* 1. Kubectl get pods --show-labels -> This will get all the pods and the labels which are there on the pods.
  2. Kubectl apply -f app.yaml -> this will apply/create the pods
  3. Kubectl label pods delhipod name=bhupinder

-> Using this command you can imperatively specify any label to an existing pod

* 1. Kubectl get pods -l env=development

-> this will list all the pods which has a label env = development

5. Kubectl get pods -l env!=development

-> this will list all the pods which does not have a label env = development

* 1. Kubectl delete pod -l env = development
  2. Labels and selectors-> labels in kubernetes is a way by which you can assign a name or label to your kubernetes objects it can be nodes, it can be pods or any objects, selectors are used to search on those objects on which labels are applied. There are basically 2 types of selectors equality based(=,!=) and set based(in not in, exists ) and set based also supports filtering by multiple labels.
  3. Kubectl get pods -l 'env in (testing,development)'
  4. Kubectl get pods -l 'env not in (testting,development)'
  5. Kubectl get pods -l myName = bhupinder, class = development
  6. Kubectl cluster-info (this command connects to the kubernetes cluster and gives info)

Install docker

Install kubectl

Install conntrack

Install Minikube

Minikube start -> It will start the minikube and configure kubectl to use minikube cluster.

* 1. Kubectl get pods
  2. Kubectl get pods -o wide
  3. Kubectl get nodes -o wide
  4. Kubectl apply -f app.yaml -> this will apply the file to the cluster and create the pod
  5. Kubectl describe node <node\_name> => kubectl describe node minikube
  6. Kubectl describe pod <pod\_name> => kubectl describe pod testpod
  7. Kubectl logs -f testpod - > this will show you the logs inside the container of the pod, in case there are multiple pods running in your container then follow step 18
  8. Kubectl logs -f testpod -c c00 -> kubectlt logs -f testpod -c <Container\_Name>
  9. Kubectl exec testpod -it -- /bin/bash -> to go inside the container
  10. kubectl exec -it testpod -c c00 -- /bin/bash;  -> to go inside the container of the pod
  11. kubectl exec testpod -it -c c01 -- hostname -i -> To check the ip of the container (basically the ip is provided only to the pod which is the host so this command will list down the ip of the pod)
  12. Kubectl delete pod testpod -> kubectl delete pod <Pod\_name>
  13. Kubectl delete -f pod1.yaml -> kubectl delete -f <Name of deployment file>
  14. kubectl label pod pod1 name=sanchit -> When you want to add a label in imperative way. Kubectl label <ObjectType> <Object\_Name> key=value
  15. kubectl delete pod -l env=development -> another way of deleting pod
  16. minikube image load imgapps -> minikube image load <local docker image name>. This will basically load the docker image directly from your local docker images.

Ref (<https://stackoverflow.com/questions/42564058/how-to-use-local-docker-images-with-minikube>)

* 1. What is nodeSelector?

In kubernetes when we apply the manifest.yaml or app.yaml file to the kubernetes cluster to create a pod then the master's kubeScheduler creates a pod and then it checks which is the best node to which pods can be assigned and it takes care of the assignment of pods to nodes, now with nodeSelector we can control this and we can specify in our app.yaml as to which node or set of node this pod should be assigned. Basically in nodeSelector we provide labels and the same labels we add to the nodes that we have in our kubernetes cluster, and when kubescheduler creates pod and then assigns the pod to the node it matches the same labels which are provides in the nodeSelector.

Below is the image shown where you can specify nodeSelector in your app.yaml file and kubeScheduler will assign the pod to one of the nodes which has the same label.

appj.yaml 
Currently Open Documents 
app.yaml 
1 
appi.yaml 
2 
D appj.yaml 
3 
4 
6 
11 
12 
13 
14 
15 
16 
17 
18 
19 
21 
(functions) C 
kind: Pod 
apiVersion: 
metadata : 
name: testingpod 
labels: 
name: sanchit 
env: dev 
annotations: 
description: 
This is a testing pod which is created for testing purpose! ! 
spec: 
containers : 
— name: testcontl 
image: ubuntu 
command : 
env: 
— name: 
value: 
— name: 
value: 
["/bin/bash", " c 
env 
, "while true; 
do echo Hello From Container 1; 
sleep 5; 
done; 
My Development Environment 
envl 
My dev environment 
nodeSelector: 
hardware: t2—mediulTi 

As seen from the above screenshot the nodeSelector key should come at the same level as containers under the spec.

