**Kubernetes – Part 2**

**Pods:**

Let’s assume, the application already developed and build into Docker images, and available in Docker Hub so Kubernetes can pull it down. We do have Kubernetes cluster has already been set up and it’s working; [that could be a Single-node or Multi-node setup] and all the services are in running state. As we discussed before Kubernetes main aim is to deploy application in the form of containers on a set of machines that are configured as worker nodes in a cluster. However, Kubernetes doesn’t deploy Containers directly on the worker nodes. The containers are encapsulated in to a Kubernetes object known as Pods. A Pod is a Single instance of an Application. A Pod is the smallest object that we can create in Kubernetes.

**Scenario 1:**



In simple, let’s go through with a setup where we have A single node kubernetes cluster with a single instance of our application running in a single Docker container encapsulated in a Pod; What if the number of users in our Application increase and we need to scale our application to balance this situation, where we need to add additional instances of our web application to share the load, Now, where would we spin up additional instances? Do we bring up new container instance within the same pod? No, instead we create new pod all together with the new instance of the same application, as per this container we got 2 instances of our same web application running on 2 separate pod on the same kubernetes system or Node. === Now, what if the users further increases and our current doesn’t have enough capacity? Well, at that time we can deploy additional pods on a new Node in the kubernetes cluster; now we will have a new node added to the cluster expand the cluster physical capacity, According to this one, Pods usually have a one to one relationship with containers running our application. To scale up we create new pods and to scale down we delete the existing pod. We will not add additional container to scale our Application.



***Multi-Container PODs*** == Are we restricted to a single container in a single POD? NO, a single POD can have multiple containers, except for the fact that there are not usually multiple containers of the same kind. As we discussed earlier one, if there is a requirement then we will additionally create a PODS, but there are situations where we might have Helper containers that will some kind of supporting tasks for our application such as processing a user ended data, processing a file uploaded by user etc., and we need this helper container to live along side of our application container, In that case, we can have both of these containers part of the same POD. So that when a new application is created the Helper is also created, and when it dies Helper goes along with them, since they are part of the same POD. These 2 containers can communicate with each other directly by referring each other as their localhost, since they share the same network + they can easily share the same storage space as well.

**Scenario 2:**

Let for a moment we will keep Kubernetes out of our discussion and talk about simple Docker containers, let’s assume we were developing a process or a script to deploy our application on a Docker host, then we would first simply deploy our application using a simple **“docker run python-app”** command, and the application runs fine and our users are able to access it. When the load increases, we deploy more instances of our application by running docker run commands many more times, this works fine. Sometimes in the future our application is further developed, undergoes architectural changes and grows and gets complex. We now have a new helper container **“docker run helper –link app1”** that helps our web application by processing or fetching data from elsewhere. These helper containers maintain a 1 to 1 relationship with our application container and thus needs to communicate with the application containers directly and access data from those containers. For this, we need to maintain a map of what app and helper containers are connected to each other, we would need to establish network connectivity between these containers ourselves using links and custom, then we need to create sharable volumes and share it among the containers, most importantly we need to monitor the state of the application container, when it dies, manually kill the helper container as well as it’s no longer required. When a new container is deployed, we would need to deploy the new helper container as well, BUT when comes to kubernetes Pods all of this will be automatically done for us; we just need to define what containers a pod consists of and the containers in a pod by default will have access to the same storage, the same network namespace and same fate as in they will be created together and destroyed together.

Even if our application didn’t happen to be so complex and we could live with a single container, Kubernetes still requires you to create pods, but this is good in the long run as your application is now equipped for architectural changes and scale in future.

[\*\*\* However, also note that multi containers pod are a rare use case and we’re going to stick to single container per pod]

Kubectl == Let us now look at how to deploy pods

**“*kubectl run nginx*”,** what this command really does? It deploys a Docker container by creating a pod. [\*\*\* make sure if you are going to use some other registry for your image other than docker, then you need to mention the full path of your image repository]

So, 1st it creates a Pod automatically and deploys an instance of the Nginx docker image.



But where does it get the application image from? For the we need to specify the image name using “image” parameter, ***“kubectl run nginx –image nginx”*** in this case, the nginx image is downloaded from the Docker Hub repository, Docker hub, as we discussed is a public repository where latest Docker images of various applications are stored. You could configure kubernetes to pull the image from the public Docker hub or a private repository within the organization. Now that we have a pod created, how do we see the list of pods available? ***“kubectl get pods”*** commands helps us to see the list of pods in our cluster.

