: |

image.public.key=771

rsa.public.key=42

$kubectl -n devteam apply -f <configmap.yaml>

$kubectl -n devteam get cm

$kubectl -n devteam describe cm test-cm

**Access the ConfigMaps from pods**

apiVersion: v1

kind: pod

metadata:

name: env-pod

spec:

containers:

- envFrom:

- configMapRef:

name: test-cm

name: test-container

image: nginx

ports:

- containerport: 80

$kubectl -n devteam apply -f <path env-pod.yaml>

$kubectl -n devteam exec -it env-pod

$Kubectl -n devteam edit cm test-cm

$kubectl -n devteam delete cm test-cm

The basic thing is that the ConfigMap is combined with the Pod right before it is run, which means that the container image and the pod definition itself can be reused across many apps by just changing the ConfigMap that is used.

**Secrets**

Secrets are objects in Kubernetes, that contain sensitive information such as credentials and tokens. They are stored in etcd.

These are accessible by the Kubernetes API server, and it can be mounted as files into a pod. The same secret can be mounted into multiple pods.

**Create the Secrets from Files using Kubectl**

$ mkdir k8s-secrets

$ cd k8s-secrets

$ echo -n 'test' > ./username.txt

$ echo -n 'jiki893kdjnsaasdsa' > ./password.txt

$ kubectl -n devteam create secret generic db --from-file=c:/Users/PhotonUser/Documents/k8s-secrets/username.txt --from-file=c:/Users/PhotonUser/Documents/k8s-secrets/password.txt

$kubectl -n devteam get secrets

$kubectl -n devteam describe secrets db

$ kubectl edit secrets db -n devteam

**Annotations**

Just like labels, annotations can be used to enrich your metadata with key value pairs, which most applications may use to process in additional parameters or customized settings**.**

"metadata": {

"annotations": {

"key1" : "value1",

"key2" : "value2"

}

}

**ServiceAccount**

A Service Account is just like any other user with respect to the cluster.

Every pod runs in the cluster with the identity of the service account.

If no service account is provided the default service account is used. The default service account is created when the namespace is created.

If the pod needs to access any of the Kubernetes api then it needs to have higher privileges in the cluster level.

Just like a users privileges are determined by the role which the role binding binds them to. The service account can also be assigned a role using its own role binding.

**Serviceaccount.yml**

apiVersion: v1

kind: ServiceAccount

metadata:

name: account

$kubectl create -f serviceaccount.yaml

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

labels:

run: ubuntu

name: ubuntu

spec:

replicas: 1

selector:

matchLabels:

run: ubuntu

template:

metadata:

labels:

run: ubuntu

spec:

serviceAccountName: account

containers:

- image: ubuntu

imagePullPolicy: Always

command: ["sleep"]

args: ["30d"]

name: ubuntu

restartPolicy: Always

securityContext:

runAsUser: 0

$kubectl create -f <path/ubuntu.yaml>

$kubectl get pods

$ kubectl exec -it ubuntu-7b94cb6fd6-p69tg bash

**Without Rolebinding**

* serviceAccounts can be added when required. Each pod is associated with exactly one serviceAccount but multiple pods can use the same serviceaccount.
* A pod can only use a serviceaccount from the same namespace.
* You can assign a serviceaccount to a pod by specifying the account’s name in the pod manifest. If you don’t assign it explicitly the pod will use the default serviceaccount in the namespace
* The default permissions for a ServiceAccount dont allow it to list or modify any resources. The default Service- Account isnt allowed to view cluster state let alone modify it in any way.
* By default, the default serviceAccount in a namespace has no permissions other than those of an unauthenticated user.
* Therefore pods by default can’t even view cluster state. Its up to you to grant them appropriate permissions to do that.

