Managing Kubernetes Controllers and Deployments
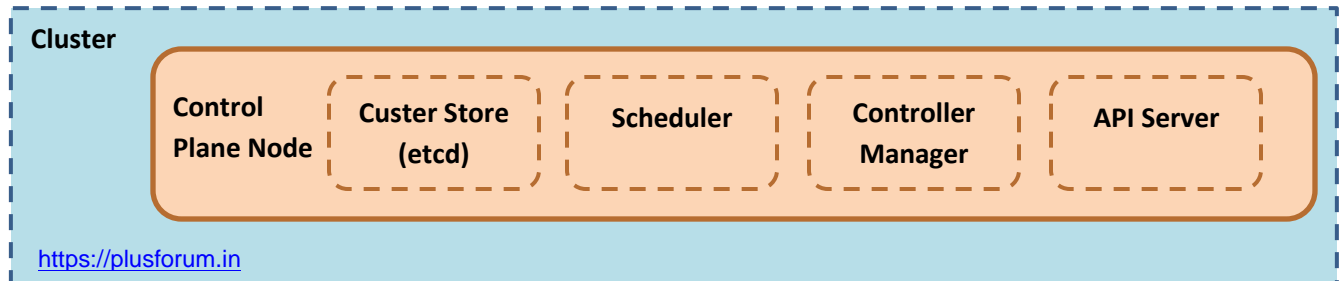
Kubernetes Principles:

1) Desired state declarative configuration:
2) Controllers Control Loops
3) The API server

Below mentioned are the components running on a control plane Node.

Cluster

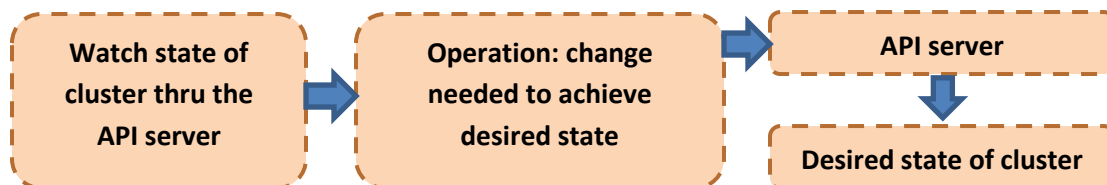| Control Plane Node | Custer Store (etcd) | Scheduler | Controller Manager | API Server |

The controller manager makes sure that the desired state of the cluster is maintained using the API server.

1) Kube-controller-manager: responsible for maintaining the resided state in the cluster. There's always only one Kube-controller-manager per cluster. Control loops that run inside the manager help in deploying the deployment, service, replica sets resources in the cluster.
2) Cloud-controller-manager: this manager helps the cloud service providers to manage the resources in cloud specific kubernetes environment and manage in the cloud centric environment.

What are controller operations?

1) Watch state: the controller watches the cluster state through API server and changes the current state of the cluster into the desired state. The change is also implemented thru the API server.

Watch state of cluster thru the API server	→	Operation: change needed to achieve desired state	→	API server
				↓
				Desired state of cluster

Two main controller types that help to achieve desired state for workload PODs.
POD controllers:

1) **ReplicaSet:** maintains the desired number of PODs using POD template in a cluster.
2) **Deployment:** manage different versions of POD templates and easily scale and deploy new version. This is core controller process that runs in cluster.
3) **DaemonSet:** All or some nodes run a POD that is used in cluster management. Example is the **kube-proxy service**, **monitoring** and **log collection agent** services that runs inside PODs that are not workload related POD.
4) **StatefulSet:** StatefulSet is the workload API object used to manage stateful applications.
   Manages the deployment and scaling of a set of Pods, and provides guarantees about the ordering and uniqueness of these Pods. Like a Deployment, a StatefulSet manages Pods that are based on an identical container spec. Unlike a Deployment, a StatefulSet maintains a sticky identity for each of their Pods. These pods are created from the same spec, but are not interchangeable: each has a persistent identifier that it maintains across any rescheduling.
5) **Job**: A Job creates one or more Pods and will continue to retry execution of the Pods until a specified number of them successfully terminate. As pods successfully completes its lifecycle, the Job tracks the successful completions. When a specified number of successful completions occur, the task (i.e., Job) is complete. Deleting a Job will clean up the Pods it created. Suspending a Job will delete its active Pods until the Job is resumed again.
6) **CronJob**: A CronJob creates Jobs on a repeating schedule.
   One CronJob object is like one line of a crontab (cron table) file. It runs a job periodically on a given schedule, written in Cron format.