To Provide labels to nodes :

Kubectl label node minikube hardware=t2-medium => kubectl label <Object Type> <Node name> key=value

Example:

----------------------------------------------------

kubectl label node minikube hardware=t2-medium

For reference to documentations

<https://kubernetes.io/>

* 1. kubectl delete -f apprc.yaml -> this will delete the replication controller and all associated pods.
  2. Replication , loadBalancers, scaling up and down is by default not there with kubernetes but it can be implemented using the apis.

**Replication Controller:**

Replication Controller is an object in kubernetes just as same as Pod, Service, Deployment we can create a ReplicationController in those scenarios where we have a requirement that we need to create multiple copies of our container running in our system just to make sure that even if one of the pod fails the service will still be available from the other pod and a new pod will get created as kubernetes will check for actual state = desired state.

Also in kubernetes once a pod gets killed, then the same cannot be restarted again.

Replication Controller is from apiVersion v1 and it supports equality based selector (=,!=)

Under spec we specify replicas( it can be 1 to any number) and selector and the selector value should match with the label value below under template -> metadata.

apprc.yaml 
Free Mode 
(functions) C 
1 
2 
3 
4 
7 
8 
10 
11 
12 
13 
15 
16 
17 
18 
19 
21 
22 
23 
24 
25 
26 
27 
kind: 
ReplicationCont roller 
apiVersion: VI 
metadata : 
name: rc 
labels: 
name: rcm 
spec: 
replicas: 2 
selector: 
name 
rc 
template: 
metadata : 
. pod3 
name 
annotations : 
description . 
labels : 
name: rc 
spec: 
containers: 
Hello from Replication controller 
— name: myrccontainer 
image : 
ubuntu 
command : 
env : 
— name: 
value: 
— name: 
value: 
['I/ bin/ bash", c", "while true; 
env 
deployment 
app 
application 
do echo Hello from replication controller! ; 
sleep 5; 
done;" ] 

**ReplicaSet**

ReplicaSet in kubernetes is an object just like ReplicationController and it helps in creating replicas or copies of your pods to provide fault tolerant behaviour. This is same like ReplicationController but the only difference is that it works with set based selectors as well as Equality based selector.

For set based selector you need to specify matchExpressions under selector and for equality selector you need to specify matchLabels.

Also since it was a new addition so it comes under apiVersion apps/v1.

Reference : <https://medium.com/avmconsulting-blog/replication-controller-replica-sets-in-kubernetes-820f3cec7170#:~:text=The%20replica%20set%20and%20the,set%20supports%20set%2Dbased%20selectors>.

apprs.yaml 
Currently Open Documents 
O apprs.yaml 
untitled text 
13 
16 
17 
18 
19 
21 
22 
['I/ bin/ bash", "while true; 
25 
26 
1 
2 
3 
4 
5 
6 
9 
10 
11 
12 
15 
23 
24 
28 
kind: 
ReplicaSet 
apiVersion: apps/vl 
metadata : 
name: myrs 
spec: 
replicas: 2 
selector: 
matchLabels: 
myname: sanchit 
#matchExpress ions : 
# — {key: myname, operator: In, 
#— {key: env, operator: Not In, 
template: 
metadata : 
name: myrs 
annotations : 
values: 
values: 
[sanchit, Bupinder, 
[production]} 
Bhopend ra]} 
description: 
Hello Welcome from ReplicaSet 
labels : 
myname: sanchit 
spec: 
containers: 
— name: mycontainer 
image: ubuntu 
(functions 
done; "] 
command : 
env : 
— name: 
value: 
do echo Hello world! ; 
sleep 5; 
sanchit 

<https://kubernetes.io/docs/reference/kubectl/cheatsheet/>

For scaling up and scaling down the containers

* 1. kubectl scale --replicas=8 rc -l name=myrc;

**Deployments in Kubernetes:**

Deployments in kubernetes helps in providing rollbacks and rollouts, suppose an application like facebook comes up with new features/bug fixes then it will be available

to the users as an update, therefore using kubernetes helps in achieving this.

Kubernetes has an object known as Deployment which we specify in kind, when you specify an object of type Deployment then it will basically have a structure where the top level is deployment and then there is ReplicaSet which is created and then the pods get created under the ReplicaSet.