**Here is where the serviceaccount token is created**

root@ubuntu-7b94cb6fd6-p69tg:/# TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)

root@ubuntu-7b94cb6fd6-p69tg:/# echo $TOKEN

**This is the default service in the default namespace for kubernetes API**

root@ubuntu-7b94cb6fd6-p69tg:/# API\_SVC="https://kubernetes.default" ns=default

root@ubuntu-7b94cb6fd6-p69tg:/# echo $API\_SVC

**This is setting the url to get all pods in the namespace**

root@ubuntu-7b94cb6fd6-p69tg:/# ns=default

root@ubuntu-7b94cb6fd6-p69tg:/# API\_POD\_URL="$API\_SVC"/api/v1/namespaces/default/pods/

root@ubuntu-7b94cb6fd6-p69tg:/# curl -s -X GET --header "Authorization: Bearer $TOKEN" --insecure $API\_POD\_URL

* As can be seen above "pods is forbidden: User "system:serviceaccount:default:account" cannot list resource "pods" in API group "" in the namespace "default"",

**With Rolebinding**

Save the following configuration code in a yaml file and Deploy the Rolebinding configuration using the following command.

apiVersion: rbac.authorization.k8s.io/v1

kind: RoleBinding

metadata:

name: account

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: edit

subjects:

- kind: ServiceAccount

name: account

namespace: default

$kubectl create -f <rolebinding.yml>

* Giving all your serviceAccounts the clusteradmin clusterrole is a bad idea its best to give everyone only the permissions they need to do their job and not a single permission more.
* It’s a good idea to create a specific serviceAccount for each pod and then associate it with a tailor-made role or a clusterrole through a rolebinding.
* If one of your pods only needs to read pods while the other also needs to modify them then create two different serviceaccounts and make those pods use them by specifying the serviceaccountName property in the pod spec.
* Retry the commands in the pods was used in previous steps, you will see the commands are successful
* Exec into the contaier using the following command, Replace podname with yours

kubectl exec -it ubuntu-7b94cb6fd6-p69tg bash

* You can also try to get all services in default namespace

API\_SVC\_URL="$API\_SVC"/api/v1/namespaces/"$ns"/services/

curl -s -X GET --header "Authorization: Bearer $TOKEN" --insecure $API\_SVC\_URL

API\_DEP\_URL="$API\_SVC"/api/v1/namespaces/"$ns"/deployments/

curl -s -X GET --header "Authorization: Bearer $TOKEN" --insecure $API\_DEP\_URL

bashroot@ubuntu-7b94cb6fd6-7v2x6:/# apt-get update && apt-get install -y curl

root@ubuntu-7b94cb6fd6-7v2x6:/# TOKEN=$(cat /run/secrets/kubernetes.io/serviceaccount/token)

root@ubuntu-7b94cb6fd6-7v2x6:/# echo $TOKEN

root@ubuntu-7b94cb6fd6-7v2x6:/# API\_SVC="https://kubernetes.default" ns=default

root@ubuntu-7b94cb6fd6-7v2x6:/# echo $API\_SVC

* Run the following commands

root@ubuntu-7b94cb6fd6-7v2x6:/# ns=default

root@ubuntu-7b94cb6fd6-7v2x6:/# API\_POD\_URL="$API\_SVC"/api/v1/namespaces/default/pods/

root@ubuntu-7b94cb6fd6-7v2x6:/# API\_POD\_URL="$API\_SVC"/api/v1/namespaces/default/pods/

root@ubuntu-7b94cb6fd6-7v2x6:/# echo $API\_POD\_URL

root@ubuntu-7b94cb6fd6-7v2x6:/# curl -s -X GET --header "Authorization: Bearer $TOKEN" --insecure $API\_POD\_URL

* Now you able to list the resource service.

The following commands should fail since you are limited to this namespace by the rolebinding and not in kube-system namespace

root@ubuntu-7b94cb6fd6-7v2x6:/# ns=kube-system

root@ubuntu-7b94cb6fd6-7v2x6:/#API\_POD\_URL="$API\_SVC"/api/v1/namespaces/kube-system/pods/

root@ubuntu-7b94cb6fd6-7v2x6:/#API\_POD\_URL="$API\_SVC"/api/v1/namespaces/kube-system/pods/

root@ubuntu-7b94cb6fd6-7v2x6:/# curl -s -X GET --header "Authorization: Bearer $TOKEN" --insecure $API\_POD\_URL

**Security Context**

* A Security Context defines the operating system security settings (uid, gid, capabilities, SELinux role, etc applied to a container).
* Both pods and containers can have securityContext field in their spec.
* If both are set then escalation privileges field determines whether the container privileges are escalated or not.
* The value of this field determines the privileges on the host vm, of the user that is running the container/pods
* By default most containers start as root user, but this is not good for production systems where a malicious actor can obtain access to underlying vm. There is a difference in the root user in the container and root user on the host machine, the root user can only become complete root user on the host if privileged is set to true or escalation of privileges is set to true.