Controller to manage resources other than POD.
1) **Node:** A controller to manage the state of each node in the cluster. Is the Node up or down and the state of resources on the node.
2) **Service:** creation and deletion of Load Balancers in the cluster infrastructure.
3) **Endpoint controller:** managed the service controller to connect to the PODs based on the labels.
4) Many more controllers like **NameSpace, Route controllers, PersistentVolume** controller are available.

To start the Practical part, start the Kubernetes cluster:

| Kubectl | Control Plane Node (master) C1-CP1 | Custer1 – node1 | Custer1 – node2 |
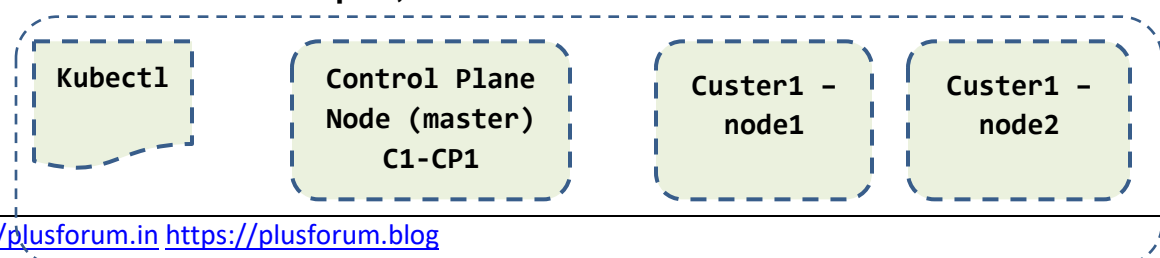
Start with examining the system PODs in cluster, run below command,

```
$ kubectl get  --namespace kube-system all
```

Now this will give list of all resources running under kube-system namespace.

Let's now get details about the 'coredns' deployment.

```
$ kubectl get --namespace kube-system deployment coredns
```

Now this will give details about 2 coredns PODs running under this deployment.

Now let's look at some of the 'DaemonSet' running in the cluster.

```
$ kubectl get –namespace kube-system daemonset
```

Daemonset will run PODs that run services needed for the cluster to run an all nodes in the cluster.

For example the **kube-proxy resource** will run on all nodes in the cluster.

The Deployment Controller:

➢ Managing Application state with Deployments
  1) Creating PODS in cluster
  2) Updating application and PODs
  3) Scaling up/down number of PODs in the cluster.

Declaratively: Preferred way to create deployment resource**.**

- Writing a deployment Spec in Code (YAML).
  ▪ Selector: defines which POD to be deployed using deployment object.
  ▪ Replicas: number of replicas of POD to be deployed
  ▪ POD Template: technical details like container image, Volume to be mapped to the POD to be deployed.

Imperatively: Define and deploy deployment resource at command line. This may not be the preferred way but a quick way to get the application running in cluster. Example:

```
$ kubectl create deployment hello-world --
image=ganeshhp/helloworld:latest
```

Using above command, we can create the deployment object with one replica. As we have not passed the replicas parameter with value.

To scale the deployment replicas, we can use the command,

```
$ kubectl scale deployment hello-world --replicas=2
```

`$ kubectl get deployment` … to list deployment resources in the cluster

`$ kubectl delete deployment hello-world` … in case we want to remove / delete the resource, we use delete command in kubectl options.

Declaratively: write a manifest file as shown below.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: hello-world
spec:
 replicas: 4
 selector:
     matchLabels:
            app: hello-world
  template:
     metadata:
         labels:
             app: hello-world
     spec:
         containers:
          - name: hello-world
            image: ganeshhp/helloworldweb:latest
            ports:
             - containerPort: 8080
--------
apiVersion: v1
kind: Service
metadata:
    name: hello-world
    labels:
        app: hello-world
spec:
    selector:
        app: hello-world
    ports:
     - port: 80
        protocol: TCP
```

```
        targetPort: 8080
```
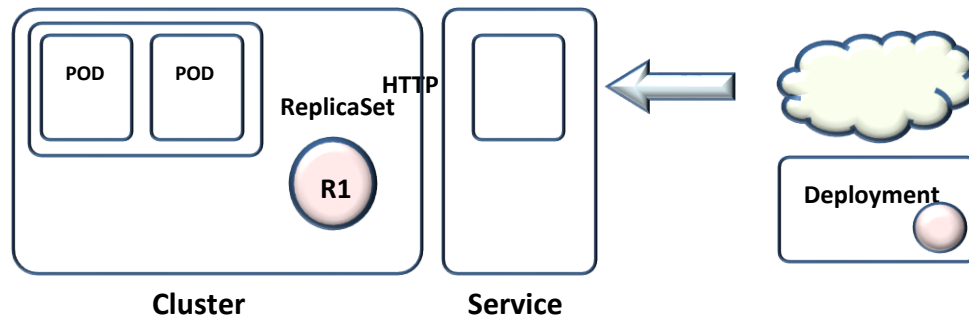
Now, to deploy the deployment resource declaratively as written in manifest file, use the command,

`$ kubectl create –f deployment.yml` ... here we can use create or apply command in kubectl.

As the manifest file has deployment and service resource definitions, we will get both resources deployed with create action.

`$ kubectl get deployments` .. To see deployment resources status