Installing **Minikube** == follow the official page of kubernetes (kubernetes.io) == documentation == tasks == install tools == install & setup on Ubuntu linux (according to your OS you can choose one) == I am using here “Install using native package management”.

Pre-requisites:

2CPU

2Gib RAM

Min 20 Gb Storage

1. Create an Ubuntu machine 20.04 – t2.medium – Open Security Group – 30Gib – Launch instance.

2. Open terminal, update your system

sudo apt update

sudo apt upgrade -y

Once all the update is over, reboot the system

sudo reboot

3. Install Docker

sudo apt-get install ca-certificates curl gnupg lsb-release

sudo mkdir -p /etc/apt/keyrings

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo \

"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin

sudo systemctl status docker

{if you are facing error msg here, it is because of the space issue; follow these commands to rectify for the same:

sudo find /var/log -type f -delete

sudo rm -rf /var/cache/apt/\*

sudo apt clean all

sudo rm -rf /var/lib/docker.bk

sudo systemctl restart docker

4. Add user to docker group

sudo groupadd docker

sudo usermod -aG docker $USER && newgrp docker

sudo systemctl enable docker

sudo systemctl status docker

5. Download and Install Minikube Binary

curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64

sudo install minikube-linux-amd64 /usr/local/bin/minikube

minikube version

6. Install Kubectl

curl -LO https://storage.googleapis.com/kubernetes-release/release/`curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt`/bin/linux/amd64/kubectl

7. Set the executable permission

chmod +x kubectl

sudo mv kubectl /usr/local/bin/

8. Verify the kubectl version

kubectl version -o yaml

9. Start Minikube Cluster

minikube start --driver=docker

minikube status

10. Interact with Minikube Cluster

kubectl get nodes

kubectl cluster-info

11. Managing Minikube Addons

minikube addons list

minikube addons enable dashboard

minikube addons enable ingress

minikube addons list (to confirm all the addons)

12. Deploy Applications

kubectl create deployment hello-minikube --image=kicbase/echo-server:1.0

kubectl expose deployment hello-minikube --type=NodePort --port=8080

kubectl port-forward service/hello-minikube 7080:8080

[Once it starts to forward, call the localhost via browser (<http://localhost:7080/>) to get the output for the same]

13. Manage you Clusters

Pause Kubernetes without impacting deployed application:

minikube pause

Unpause a paused instance:

minikube unpause

Halt the Cluster:

minikube stop

Change the default memory limit (requires a restart):

minikube config set memory 9001

Browse the catalog of easily installed Kubernetes services:

minikube addons list

Create a second cluster running an older Kubernetes release:

minikube start –p aged –kubernetes-version=v1.16.1

Delete all of the minikube clusters:

minikube delete –all

**KOPS Installation:**

1. Create 1 ubuntu machine (20.04) - t2.micro

2. Open putty - switch to root user (sudo su -)

3. We need to install kops binaries from kubernetes official page (curl -Lo kops https://github.com/kubernetes/kops/releases/download/$(curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut -d '"' -f 4)/kops-linux-amd64) [by default all our setup files will be presented inside our binary files]

4. Then we are providing the exe permission to the kops file (chmod +x ./kops)

5. We are going to move this to the bin location (sudo mv ./kops /usr/local/bin/) [/usr/local/bin/ is the default location where our binary/exe files will be presented]

6. Now we are trying to download the kubectl binary file using curl (curl -Lo kubectl https://storage.googleapis.com/kubernetes-release/release/$(curl -s https://storage.googleapis.com/kubernetes-release/release/stable.txt)/bin/linux/amd64/kubectl)

7. Then we are providing the exe permission to the kubectl file (chmod +x ./kubectl)

8. We are going to move this to the bin location (sudo mv ./kubectl /usr/local/bin/kubectl)

9. Now, I am trying to access this by a specific user, hence we are creating a user in IAM (only programatic access is enough) - administrator permission (real-time the permissions will be different)

10. When we are giving (aws configure) - we are getting an error msg that cli is missing, hence we need to install it - 1st update (apt-get update) - now install cli (apt install awscli) if still getting an error msg, then try to update the packages (apt-get update)