**ubuntu-privilage.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

labels:

run: ubuntu

name: ubuntuprivileged

spec:

replicas: 1

selector:

matchLabels:

run: ubuntu

template:

metadata:

labels:

run: ubuntu

spec:

containers:

- image: ubuntu

imagePullPolicy: Always

command: ["sleep"]

args: ["30d"]

name: ubuntu

volumeMounts:

- name: all

mountPath: /host

securityContext:

runAsUser: 0

privileged: **true**

volumes:

- name: all

hostPath:

path: /

restartPolicy: Always

securityContext:

runAsUser: 0

$kubectl create -f ubuntu-privilage.yaml

$kubectl get pods

* The **runAsUser** field defines which users a container can run as. Most commonly, it is used to prevent pods from running as the root user.

Get a shell into the container using the following command

kubectl exec -it ubuntu0wvolprivileged-7475f6df7c-4lbqf bash

root@ubuntuprivileged-7475f6df7c-j5vnd:/# id

**Output:**

uid=0(root) gid=0(root) groups=0(root)

Now you will see that gid is 0 which is same as runAsGroup specified in a yaml configuration file, the primary group ID of the containers will be root(0).

root@ubuntuprivileged-7475f6df7c-j5vnd:/# ls -ltr /host/proc/1/map\_files

**ubuntunosec.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

labels:

run: ubuntu

name: ubuntunosecuritycontext

spec:

replicas: 1

selector:

matchLabels:

run: ubuntu

template:

metadata:

labels:

run: ubuntu

spec:

containers:

- image: ubuntu

imagePullPolicy: Always

command: ["sleep"]

args: ["30d"]

name: ubuntu

volumeMounts:

- name: all

mountPath: /host

volumes:

- name: all

hostPath:

path: /

restartPolicy: Always

$kubectl create -f ubuntunosec.yaml

$kubectl get pods

* Get a shell into the container using the following command.

kubectl exec -it ubuntunosecuritycontext-7ddff8d96c-ppxlk bash

root@ubuntunosecuritycontext-7ddff8d96c-ppxlk:/# id

root@ubuntunosecuritycontext-7ddff8d96c-ppxlk:/# ls -ltr /host/proc/1/map\_files

**ubuntuuser-1000.yaml**

apiVersion: extensions/v1beta1

kind: Deployment

metadata:

labels:

run: ubuntu

name: ubuntuwuser1000

spec:

replicas: 1

selector:

matchLabels:

run: ubuntu

template:

metadata:

labels:

run: ubuntu

spec:

containers:

- image: ubuntu

imagePullPolicy: Always

command: ["sleep"]

args: ["30d"]

name: ubuntu

volumeMounts:

- name: all

mountPath: /host

volumes:

- name: all

hostPath:

path: /

restartPolicy: Always

securityContext:

runAsUser: 1000

* In the configuration file, the runAsUser field specifies that for any Containers in the Pod, all processes run with user ID 1000.

$kubectl create -f ubuntuuser-1000.yaml

$kubectl get pods

kubectl exec -it ubuntuwuser1000-7798964455-s9bsh bash

@ubuntuwuser1000-7798964455-s9bsh:/$ id

**Output:**

uid=1000 gid=0(root) groups=0(root)

Run the **ls -ltr /host/proc/1/map\_files** command

**Liveness Probes**

* The kubelet uses liveness probes to know when to restart a container.
* For example, liveness probes could catch a deadlock, where an application is running, but unable to make progress. Restarting a container in such a state can help to make the application more available despite bugs.
* Many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted.
* Kubernetes provides liveness probes to detect and remedy such situations.

apiVersion: v1

kind: Pod

metadata:

name: my-liveness-pod

spec:

containers:

- name: myapp-container

image: busybox

command: ['sh', '-c', "touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 600"]

livenessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

$kubectl -n devteam create -f liveness-probe.yml

$kubectl -n devteam get pods

$ kubectl -n devteam describe pod my-liveness-pod

**Readiness probes**

* The kubelet uses readiness probes to know when a container is ready to start accepting traffic.
* A pod is considered ready when all of its containers are ready.
* One use of this signal is to control which Pods are used as backends for Services. When a Pod is not ready, it is removed from Service load balancers.
* Kubernetes provides readiness probes to detect and mitigate these situations. A pod with containers reporting that they are not ready does not receive traffic through Kubernetes Services.