```
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
hello-world   1/1     1            1           3d18h
```

`$ kubectl get all` ..

This will show all available resources deployed in default namespace.

If we look at the description of a POD that's created using Deployment / ReplicaSet, we can see the information about resource that is controlling the POD.

https://plusforum.in

```
$ kubectl get replicasets
```

Command to see ReplicaSet resource created by the deployment object.
ReplicaSet is responsible for keeping the number of PODs created and
available.

```
NAME                      DESIRED   CURRENT   READY   AGE
hello-world-6f8d5c6d89    1         1         1       3d18h
```

```
$ kubectl describe pod --namespace-default | head –n=15
```

```
Name:          hello-world-6f8d5c6d89-rdb2c
Namespace:     default
Priority:      0
Node:          minikube/192.168.99.100
Start Time:    Sun, 19 Dec 2021 00:01:07 +0530
Labels:        app=hello-world
               pod-template-hash=6f8d5c6d89
Annotations:   <none>
Status:        Running
IP:            172.17.0.3
IPs:
  IP:          172.17.0.3
Controlled By:  ReplicaSet/hello-world-6f8d5c6d89
Containers:
```

Understanding Replicasets:

What does the ReplicaSet provides:

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

1) Deploys a defined number of PODs to maintain the desired state of PODs in the cluster.
2) **Consists of Selector, Number of Replicas, and POD templates**
3) As a practice we do not create ReplicaSet directly, but instead is created by the deployment resource while we define number of replicas in the Deployment resource definition. It is the underlying building block of deployment object.
4) As part of ReplicaSet definition, we can use different type of selectors. As shown below. *matchLabels* and *matchExpressions*

```
apiVersion: apps/v1
kind: ReplicaSet
...
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-world-pod
  template:
    metadata:
      labels:
        app: hello-world-pod
    spec:
      containers:
      ...
```

```
apiVersion: apps/v1
kind: ReplicaSet
...
spec:
  replicas: 1
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
          - hello-world-pod-me
  template:
    metadata:
```

How ReplicaSet handled failure.
While in the cluster if there's a POD failure, the ReplicaSet send a request to the APIserver to terminate the failed POD and create a new POD in that place.
In case if the Node failure happens, this is treated in a different way. The Node failure can be a transient failure or a permanent failure. Transient failure can be seen as a brief unavailability of the node due to network or hardware issue. If the node is not reachable then the Containers running on that node are assigned with error status. If the nodes come up again in some time and if the POD / Containers are not running, then the **POD restart policy** is applied using the kube-controller manager and the PODs are restarted as the PODs are still scheduled on that Node.

Now if the Node is down for a longer time, intentionally taken down then the Kube-Controller manager has a pod-eviction-timeout setting that is set to 5 minutes default value. After 5 minutes the PODs on the failed node are unscheduled and recreated on available Node in the cluster to match the desired state of POD replicas.

Demo for ReplicaSet:

We have a YAML file (*as stated in this document on page 41*) that has the Deployment and Service object defined. Let's deploy the objects with create command.

```
$ kubectl create -f deployment.yaml
```

```
$ kubectl get replicaset
```

```
NAME                       DESIRED   CURRENT   READY   AGE
hello-world-5646fcc96b     5         5         5       14s
```

```
$ kubectl describe replicaset hello-world
```

```
Name:           hello-world-5646fcc96b
Namespace:      default
Selector:       app=hello-world,pod-template-hash=5646fcc96b
Labels:         app=hello-world
                pod-template-hash=5646fcc96b
Annotations:    deployment.kubernetes.io/desired-replicas: 5
                deployment.kubernetes.io/max-replicas: 7
                deployment.kubernetes.io/revision: 1
Controlled By:  Deployment/hello-world
Replicas:       5 current / 5 desired
Pods Status:    5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
  Labels:   app=hello-world
            pod-template-hash=5646fcc96b
```

Now after checking the details about the replicaset object, let's delete the Deployment object which in turn will remove the ReplicaSet object as well.

Now, after deleting the Deployment object let's **create a new one**, but this time with `matchExpressions` instead of `matchLabels.`