[\*\*\* In ubuntu machine, cli is not pre-installed]

11. Now configure the user, with all the credentials

12. We need to export the access key and secret access key as variable for kops further communications [as we cannot provide this information's manually all the time]

export AWS\_ACCESS\_KEY\_ID=(copy paste access key)

export AWS\_SECRET\_ACCESS\_KEY=(copy paste secret access key)

13. Now, we are going to setup cluster with multi-node; before doing this we need to make sure the kubernetes state file are available on the s3 bucket [this is mandatory step for kops], give command (aws s3 mb s3://may2k24 --region us-east-2) - confirm the same on the s3 bucket

14. We need to enable versioning for this bucket - (aws s3api put-bucket-versioning --bucket may2k24 --versioning-configuration Status=Enabled) [there may be permission denied, due to recent update - if so, then update the bucket versioning manually - open the bucket - properties - enable versioning]

[Hereafter, all our kubernetes setup files, config files, state files everything will get saved on this s3 bucket for backend purpose]

15. We are going to create a separate authentication for communication in-between master to worker node - to do this give command (ssh-keygen) - now we will get public and private key; henceforth we are going to use only this for further communications. [Click enter wherever it is asking for location]

16. Next step we are going to create a cluster, this can be done by 2 different ways [1. normal way, we need domain name for connection; 2. gossip-based cluster, don't require a register domain, we can use dummy domains for that - and this needs to ends with (prefix name.k8s.local) prefix can be any name]

17. Give command (export NAME=shiv.k8s.local) - once the cluster got created we need to mention the location where the state file needs to be saved (export KOPS\_STATE\_STORE=s3://may2k24)

18. We need to create our cluster now, using this command (kops create cluster --zones us-east-2a ${NAME}) [This entire setup with take atleast 5 - 7 mins to get created]

19. Now we need to update the content - give command (kops update cluster --name shiv.k8s.local --yes --admin)

[After these commands we are going to get some suggestion, please follow; ex: we need to validate this only after 10m if time; and they will provide with the ssh key for future communication, copy paste those contents on a notepad]

20. After sometime, we can validate the cluster using the command (kops validate cluster) - once it got successful, we will be able to see 1 master and 1 node got created [but we haven't mention this anywhere, by default this is how it is going to get created] - but if you would like to edit, then on the suggestions itself they have provided with the necessary commands like, (kops edit ig --name=shiv.k8s.local master-us-east-2a) [this command is for master edit] (kops edit ig --name=shiv.k8s.local nodes-us-east-2a) [this is for node edit] (kops edit cluster shiv.k8s.local) [this is for cluster edit]

***Task:***

Try changing the node size, update and check for the output.

21. Check the s3 bucket for state files, this will create load balancer, auto-scaling [by default this is providing our cluster setup with high- availability]

22. As we know that we created an instance and did all this process; that instance are called as (client machine) - henceforth I am going to do all the activity by using this client machine only by using ssh, we are not going to touch our master/node anywhere.

23. By giving this command (kubectl get nodes) - we can see both our nodes master as well as worker

[\*\*\*FYI: We can stop this temporarily also, as we are going to use this repeatedly - go to AutoScaling - edit change all the capacity to "0" - save, do the same for both master and worker, so that we can run accordingly; stop the client machine for time-being]

In real-time we will do the setup by using Ansible, as we are going to have a lot of machines in our env, search in google for (kubernetes installation using Ansible).

To delete the cluster: kops delete cluster shiv.k8s.local

kops delete cluster shiv.k8s.local --yes

**PODs:**

Let’s deploy a pod in our Minikube cluster, as we discussed a Pod is the most basic and the smallest unit in Kubernetes.

1. kubectl run nginx --image=nginx [Pod name can be anything but the image name has to the same name of an image available on Docker, as we are trying to pull image from Docker Hub]
2. kubectl get pods [will list all the pods available]
3. kubectl describe pod nginx [describe give a detailed information of the specific item, here it is providing information about Pod]
4. kubectl get pods –o wide [this command gives information just like “get pod” but with some additional data along with that like “node” attach to it]

{Each pod that’s getting created will get an internal ip}

Reference Link: <https://kubernetes.io/docs/concepts/workloads/pods/>

To create a deployment using imperative command, use (kubectl create deployment nginx –image-nginx)

**Introduction to YAML:**

What is YAML?