**Readiness-probe.yml**

apiVersion: v1

kind: Pod

metadata:

name: my-rediness-pod

labels:

app: readytest

spec:

containers:

- name: myapp-container

image: nginx

readinessProbe:

exec:

command:

- cat

- /tmp/healthy

initialDelaySeconds: 5

periodSeconds: 5

$kubectl -n devteam create -f Readiness-probe.yaml

$kubectl -n debteam get po

* Let's create a sample service using the following configuration code.

Note: Make sure choose selector as **app=readytest** so that service will communicate to the Readiness pod.

apiVersion: v1

kind: Service

metadata:

name: test-svc

spec:

selector:

app: readytest

ports:

- port: 8088

targetPort: 80

type: ClusterIP

$kubectl -n devteam create -f <test-svc.yml>

$kubectl -n devteam get svc

$ kubectl -n devteam exec -it my-rediness-pod sh

* Now open a new gitbash terminal, describe the service using the following command. You should see a endpoint without ip address and port.

$ kubectl -n devteam describe svc test-svc

* Now go back to previous Powershell/gitbash window which has readiness-probe pod, create a directory using **touch /tmp/healthy** inside a pod. wait for 5 seconds describe the service, you should see a endpoint with ip adress and port, shown below.

$ kubectl -n devteam describe svc test-svc

Before that, you needs to be install **wget** to access the service, use the following command to install wget inside a pod.

root@my-rediness-pod:/# apt-get update && apt-get install wget -y

* Use the service name with port to verify the both pod and service are communicating.

root@my-rediness-pod:/# wget -q --timeout=5 test-svc:8088 -O -

**Limit Range**

* Assigning a memory request and a memory limit or a cpu request and a cpu limit to a container is called LimitRange.
* A Container is guaranteed to have as much memory as it requests, but is not allowed to use more memory than its limit.

Note: Make sure you have the metrics server installed in your cluster before applying LimitRanges.

* Create a LimitRange to a namespace then all the pods that exists in that namespace will get applied by the LimitRange

$kubectl create ns devteam

**limitrange.yaml**

apiVersion: v1

kind: LimitRange

metadata:

name: cpu-memory

namespace: devteam

spec:

limits:

- default:

cpu: 250m

memory: 300Mi

defaultRequest:

cpu: 100m

memory: 200Mi

type: Container

$kubectl -n devteam create -f <limitrange.yml

**Deploy a sample pod and service**

* Let's deploy a pod and service that creates a single container to demonstrate how default values are applied to each pod. Use the following command in Powershell.

kubectl -n devteam run php-apache --image=k8s.gcr.io/hpa-example --expose --port=80

kubectl -n devteam get pods

kubectl -n devteam get services

**Get the configuration of the pod**

kubectl -n devteam get pod php-apache -o yaml

kubectl top pods -n devteam

kubectl top nodes

**ResourceQuota**

* In Kubernetes, ResourceQuota is used to limit the resources per namespace when multiple users sharing the cluster. We will apply the ResourceQuota for the compute resources.

resource-quota.yaml

apiVersion: v1

kind: ResourceQuota

metadata:

name: resourcequota-name

namespace: devteam-test

spec:

hard:

requests.cpu: "1"

requests.memory: 1Gi

limits.cpu: "2"

limits.memory: 2Gi

$kubectl create -f resource-quota.yaml

Now use the **kubectl -n devteam-test get resourcequota resourcequota-name -o yaml** command to get detailed information about the ResourceQuota.

**Create a Pod**

apiVersion: v1

kind: Pod

metadata:

name: nginx-cpu-memory

namespace: devteam-test

spec:

containers:

- name: nginx-cpu-memory-quota

image: nginx

resources:

limits:

memory: "700Mi"

cpu: "700m"

requests:

memory: "500Mi"

cpu: "300m"

$kubectl create -f nginx-memory.yaml

$kubectl -n devteam-test get pods

* Get more detailed information about the ResourceQuota, Use the following command in Powershell.

kubectl -n devteam-test get resourcequota resourcequota-name -o yaml

* The output shows how much of the quota has been used along with the quota limits.