Deployment object deals with ReplicaSet and ReplicaSet deals with pods and pods run containers. With the help of deployments users can jump from one revision to another revision and this helps in going to previous deployments as well. Also when you apply deployment.yaml file again to the cluster after making modifications then it will create a version of the deployment which you can check by using kubectl get deployments it will list down the versions.

Example of a deployment.yaml file.

deployment.yaml 
Free Mode 
Currently Open Documents 
O deployment.yaml 
1 
untitled text 3 
2 
3 
4 
9 
10 
11 
12 
13 
15 
16 
17 
18 
19 
21 
22 
23 
24 
25 
26 
27 
28 
29 
kind: 
Deployment 
apiVersion: apps/vl 
metadata : 
name : testdeployment 
annotations : 
description: 
This is my first deployment! 
labels: 
name: deployment 
spec: 
replicas: 3 
selector: 
matchLabels: 
name: testdeployment 
template: 
metadata : 
name: deployment 
labels : 
name: testdeployment 
spec: 
containers: 
— name: containerl 
image: centos 
, "while 
using kind as Deployment 
true; do echo Love All! 
sleep 5; 
(functions) C 
done;" ] 
command : 
env: 
— name: 
value: 
["/bin/bash", " c 
greetings 
hello 

Commands that we can use

1. kubectl apply -f deployment.yaml => This will apply the deployment.yaml to the kubernetes cluster.

2. kubectl get deployments

3. Kubectl describe deployments testdeployment => This will basically show or list the deployment details

4. kubectl get rs

5. kubectl get pods

6. kubectl logs -f <Pod\_name>

Example of scaling up

* 1. kubectl scale --replicas=5 deployment testdeployment
  2. Kubectl scale --repliacs=1 deployment testdeployment

Rollouts:

1. kubectl rollout undo deployments testdeployment => this will undo or take you to the previous deployment.

2. kubectl rollout status deployments testdeployment => checking the status of deployments

3. kubectl rollout history deployments testdeployment => This will show the history of deployments

Minikube dashboard -> this will open a UI and

**Kubernetes services:**

Every pod in kuberenets has an IP associated to it and these are private IP's so these IP's cannot be access from outside the cluster by default. Although we can expose these IP's for the internet outside our cluster by using services.

Pods can be created on any node and kube-scheduler decides which node is appropriate to create a pod and hence if a pod in a ReplicaSet was running on node-1 and it gets killed then the replacement might get created on any other node suppose node-2 and the IP assigned to the replacement pod will also change.

So when we access the virtual IP address it redirects the request to the pods which are mapped with this virtual IP address. So even if the IP's gets changed in the backend the Services would ensure that the IP to access the services will be static.

**Services** in kubernetes helps in solving one important problem where in case of ReplicaSets/ReplicationController the IP of the pods gets changed over time, suppose

When a new deployment is created then all the previous pods are killed and new pods are created for the second deployment, Therefore for a scenario where suppose there is a pod which helps some other set of pods to run and they communicate with the private IPs of this pod which get's killed or scaling down happens then those set of pods would not be able to communicate as the new pods which will get created in place of this pod will again have a new IP address, therefore to solve this problem we use services in kubernetes where services provides a virtual IP address to the ReplicaSet and the pods which are created under the replica sets will have their ip's mapped with the virtual IP provided by the service object.

Services help in providing a static virtual IP address which gets mapped to the pods which are created.

Kubeproxy which resides in the slave nodes also plays an important role here and helps in mapping the virtual-IP to the pod IP and it keeps on querying the apiserver which resides on the master node to check learn about all the new services.

Services basically help to expose the Virtual IP address mapped to the pod's IP address to the outside world/internet.

Services can be exposed in 4 different ways

1. ClusterIP (Default)

2. NodePort

3. loadBalancer

4. headless

By default the NodePort range can be in between ports 30,000 -> 32,767  
The set of pods which are to be exposed to the outside world/internet are usually determined by a selector.

(based on Labels the pods will get selected)

Basically NodePort is one level up the ClusterIP and it helps in exposing the application which is running in the container inside the pod to the outside world, and in the service we expose the port number to which the NodePort assigns the port and using the public DNS address of the host system we ca use the NodePort's port and access the application from outside the cluster or the internet.