A YAML file is used to represent data, in this case configuration data, let us see the quick comparison of sample data in 3 different formats (like XML [eXtensible Markup Language], JSON [JavaScript Object Notation] & YAML [YAML Ain’t Markup Language])

**Data representation:**

XML is a markup language, whereas JSON and YAML are data formats. XML uses tags to define the elements and stores data in a tree structure, whereas data in JSON is stored like a map with key/value pairs. YAML, on the other hand, allows representation of data both in list or sequence format and in the form of a map with key/value pairs. JSON and YAML uses different indentation styles: JSON uses tabs, whereas YAML uses a hyphen (-) followed by whitespace.

**Comments:**

Comments makes it easier to understand and interpret data. Whereas JSON has no concept of comments, XML allows you to add comments within a document. YAML was designed for readability and thus allows comments.

**Data types:**

XML supports complex data types such as charts, images, and other non-primitive data types. JSON supports only strings, numbers, arrays, Boolean, and objects. YAML, on the other hand, supports complex data types such as date and time stamps, sequences, nested and recursive values, and primitive data types.

**Data readability:**

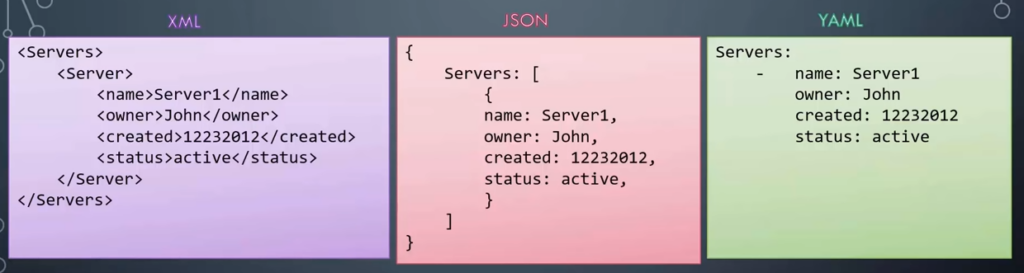
It is difficult to read and interpret data written in XML, but it is fairly easy to interpret data in JSON format, and it is much easier to read data in YAML than in JSON format.

**Usability and purpose:**

XML is used for data interchange (that is, when a user wants to exchange data between two applications). JSON is better as a serialization format and is used for serving data to application programming interfaces (APIs). YAML is best suited for configuration.

**Speed:**

XML is bulky and slow in parsing, leading to relatively slow data transmission. JSON files are considerably smaller than XML files, and JSON data is quickly parsed by the JavaScript engine, enabling faster data transmission. YAML, as a super-set of JSON, also delivers faster data transmission, but it's important to remember that JSON and YAML are used in different scenarios.



Let’s check YAML, if you take the data in its simplest form such as key value pair.



Here we can see the key and value are separated by a colon, Remember we need to give a space followed a colon differentiating the key and the value. Now, let’s take a look at how an array is represented,



In Array/lists, we can see fruits followed by a colon on the next line enter each item with a dash in the front, here the “dash” indicates that it’s an element of an array.

In Dictionary is a set of properties grouped together under an item, here we try to represent nutritional information of 2 fruits, notice the blank space before each item, and here we need to have equal number of blank spaces before the properties of a single item, so they are all aligned together.



Let’s learn about the spacing, notice the number of spaces before each property that indicates these key value pairs fall, but what if we have extra spaces for fat and carbs, then they will fall under calories and thus become properties of calories and which doesn’t make any sense, this will result in syntax error which will tell you that mapping values are not allowed here because calories already have a value set which is (105), you can either set a direct value or a hash map, but we cannot set both; So the number of spaces before each property is “key” in YAML, where we must be sure we are in the right form to represent the data correctly,



Let’s take it to another level you can have a list containing dictionaries containing lists, in this case I have a list of fruits and the elements of the list are banana and grape, but each of these elements are further dictionaries containing nutrition information.



When to use a Dictionary (or) Lists?

As we already know these formats used to represent data, that data can be anything like a school and their students along with grades or it could be an organization and their employees along with their personal details etc.,

Let’s take an example of a car, it is a single object and it has properties such as color, model, transition, and price, to store different information or properties of a single object, we use a Dictionary, in this simple we have a properties of the car defined in a key: value format.