**Create Another Pod**

**Create another pod by specifying the memory request more than its ResuorceQuota request.**

$kubectl create ns resourcequota

apiVersion: v1

kind: Pod

metadata:

name: redis-cpu-memory

namespace: resourcequota

spec:

containers:

- name: redis-cpu-memory-quota

image: redis

resources:

limits:

memory: "1Gi"

cpu: "900m"

requests:

memory: "600Mi"

cpu: "500m"

kubectl create -f redis-cpu-memory.yaml

**Create Another Pod:**

apiVersion: v1

kind: Pod

metadata:

name: redis-cpu-memory

namespace: devteam-test

spec:

containers:

- name: redis-cpu-memory-quota

image: redis

resources:

limits:

memory: "1Gi"

cpu: "900m"

requests:

memory: "600Mi"

cpu: "500m"

$kubectl -n devteam-test create -f <file path>

* Now the second pod doesn't get created and gives an error because of the exceeding memory request.
* You can also restrict the totals for memory limit, CPU request, and CPU limit.

**Horizontal Pod Autoscaler**

* Horizontal Pod Autoscaler scales the number pods in a deployment, replica controller or replica set automatically based on the CPU utilization. ( or other custom metric )
* Apply the HPA configuration to the existing deployment.

kubectl -n devteam autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10

* Get the HPA configuration using the following command in Powershell.

$kubectl -n devteam get hpa -o yaml

**Generate Load**

* Now we will use load generator to generate some load on apache.

kubectl -n devteam run -i --tty load-generator --image=busybox /bin/sh

Hit enter and run below command to generate load on apache.

**while true; do wget -q -O- http://php-apache.devteam.svc.cluster.local; done**

* Within few minutes, we should see the higher CPU load by executing the following command.

kubectl -n devteam get hpa

kubectl -n devteam get pods

**Stop Load**

* You can stop the load on apache by typing <Ctrl> + C on new terminal.
* You can verify the result within a minute using the following command,

$kubectl -n devteam get hpa

**Multi-Container Pods**

* Multiple containers in a pod run on the same Network IP and same IPC(inter-process communication),this makes the containers use same port so that it can be a tightly coupled instead of loosely coupled.
* Multicontainer pod uses shared volumes so they can easily communicate data between the containers, ensuring data localization.

$ kubectl create ns devteam

**multi-containerpods.yml**

apiVersion: v1

kind: Pod

metadata:

name: two-containers

spec:

containers:

- name: nginx

image: nginx

imagePullPolicy: Always

ports:

- containerPort: 80

- name: ubuntu

image: ubuntu

imagePullPolicy: Always

ports:

- containerPort: 80

command: ["sleep", "30d"]

restartPolicy: Always

kubectl -n devteam create -f c:\user\PhotonUser\Desktop\multi-containerpods.yml

kubectl -n devteam get pods

**Running commands in a container with exec:**

* Execute the commands inside a container, use the following command to get into a shell. use -c at end of the command to go specific container.

kubectl -n devteam exec -it two-containers -c nginx bash

* Update the packages inside a container and install curl using the following command.

apt-get update

apt-get install curl

Access the service with use of **curl**

curl localhost:80

* Viewing full logs of a pod running a container inside it. This will also show the appending logs at run time.

kubectl -n devteam logs -f two-containers

**Note:** If you are using multi-container pod, want to inspect the logs of single container use -c at end of the command, and then container name.

kubectl -n devteam logs -f two-containers -c nginx

**Services & Deployments**

**Deployments**

* A deployment is a supervisor for pods. It provides declarative updates for Pods and ReplicaSets
* A ReplicaSet’s purpose is to maintain a stable set of replica Pods running at any given time.

$ kubectl create ns devteam

* Create a deployment that creates a replicaset to bring up three nginx pods. You can create the deployment in a particular namespace with specification of -n <namespace-name >, if you don't mention namespace while deploying into cluster, the default namespace is used.

**nginx-deployment.yml**

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

namespace: devteam

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx

ports:

- containerPort: 80

$kubectl create -f <file-path>

$ kubectl -n devteam get deployments

kubectl -n devteam get pods --show-labels

**Updating a Deployment**

$kubectl -n devteam edit deployment nginx-deployment

* This command will pop up a notepad window change the no.of replicas to 2, notice that the no.of pods gone to 2.

kubectl -n devteam get po

**Delete the pod**

kubectl -n devteam delete pod nginx-deployment-c5b5c6f7c-74brr

**Services**

* A Service is a Kubernetes object used to define a logical set of Pods and also create a policy by which other service can access them.