Basically when you define a service through cluster-IP it will group all the pods having the labels which matches the selector in the service.yaml file and make a cluster of all those pods, then it will provide a cluster IP which would be a static IP address that can be made use of to access your services running inside the pods.

Example 1 of Service (ClusterIP)

service.yaml 
Free Mode 
Currently Open Documents 
service.yaml 
1 
untitled text 
2 
3 
4 
5 
9 
11 
(functions) C 
kind: Service 
apiVersion: VI 
metadata : 
name: clusterservice 
spec: 
po rts : 
— port: 80 
targetPort: 80 
selector: 
name: my label 
type: NodePort 

Example 2 of Service (NodePort)

**Volumes:**

In kubernetes volumes are within containers, and if suppose the containers crashes then volume inside those containers also crashes although kubelet restarts the container but the data inside the container is lost and when a fresh container is created the volume inside the newly created container is also created fresh.

Therefore to overcome this problem in kubernetes we attach a volume to a pod and even though the container crashes the volume attached to the pod remains unaffected

And the lifespan of the volume is as good as the pod, which means the volume attached to the pod will be available only till the pod is not crashed and if the pod crashes the volume will also be lost.

There are two types of Volumes:

1. emptyDIR:-> this will be mounted to the pod as soon as the pod gets assigned to the node, initially it is empty and it will have a lifespan of the pod when pod will get crashed then this volume will also be lost and when this pod will be re-created the volume would be blank initially.

Here you can see the below yaml file which uses emptyDir.

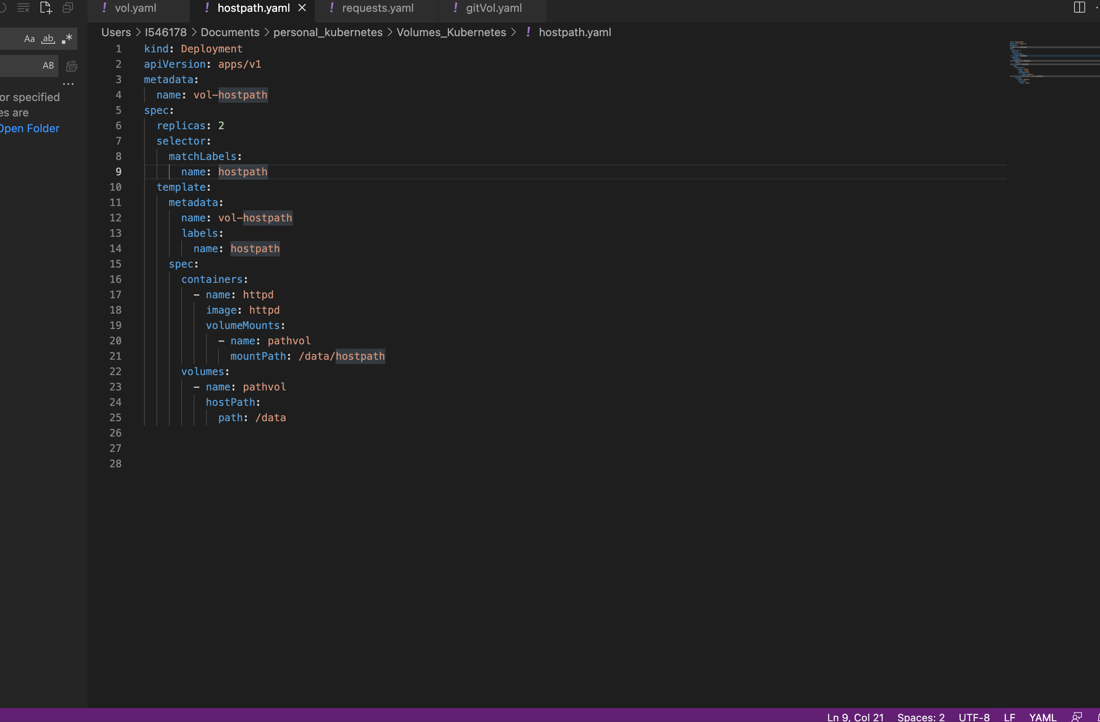
volumes_emptydir.yaml 
Free Mode 
Currently Open Documents 
volumes_emptydir.yaml 
1 
2 
3 
4 
5 
6 
11 
12 
13 
14 
15 
16 
17 
18 
20 
21 
22 
23 
24 
25 
26 
27 
28 
29 
30 
31 
32 
33 
34 
35 
36 
37 
38 
39 
40 
kind: 
Deployment 
apiVersion: apps/vl 
metadata : 
deplyomentvol 
name: 
spec: 
replicas: 1 
selector: 
matchLabels: 
name: deployment 
template: 
metadata : 
name: deploymentvol 
labels : 
name: deployment 
annotations : 
description: 
Hello from volumes 
spec: 
containers: 
— name: COO 
image : 
ubuntu 
command: ["/bin/bash", , "while true; 
env : 
— name: spoc 
value: 
sanchit 
vo lumeMounts : 
— name: volumel 
mountPath : 
"/tmp/data" 
(functions) C 
do echo Hello from container cøø;sleep 5;done;"] 
— name: COI 
image 
. centos 
command: ["/bin/bash"," c 
env : 
— name: spec 
value: 
hello 
vo lumeMounts : 
— name: volumel 
mountPath : 
"/tmp/data" 
volumes : 
— name: volumel 
emptyDir: { } 
, "sleep 10000; 