**Dictionary**

Color: Blue

Model: Corvette

Transmission: Manual

Price: 7, 00,000

What if, we need to split the model further into model name and make either you can then represent this a as a dictionary within another dictionary. In this case, the single value of model is now replaced by a small dictionary with two properties name and year.

**Dictionary**

Color: Blue

Model:

Name: Corvette

Year: 1995

Transmission: Manual

Price: 7, 00,000

Let’s say we need to store the name of 6 cars, The name are formed by the color and the model of the car, to store this we would use a list or an array as it is multiple items of the same type of object since we are only storing the name we have a single list of strings.

**List**

* Blue Corvette
* Grey Corvette
* Red Corvette
* Green Corvette
* Black Corvette

What if we would like to store all information about each car, everything that we listed before such as the color, model, transition, and price, we will then modify the array from a list of strings to a list of dictionaries? So we expand each item in the array and replace the name with the dictionary we built earlier. This way we are able to represent all information about multiple cars in a single YAML file using a list of dictionaries.

**List of Dictionaries**

* Color: Blue

Model:

Name: Corvette

Model: 1995

Transmission: Manual

Price: $20,000

* Color: Grey

Model:

Name: Corvette

Model: 1995

Transmission: Manual

Price: $21,000

* Color: Red

Model:

Name: Corvette

Model: 1995

Transmission: Manual

Price: $22,000

* Color: Green

Model:

Name: Corvette

Model: 1995

Transmission: Manual

Price: $23,000

* Color: Green

Model:

Name: Corvette

Model: 1995

Transmission: Manual

Price: $22,000

**YAML – Notes:**

Dictionary is an unordered collection whereas lists are ordered collection,

SO, what does that mean?

**Dictionary/Map**



Banana: Banana:

Calories: 105 Calories: 105



Fat: 0.4g Carbs: 27g



Carbs: 27g Fat: 0.4g

The 2 dictionaries that you see here have the same properties for banana, however we can see that the order of properties fat and carbs do not match, but that doesn’t really matter, the properties can be defined in any order but the 2 dictionaries will still be the same.

**Array/List**

Fruits: Fruits:

* Orange - Orange



* Apple - Banana



* Banana - Apple

This is not the same for lists or arrays, Arrays are ordered collection, so the order of items matter, and this is something to keep in mind while working with data structures.

Also remember any line beginning with a Hash is automatically ignored and considered as a comment

**# List of Fruits**

**YAML in Kubernetes:**

Let’s create a Pod using YAML based configuration files, as we already discussed how to develop YAML files, let’s focus on developing YAML files specifically for Kubernetes. As K8s uses YAML files as inputs for the creation of objects such as POD’s, replicas, deployment, services, etc., All of these follows a similar structure, where this definition file always contains 4 top level fields, (apiVersion, kind, metadata & spec), in other words these are the root level properties, these are also required fields so you must have them in your configuration files, let’s see in detailed,

***apiVersion:*** This is the version of the kubernetes API you’re using to create the objects, depending on what we are trying to create we must use the right API version. For now, since we are working on PODs, we will set the API version as we want (v1), few other possible values for this filed are (apps/v1) (extension/v1Beta)

***kind:*** refers to the type of object we try to create which in this case happens to a (Pod), some of the other possible options are (replica set, deployment, service etc.,)

***metadata***: is data about the object like its name labels etc., as you can see unlike the first 2 where we have specified a string value, but metadata in the form of a Dictionary, so everything under metadata is intended to the right a little bit and so names and labels are children of metadata. The number of spaces before the 2 properties name and labels doesn’t matter but they should be the same as they are siblings. (For example: there are hundreds of pods running a front-end application and hundreds of pods running a backend application or a database, it will be difficult for you to group these pods once they are deployed; but by labeling them now as frontend, backend or database we will be able to filter the pods based on this label at the later point in time it’s important to note that under metadata you can only specify name or labels or anything else that kubernetes expects to be under metadata, we will not be able to add any other property as we wish under this. However, under labels you can have any kind of key or value pairs as you see fit. So, it’s really important to understand what each of these parameters expects. So far we have only mentioned that type and name of the object we need to create which happens to be a Pod with a name, but we haven’t really specified the container or image we need in the Pod.