Types of Services

ClusterIp

NodePort

LoadBalancer

ExternalName

ExternalIPs

**ClusterIP**

ClusterIP service is the default service type in kubernetes cluster. It exposes the service on a cluster-internal IP. This type of service are reachable from within the cluster.

**clusterIP-service.yml**

apiVersion: v1

kind: Service

metadata:

name: test-service

spec:

ports:

- name: tcp-8089

protocol: TCP

port: 8089

targetPort: 80

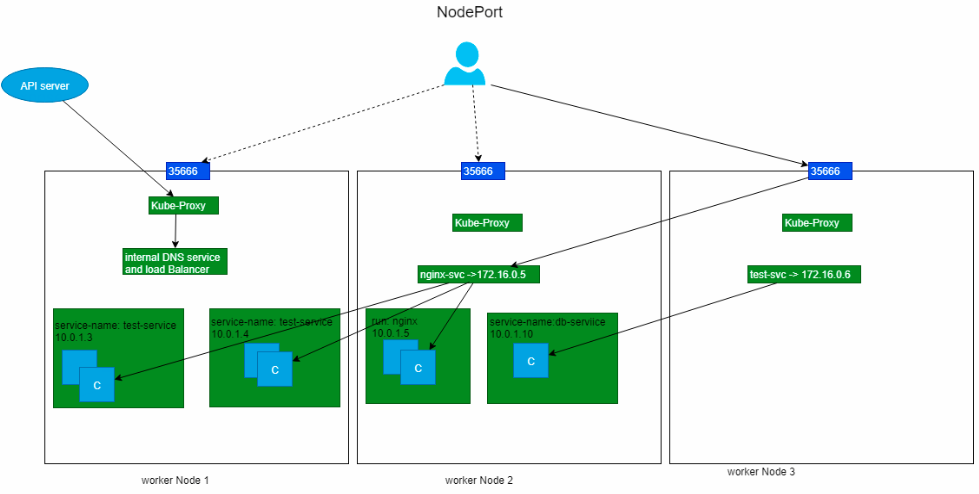
selector:

app: nginx

kubectl -n devteam create -f /c/Users/PhotonUser/Desktop/clusterIP-service.yml

**NodePort**

* NodePort type of service exposes the service on each node's IP at a static port.
* One can access a NodePort type service from outside of the cluster, by accessing the **NodeIP:StaticPort**.



apiVersion: v1

kind: Service

metadata:

name: nodeport-service

spec:

type: NodePort

ports:

- name: tcp-8089

protocol: TCP

port: 8089

targetPort: 80

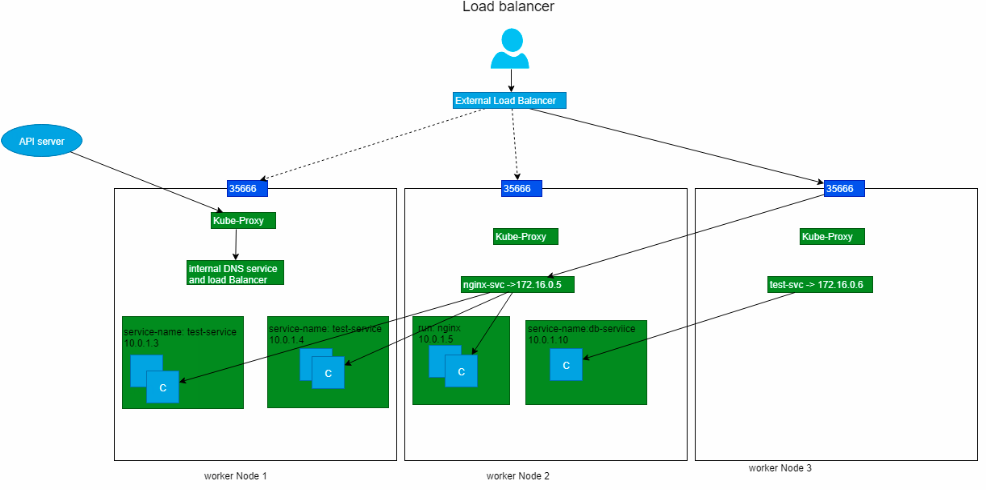
selector:

app: nginx

kubectl -n devteam create -f /c/Users/PhotonUser/Desktop/nodeport-service.yml

**LoadBalancer**

LoadBalancer type of service exposes the service externally using a cloud provider's Load Balancer resource (i.e. Load Balancer in OCI). One can access a LoadBalancer type service from outside of the cluster, by accessing the **LoadBalancerPublicIP:Port**.