2. HostPath:-> Just like in Docker where we were able to share volume from host to container by mapping a host machine directory to container's directory the same way in kubernetes we can map a host machine's directory to a pod's volume. The pod's volume will reflect the changes done from host machine and vice versa.

This becomes helpful when you want to access the volume data from your pod outside of your pod.

Hostpath volume is created on the node on which the pod is created, so the life of the hostpath volume depends upon the life of the node, if the node goes down the storage too would be down and therefore it is not advisable to be used.

Changes which are done via one pod would be visible to another pod when it is created on the very same node.



Kubectl create deployment —image=testimg —ImagePullPolicy=Never (This command would not work instead use the deploy.yaml)

—/Documents/spring_ws/testapp/deploy.yaml C 
13 
15 
16 
17 
18 
1 
2 
3 
4 
5 
6 
11 
12 
14 
20 
21 
kind: 
Deployment 
apiVersion: apps/vl 
metadata : 
g reeting 
name: 
spec: 
replicas: 2 
selector: 
matchLabels: 
name: Hello 
template: 
metadata : 
name: greeting 
labels : 
Hello 
name : 
annotations : 
description: 
spec: 
containers: 
— name: COO 
Hello from Greet service 
image: imgtest 
imagePullPolicy: 
Neverl 

Kubectl apply -f deploy.yaml

Kubectl expose deployment greeting —type=NodePort —port=9890

Kubectl get service

minikube service mi(This will open the url for you and you will be able to access your project through browser)

kubectl scale --replicas=3 deployment greeting

minikube image load <image name>

Minikube on mac os M1 chip

<https://cs101.blog/2023/01/12/run-a-minikube-on-mac-with-m1-chip/>

docker push sanchit007:springboot\_docker\_kubernetes

docker tag local-image:tagname new-repo:tagname

docker tag imgtest:latest sanchit007:springboot\_docker\_kubernetes

Example :

docker tag imgtest:latest sanchit007/springboot\_docker\_kubernetes:latest

docker push sanchit007/springboot\_docker\_kubernetes:latest

What is requests and limits?

A pod by default can work with no limits on cpu and memory however we can specify from our end optionally on how much cpu and memory each container needs.

Kube-Scheduler would decide on which nodes the pods would be placed provided the node has the available resources to satisfy the pod request.

Cpu requests are specified in millicpu and memory is specified as MB.

In kubernetes a cluster can be divided into namespaces, if a pod is created into a namespace that has a default cpu limit and the pod has not specified it's own cpu limit then the limit would be the default namespace limit.

Namespace can be assigned a resource Quota Object this will limit the amount of usage allowed to the objects of that namespace.

You can limit

* 1. Compute (cpu)
  2. Memory
  3. Storage (pvc)

Here are the restrictions that a resource quota imposes on a namespace

* 1. Every container that runs in the namespace must have it's own cpu limit.
  2. The total amount of cpu used by all the containers in the namespace must not exceed a specified limit

Request: It is the no. of resources which are required by the pod and kuberenets will use this value to place the pod on which node (node should have the available resources).

Limit: It is the maximum no. of resources that kubernetes would allow the container to use. In case when the limit is defined and request is not defined then request becomes equal to limit (limit = request) however when limit is not specified and request is only provided then limit would be taken as the default range of that namespace.

What is namespace?