***spec:*** thelast section in the configuration file is the specification section which is written as spec. depending on the object we are going to create, this is where we would provide additional information to kubernetes pertaining to that object, this is going to be different for different objects so it’s important to understand or refer to the documentation section to get the right format for each since we are only creating pod with a single container in it is easy, spec is a dictionary same like metadata, so add a property under it called containers. Containers is a list or an array. The reason this property is a list is because the pods can have multiple container within them as we learned earlier, in this case though we will only add a single item in the list since we plan to have only a single container in the pod that (-) before the name indicates that this is the 1st item in the list, the item in the list is a dictionary, so add a name and image property the value for image is nginx, which is the name of the docker image. Once the file is created from the command “kubectl create –f (followed by the file name which is) pod-definition.yml” and kubernetes creates the pod.



***kubectl create –f pod-definition.yaml***

***pod-definition.yaml***

apiVersion: v1 ------------ string

kind: Pod ------------ string

metadata:

name: myapp-pod

labels: --- Dictionary



app: myapp

type: front-end

spec:

containers: -------------- List/Array

- name: nginx-container



image: nginx



**Kind Version**

POD v1

Service v1

Replicaset apps/v1

Deployment apps/v1

Once we are clear with this basic structure, we can add values to those depending on the object we are going to create.

Steps:

1. Create a file (vim pod.yaml)

apiVersion: v1

kind: Pod

metadata:

name: mynginx

labels:

app: nginx

tier: frontend

spec:

containers:

- name: mynginx

image: nginx

1. Now we need to create the pod, (kubectl create –f pod.yaml)
2. Confirm the same by calling (kubectl get pods)
3. To get more details about the pod (kubectl describe pod mynginx)

**Controllers:**

Controllers are the brain behind kubernetes, they are the processes that monitor kubernetes objects and respond accordingly,

**Replication Controller and Replicasets:**

What is a replica and why do we need a replication controller? In our previous scenario, we have our application running on single pod, what if for some reasons our application crashes and the pod fails, users will no longer be able to access the application, to prevent users from losing access to our application, we would like to have more than one instance or pod running at the same time, that way, if one fails we still have out application running on the other one. The replication controller helps us to run multiple instances of a single pod in a kubernetes cluster, thus providing high availability, Even if you have a single pod, the replication controller can help by automatically bringing up a new pod when the existing one fails, Thus the replication controller ensures that the specified number of pods are running at all times.

Another reason we need replication controller is to create multiple pods to share the load across them, (for ex: let’s consider I have a single pod serving set of users, when the number of users increases we deploy additional POD to balance the load across the 2 pods, if the demand further increases and if were to run out of resources on the 1st node, we could deploy additional pods across the other nodes in the cluster) as you can see, the replication controller spans across multiple nodes in the cluster. It helps us balance the load across multiple pods on different nodes as well as scale our application when the demand increases.



**Difference between Replication controller and Replicaset:**

**ReplicationController:**

Even though the purpose are same, but they are not the same, where replication controller is the older technology that has been replaced by replicaset, but the above examples are same for both these technologies., when comes to replicas as the name itself telling they are going to follow (template) for proceed, hence in this situation we need to specify (template) under spec category in our .yaml file.

How to bring this template? It is very easy to do, because in our previous exercise we used a setup for our pod creation and we are going to follow the same here also mentioning which format/image/name they need to follow.

***Steps:***

1. Create a file (vi rc-definition.yaml)

apiVersion: v1

kind: ReplicationController

metadata: --------- Replication Controller

name: myapp-rc

labels:

app: myapp

type: front-end

spec: ---------- Replication Controller

template:

metadata: ---------- POD

name: myapp-pod

labels:

app: myapp

tier: front-end

spec: ----------- POD

containers:

- name: nginx-container

image: nginx

replicas: 3

[as we can see here we have 2 metadata sections one is for Replication controller & other is for Pod, same for spec section also; we have nested 2 definition files together, the replication controller being the parent and the pod definition being the child, it is important to mention the number of replicas need to be created, now we need to add another property under replicas and add the number of replicas required, remember the replicas and templates are direct children of spec section]