apiVersion: v1

kind: Service

metadata:

name: loadbalancer-service

spec:

type: LoadBalancer

ports:

- name: tcp-8089

protocol: TCP

port: 8089

targetPort: 80

selector:

app: nginx

kubectl -n devteam create -f /c/Users/PhotonUser/Desktop/loadbalancer-service.yml

**ExternalName**

ExternalName type of the service maps the service to the contents of the externalName field, by returning a CNAME record with its vault.

kind: Service

apiVersion: v1

metadata:

name: test-service

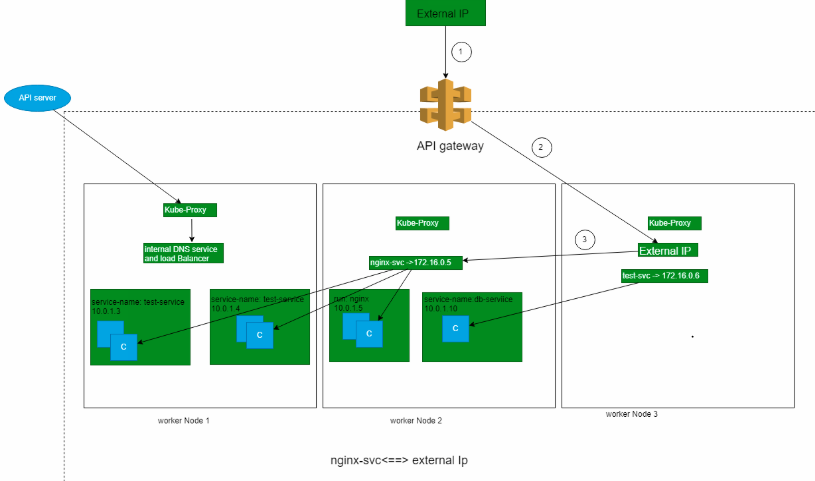
spec:

type: ExternalName

externalName: test-service.example.com

$kubectl create -f <externalName.yml>

**ExternalIPs**



* If there are ExternalIP's that route traffic to the cluster nodes, then kubernetes services can be exposed on those externalIPs.
* Traffic that ingresses into the cluster with the external IP (as destination IP), on the service port, will be routed to one of the service endpoints.
* In the ServiceSpec, externalIPs can be specified along with any of the ServiceTypes.

kind: Service

apiVersion: v1

metadata:

name: db-service

spec:

selector:

service-name: db-service

ports:

- name: http

protocol: TCP

port: 28015

targetPort: 28015

externalIPs:

- <replace-with-external-service-IP-address>

$kubectl create -f <externalIp.yml>

$ kubectl -n devteam get services

$kubectl -n devteam describe service <service-name>

$ kubectl -n devteam edit service <service-name>

$ kubectl -n devteam get po --show-labels

$ kubectl -n devteam get po -l app=nginx

* If you want to add a label **label=myservice** to pods, use the following command.

$ kubectl -n (namespacename) label po (podname) <key-value>

**Access the Services**

* Services can be accessed from the pod with service names and port assigned, shown below.

kubectl -n devteam exec -it <your pod id> bash

* Update the **apt-get** repository, use the following command to update.

$apt-get update && apt-get install -y curl

* Access Cluster-IP service with undefined has shown below.

$curl test-service:8089

* Note down the nodename and nodeport-service port (e.g.31368) shown below. If you want to get the port (PORT) of nodeport-service use the following command.

kubectl -n devteam get svc

kubectl -n devteam exec -it my-nginx-64fc468bd4-2wqtm bash

* Install curl if necessary and execute below command. Make sure you replace the IP's as per your deployment.