Namespace in kubernetes helps to separate a cluster into logical units.

In the cluster a lot of kubernetes objects are created if multiple users are using the same cluster it would be difficult to organise and manage the kubernetes objects created within a cluster therefore by using namespace it is a better way to re-organise and have a better way of grouping kubernetes objects. For example in a cluster there can be a namespace which would be only having objects related to the development environment, there can be another namespace for QA which would be a collection of all the kubernetes objects related to the QA or testing environment and there can be another namespace which would contain all the kubernetes objects related to the PROD environment. Therefore namespaces in kubernetes is a group or collection of objects which belongs to a certain group.

By default there are these many namespaces which are available, and whatever objects we create in kubernetes cluster by default it goes into the default namespace. But we can create objects into our new namespace.

default                Active   191d

kube-node-lease        Active   191d

kube-public            Active   191d

kube-system            Active   191d

kubernetes-dashboard   Active   191d

Command for getting the namespace which is used by kubernetes

kubectl config view | grep namespace

What is resource Quota?

Resource Quota is basically the max amount of resources that can be used by a namespace we can define a resource quota for each of the namespace like suppose we have defined 5GB of storage to a namespace let's say Development similarly we can have another namespace named as QA and we can define suppose 10 GB of resource quota for the QA namespace.

So every namespace has a resource quota.

Command for creating a namespace:

kubectl create namespace development

Kubectl apply -f app.yaml -n development (this will create a pod in the development namespace)

Kubectl get pods -n development (this will list all the pods belonging to the development namespace)

Kubectl delete pod -n development

What is HPA (Horizontal Pod Autoscaling)?

HPA or horizontal pod autoscaling is a feature in kubernetes by which the pods inside kubernetes cluster would be scaled automatically as soon as the load increases on the application and it can be scaled down as well after a certain period of time which is configurable and the pods would be deleted after certain period of time (scaling down) as soon as the load decreases on the application.

1 . Query for metrics 
2. Calculate the replicas 
POD 1 
Metrics Server 
3. Update the Replica count 
Deployment 
Replica Set 
4. Scale the desired Replicas 
POD 2 
POD N 

Metrics Server: There is a metrics server which needs to be installed in the cluster and the HPA would query the metrics server every 30sec to get the load on the application and it would calculate how many pods are required based on the current load on the application and as soon as the load on the application increases from a given threshold percentage then it will automatically calculate as to how many pods are required for satisfying the current requirement.

There is a cooling period of 5 mins after which the HPA would scale down the number of pods which are currently required to satisfy the current requirement as soon as the load on the application decreases.

For example if the threshold value is defined as 40% and initially 2 pods are there having the cpu-consumption as 20% each respectively and when more load increases on the application suppose the cpu-consumption is now 90% and the threshold defined initially was 40% then since the current cpu-consumption is now greater than the defined threshold value of 40%, so HPA would calculate how many replicas would be required to distribute the load so that the load goes below the threshold value and then it creates those many pods, so in this case 1 more pod would be created so that each of the pods would have 30% of load. In total 3 pods would be running after the load increases to 90%.

Pod 1 -> 20%

Pod 2 -> 20%

current cpu-consumption -> (20% + 20%)/2 => 20% is the current load initially.

As soon as the load increases to 90% (which is greater than the threshold of 40%) then HPA calculates the new number of Pods it has to spin up so the the load goes below the threshold value so it will spin up one more pod and the load of the cpu consumption would be distributed to 30% each respectively.

Pod 1 -> 30%

Pod 2 -> 30%

Pod 3 -> 30% (new pod created after HPA operation)

The collective load on the system would be now calculated as (30%+30%+30%  
)/3 = 30% which is below the threshold of 40%.

Similarly when the load on the application decreases the HPA would get all the metrics of the current load from the metrics server and then it will wait for certain amount of time which is the cooling period (5 mins) and then HPA will calculate how many pods are required in this situation and it will scale down and remove the pods which are not required.