1. Create the file (kubectl create –f rc-definition.yml)
2. To know the list of replicationcontrollers (kubectl get replicationcontroller), as we can see the result of (3)
3. Now run the (kubectl get pods) command to see the result of how many pods created by replicationcontroller

**Replicaset:**

***Steps:***

1. Create a file (vi replicaset-definition.yml)

apiVersion: apps/v1

kind: ReplicaSet

metadata:

name: myapp-replicaset

labels:

app: myapp

type: front-end

spec:

template:

metadata:

name: myapp-pod

labels:

app: myapp

type: front-end

spec:

containers:

- name: nginx-container

image: nginx

replicas: 3

selector:

matchLabels:

type: front-end

[if you are going to mention (v1) for apiVersion, then we are going to receive an error like (unable to recognize “replicaset” definition.yml”: no matches for /, Kind=ReplicaSet); all the other things are same, but at the end we can see a line called “selector” = this section helps the replica set to identify what parts fall under it, [but why would you have to specify what parts fall under it if you have provided the contents of the definition file itself in the template?; it’s because replica set can also manage pods that were not created as part of the replica set creation; say for example, the reports created before the creation of the replica set that match labels specified, the replica set will also take those pods into consideration when creating the replicas] In replica set, a user input is required for this property and it has to be written in the form of match labels as shown here. This label selector simply matches the labels specified under it to the labels on the pod. And this is available only for replicaset and not for replication controller]

1. Create the file (kubectl create –f replicaset-definition.yml)
2. To get the list, give (kubectl get replicaset)
3. Now run the (kubectl get pods) command to see the result of how many pods created

Why do we need to label our objects? And what is the deal with labels and selectors?

Say we deployed 3 instances of our front end web application as 3 parts, we would like to create a replication controller or replica set to ensure that we have 3 active pods at any time, and that is the use case of replicaset; we can use it to monitor existing pods if you have them already created; in case they haven’t created the replicaset will create them for us. The role of the replicaset is to monitor the pods and if any of them were to fail deploy new ones.

How the replicaset does knows what pods to monitor?

This is where labeling our pods during creation comes in handy, we could now provide these labels as a filter for a replica set; under the selector section we use the match labels filter and provide the same label that we used while creating the pods. These way the replica set knows which pods to monitor.

How to Scale the replicas?

There are multiple ways to do that same, 1st way is to update the definition file from 3 to 6; then by running the (kubectl replace –f replicaset-definition.yml)

2nd way to run the scale command (kubectl scale –replicas=6 –f replicaset-definition.yml)

**DaemonSets:**

This is like a Replicasets, as it helps you deploy multiple instances of pods. But it runs one copy of your pod on each node in your cluster. Whenever a new node created a replica of the pod is automatically added to that node, and the node removed automatically the pod is also removed. The Daemonset ensures that one copy of the pod is always present in all nodes in the cluster.

Use cases:

Consider, we need to deploy a Monitoring agent or log collector on each of your nodes in the cluster, so we can monitor our cluster better, a Daemonset is perfect for that as it can deploy the monitoring agent in the form of a pod in all the nodes in the cluster, then we don’t want to worry about adding or removing monitoring agents from these nodes when there are changes in your cluster as the Daemonset will take care of that.

Steps:

1. Create a file (vi daemonset.yml)

apiVersion: apps/v1

kind: DaemonSet

metadata:

# Unique key of the DaemonSet instance

name: mynode-exporter

labels:

app: mynode-exporter

spec:

selector:

matchLabels:

app: mynode-exporter

template:

metadata:

labels:

app: mynode-exporter

spec:

containers:

# This container is run once on each Node in the cluster

- name: mynode-exporter

image: prom/node-exporter:v0.18.1

ports:

- containerPort: 9100

hostPort: 9100

protocol: TCP

1. Create the file (kubectl apply –f daemonset.yaml)
2. To get the list, give (kubectl get daemonset)
3. Now run the (kubectl describe daemonset) to know more details about that specific one.

How does a Daemonset works? How does it schedule pods on each node? And how does it ensure that every node has a pod?

The daemonset uses a default scheduler and NodeAffinity rules to schedule pods on nodes.

# [Node Affinity

Node affinity is a set of rules the Kubernetes scheduler uses to determine where a pod can be placed. It is similar to the nodeSelector parameter but offers more flexibility and functionality.]