$curl <nodename>:30558

$curl aks-agentpool-16957051-0:30558

**Delete a service**

kubectl -n devteam delete service test-service

**Jobs & Daemonsets**

**Jobs**

* Job is a type of kubernetes object which runs a task or a job/script inside one or more pods and ensures that a specified number of them successfully terminate.
* As pods finish a task described in it successfully, the Job object tracks the completion and terminates the pod.
* When a job is deleted, it will also clean up the the pods which are created by it.
* One can also use Job to run multiple pods in parallel which runs for the duration of the task completion.

apiVersion: batch/v1

kind: Job

metadata:

name: job-sample

spec:

template:

spec:

containers:

- image: ashorg/sample:jobsleep

name: job-sample

restartPolicy: Never

backoffLimit: 4

$ kubectl create ns devteam

$ kubectl -n devteam apply -f /c/Users/PhotonUser/Desktop/job.yaml

$ kubectl -n devteam get jobs

$ kubectl -n devteam describe job job-sample

* To view the list of pods created by job use the following command in Powershell.

$ kubectl -n devteam get pods -o wide

**Output**

NAME READY STATUS RESTARTS AGE IP NODE

job-sample-mzhbt 1/1 Running 0 3s 10.244.2.88 10.0.2.2

* Here one can see that the pods has completed its task and the job is succesffully completed with pod showing the status as Completed.

$ kubectl -n devteam get pods -o wide

NAME READY STATUS RESTARTS AGE IP NODE

job-sample-mzhbt 0/1 Completed 0 3m 10.244.2.88 10.0.2.2

Now you have successfully deployed a Job and the pods of that job has completed the task successfully.

**DaemonSet**

* A DaemonSet is a kubernetes object which ensures every node present in the cluster runs a Pod.
* Whenever new nodes are added to the cluster, a pod is added to that node.
* As the nodes are removed from the cluster, pods are deleted and garbage collected.
* Once the DaemonSet is deleted the pods scheduled by it also gets deleted.

apiVersion: apps/v1

kind: DaemonSet

metadata:

name: daemonset-sample

labels:

service-name: daemonset-sample

environment: daemonset-sample

spec:

selector:

matchLabels:

service-name: daemonset-sample

environment: daemonset-sample

template:

metadata:

labels:

service-name: daemonset-sample

environment: daemonset-sample

spec:

containers:

- image: ashorg/sample:v1

name: daemonset-sample

resources:

ports:

- containerPort: 8089

$ kubectl -n devteam apply -f /c/Users/PhotonUser/Desktop/job.yaml

$ kubectl -n devteam get ds daemonset-sample

$ kubectl -n devteam describe ds daemonset-sample

$ kubectl -n devteam get pods -o wide

* Here one can see that the pod has been created in all the node (For this example if it is a 3 node cluster is used.)
* Now let's delete one of the pods of DaemonSet and observe what happens

$ kubectl -n devteam delete pod daemonset-sample-7blxh

$ kubectl -n devteam get pods -o wide

* One can see that the pod is started again in the same node with AGE as 1m

$kubectl -n namespace-name edit daemonset DaemonsSet-name

$ kubectl -n devteam delete daemonset daemonset-sample

**Cronjobs**

* CronJobs are just like cron on linux, they can run at any specified time intervals and also take the schedule just like on linux in the following format

\* \* \* \* \*

- - - - -

| | | | |

| | | | ----- Day of week (0 - 7) (Sunday=0 or 7)

| | | ------- Month (1 - 12)

| | --------- Day of month (1 - 31)

| ----------- Hour (0 - 23)

------------- Minute (0 - 59)

**Cronjob.yaml**

apiVersion: batch/v1beta1

kind: CronJob

metadata:

name: minute

spec:

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

jobTemplate:

spec:

template:

spec:

containers:

- name: hello

image: ubuntu

args:

- /bin/bash

- -c

- date; echo cronjob minute says hi; sleep 30d

restartPolicy: OnFailure

$ kubectl create -f cronjob.yaml

$ kubectl get cronjob

* Now you should see that the cron job minute successfully scheduled a job at the time
* The cron job has not scheduled or run any jobs yet. Watch for the job to be created in around one minute

$ kubectl get jobs

To get the pods for the job , run the following command in gitbash terminal

pods=$(kubectl get pods --selector=job-name=minute-1578378000 --output=jsonpath={.items[].metadata.name})

To get the logs of pod run the following command

kubectl logs $pods

kubectl describe jobs

kubectl delete cronjob minute