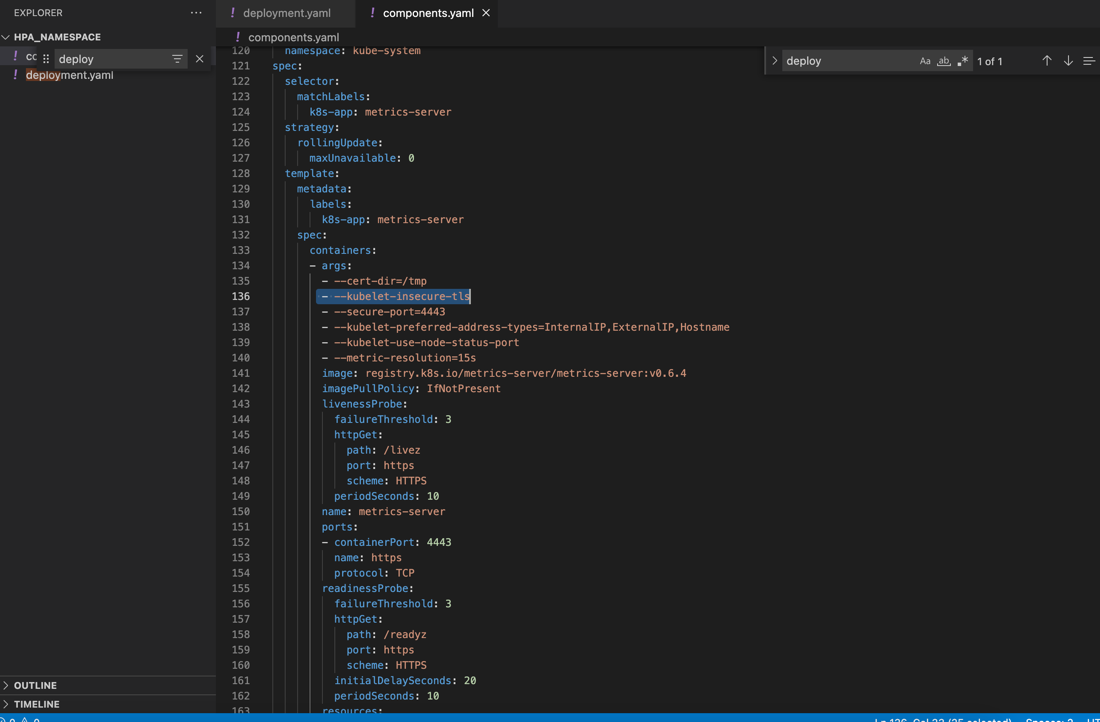
Kubectl autocale deployment mydep --cpu-percent=20 --min 1 --max 10

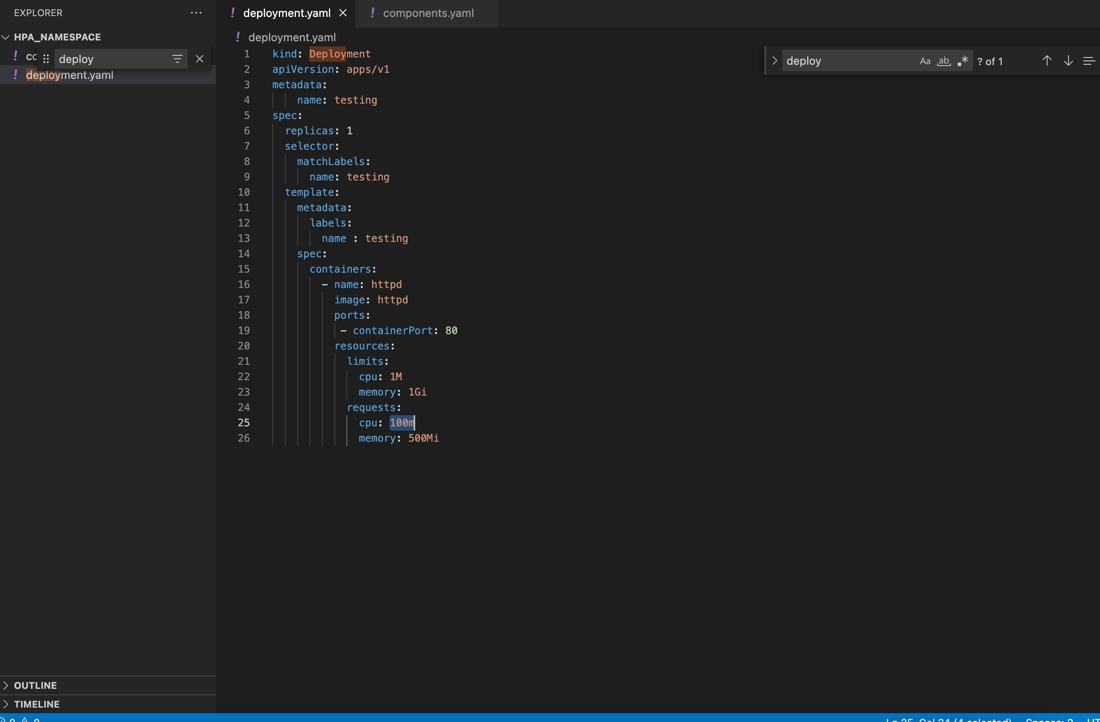
Metric server link:

<https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml>

Command you need to set inside the components.yaml under deployment

- --kubelet-insecure-tls





You can use the service of type nodeport to expose your application.

kubectl expose deployment testing --type=NodePort --port=80

minikube service testing

Kubectl get hpa -w (to check the current status of the HPA)

To increase the load on the application you can open another terminal and increase the load on the application like below:

for ((i=1;i<=10000;i++)); do curl  -I -k "<http://127.0.0.1:50394>”; done

Once you have done this you will be able to see that the load increases on the application.

After sometime you will be able to see that as the load reduces the pods that were created would be scaled down automatically.

Also the HPA would be applied to only objects such as Deployment, ReplicaSet, Controller etc

Steps for Horizontal Pod Autoscaling:

* 1. Install a metrics server in your cluster ( <https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml>)
  2. Now, in the components.yaml add the parameter --kubelet-insecure-tls under deployment (this is done basically to avoid installation of additional certificates.)
  3. Now, create a deployment file and then apply it on the cluster
  4. Now, expose the application under your deployment using services
  5. Now, create the HPA using command like, kubectl autoscale deployment depl --cpu-percent=20 --min=1 --max=10 this will create an HPA on the deployment object and now to see the HPA in action increase the load by making 10000 requests to the exposed application and witness the HPA in action.

What is Helm?

Helm is a tool just like apt for ubuntu and yum for linux, it is a packaging manager which installs the packages.

Now, helm charts is nothing but the group of kubernetes objects (like Deployment, secrets, configMap, Namespace, persistentvolumes,persistentVolumeClaim, services) that are created inorder to run your application so, helm charts is nothing but grouping all of these kubernetes objects that are required for your application to run into a single package.

A chart is a helm package, it contains all of the resource definitions necessary to run an application, tool or service inside your kubernetes cluster think of it as like the kubernetes equivalent of a homebrew formula, apt, yum or RPM file.

Helm Hub or helm repository?

Helm hub is nothing but a repository, from where all the packages related to your application would be residing. It is same like docker hub or github.

Why we need Helm?

Deploying and maintaining the kubernetes objects typically .yaml files is a tedious and time consuming task, therefore for deployment Helm simplify this process and it creates a single package that can be applied to the cluster.

Helm3 was introduced in Nov 2019, and it automatically maintains a database of all the versions of your releases whenever something goes wrong during deployment rolling back to the previous version is just a command away.

What is a Release?

A release is an instance of a chart running in a k8s cluster one chart can often be installed many times into the same cluster and each time it is installed a new version is created

Consider a mysql chart if you can install that chart twice each one will have it’s own release which will in turn have it's own release name

Revision | Request

* 1. | Installed Chart

2. | Upgrade to 11

Helm keeps a track of all charts execution (Install Upgrade Rollback)

Repository: Location or repo where packaged charts can be stored and shared

Helm 3 :

Helm3 is a single client architecture tool, earlier helm2 involved client-server architecture and it had some limitations which were fixed with helm3 and since kubernetes did not supported the concept of RBAC (Role based access components) therefore client-server architecture was there with heml 2 but now since RBAC is supported with kubernetes that is why now a single client architecture is provided.

Helm3 is a single client architecture, one executable is responsible for implementing helm there is client-server split nor is the core processing logic is distributed among components.

Implementation of Helm3 is a single command line client with no in cluster server or controller this tool exposes command line operations and unilaterally handles the package management process. Helm and helm library are coded in GO language developed by Google.

We now write our yaml files in a way that values are provided via external yaml files and those files would be referred in the manifest files.

Helm install jenkins

Now helm will install jenkins from the helm repository and then it will ask the api server in the kubernetes cluster to apply the package.

With this approach the tedious task of creating a deployment file for applying a container on the cluster would be simplified.