```
$ kubectl create -f deployment-me.yaml
```

```
--------

  apiVersion: apps/v1
  kind: Deployment
  metadata:
      name: hello-world
  spec:
```

```
    replicas: 4
    selector:
        matchExpressions:
              - key: app
                operator: In
                values:
                     - hello-world-pod-me
    template:
        metadata:
            labels:
              app: hello-world
        spec:
            containers:
             - name: hello-world
               image: ganeshhp/helloworldweb:latest
               ports:
                - containerPort: 8080
```

Let get details of the depoyment object.

`$ kubectl describe deployment/hello-world`

```
Name:          hello-world-f597dc95
Namespace:     default
Selector:      app in (hello-world-pod-me),pod-template-hash=f597dc95
Labels:        app=hello-world-pod-me
               pod-template-hash=f597dc95
Annotations:   deployment.kubernetes.io/desired-replicas: 5
               deployment.kubernetes.io/max-replicas: 7
               deployment.kubernetes.io/revision: 1
Controlled By: Deployment/hello-world
Replicas:      5 current / 5 desired
Pods Status:   5 Running / 0 Waiting / 0 Succeeded / 0 Failed
Pod Template:
   Labels:  app=hello-world-pod-me
            pod-template-hash=f597dc95
   Containers:
    hello-world:
      Image:        gcr.io/google-samples/hello-app:1.0
      Port:         8080/TCP
      Host Port:    0/TCP
      Environment:  <none>
```

Now to see if the replicaset creates a POD in place of a failed / deleted POD,
try deleting a POD from the running list of PODs. Also try changing Label of a
running pod by using below command and note how replicaset creates a new
PODs in the place of non matching POD.

```
$ kubectl label pod hello-world-[tab]-[tab] app=DEBUG –
overwrite
```

Now check if the labels are changed, by running below command,

```
$ kubectl get pods --show-labels
```

Now as soon as a POD has a POD with different label, not matching the label to the replicaset, then a new POD will get created to match required number of PODs matching the labels.

Now let's change the Label back again to original state for the POD, we will see that to match the number of PODs in cluster matching the Label, one of the POD will get deleted if required.

Node Failure Operation:

Now what if a node fails in the cluster.. check that with forcefull stopping a Node and observing changes in the cluster.

For this we would need atleast two worker nodes in the cluster.

1) Shutdown one of the node of the worker node in the cluster.

```
$ shutdown –h now
```

While the node is shutting down, we can watch the status of cluster using,

```
$ kubectl get nodes --watch
```

This will report the node status as 'NotReady'.

```
NAME      STATUS      ROLE       AGE        VERSION
node1     NotReady    <none>     31d        v1.13.1
```

Now while the node is shutdown, the Controller still treats this as a **transient error** and waits for 5 minutes (default) time till the node comes back again. During this time if the node doesn't come back the POD status will report below.

NAME	READY	STATUS	RESTARTS	AGE
hello-world-f597dc95-fcs8z	0/1	Error_	0	6m55s

After some time when the node recovers and is reachable within the timeline of 5 minutes, we will see the POD getting back again after the controller attemps a restart.

NAME	READY	STATUS	RESTARTS	AGE
hello-world-f597dc95-fcs8z	1/1	Running	1	7m21s

Now let's see if the nodes is permanently shutdown and doesn't come back in the 5 minutes `(pod-eviction-timeout)` timeline window. The POD will get deleted from the unavailable Node and a new POD will get deployed on the available POD to match the required replica numbers.

Deploying and Maintaining Applications with Deployment object.

1) Updating `Deployments`
2) Controlling Rollouts
3) Scaling Applications
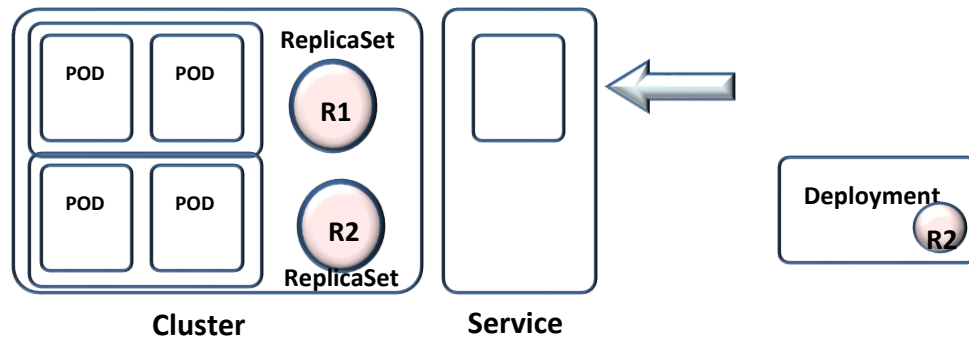
The Deployment object helps in updating the application without compromising the application availability.When the specification of the POD templates changes, the update is triggered and causes a new ReplicaSet is create and this makes sure to delete and remove a POD from earlier replicaset and create new set of PODs with new replicaset.

Updating deployment object:

Now, to update already deployed deployment object, we can use different ways, first we can use **set image** command.

```
$ kubectl set image deployment hello-world hello-world=web-app:2.0
```

Object type and object name   Image name

We also add a `--record` flag to this command to add annotation, which can later be used for comparing earlier version of deployment o newer versoin and then help in rollback of the deployment.

```
$ kubectl set image deployment hello-world hello-world=new-app:2.0 --record
```

Similarly we can use edit command to edit the object that is defined in a manifest file.

```
$ kubectl edit deployment hello-world
```

With this command the api-server is presented with request to get details of the object and open that in an editor.

And finally, we have the `kubectl apply -f` command to apply changes to existing object using changes mentioned in manifest file. We can export defination of an object into a YAML file and then edit the YAML file and send it back to the API-Server.

```
$ kubectl apply -f hello-world-deployment.yaml --record
```

Checking deployment status

While we are updating a deployment object, we can check the status of update by using **rollout** and **describe** command option.

```
rollout Available Commands:
  history     View rollout history
  pause       Mark the provided resource as paused
  restart     Restart a resource
  resume      Resume a paused resource
  status      Show the status of the rollout
  undo        Undo a previous rollout
```

```
$ kubectl rollout status deployment hello-world
```

```
$ kubectl describe deployment hello-world
```

Deployment status.

1) `Complete` – all update work in finished
2) `Progressing` – update in progress
3) `Failed` – update could not be completed.

When we are deploying a new version of deployment (image version) using declarative way, when we deploy the YAML file, it creates a new ReplicaSet and stop all PODs started by older ReplicaSet. Both ReplicaSets will continue to exists till we remove it manually. We can use this for undoing new deployment if needed. Use below command,

```
$ kubectl get replicaset
```

```
NAME                      DESIRED   CURRENT   READY   AGE
hello-world-54875c5d5c    10        10        10      6m41s
hello-world-5646fcc96b    0         0         _ 0     9m21s
```

Now try getting information on the new replicaset

```
$ kubectl describe replicaset hello-world-54875c5d5c
```

Also try getting description for the earlier replicaset. This should show POD status as deleted.

Using Deployments to Change State:

Control rollouts of a new versoin of your applications.

- Update strategy
- Pause rollout to update and make correction toan object
- Rollback to an earlier version
- Restart deployments

Update Strategy:

RollingUpdate (default)

- A new ReplicaSet starts scaling up and,
- old ReplicaSet start scaling down.

Recreate

Terminates all PODs in the current ReplicaSet and Set priot to scalling up new Replicas Set. When application doesn't support running different version concurrently.

RollingUpdate strategy:

maxUnavailable … ensures only a certain number of PODs are made unavailable. *(Default is 25%)*

maxSurge… ensures only a certain number of PODs are new created above the number of desired state. (*Default us 25%)*

Controlling Deployment Rollout

With using **rolling update strategy** and POD **readyness probe** defined, we can controll the deployment of POD in a deployment object.

```
apiVersion: apps/v1
kind: Deployment
---
specs:
   replicas: 20
     strategy:
         type: RollingUpdate
             rollingUpdate:
```

```
              maxUnavailable:
                 maxSurge: 5
        ---
      template:
        ---
        spec:
           containers:
        ---
              readinessProbe
                 httpGet:
                     path: /index.html
                     port: 8080
                 initialDelaySeconds: 10
                 perdiosSeconds: 10
```

Pausing and Resuming a Deployment:

1) While the deployment is paused, changes are not rolled out.
2) We then can batch multiple changes together and then resume the deployment
3) Until the deployment is paused, the current state of deployment is maintained
4) When we resume the deployment and if we have added new changes to the deployment, then a new Replicaset is created and deployed with resumed deployment.
   Imperatively this can be done by running command,

```
$ kubectl rollout pause deployment my-deployment
```

```
$ kubectl rollout resume deployment my-deployment
```

Rolling back a Deployment

1) In case there's application failure after rolling out a new version and in order to recover from failure, we have only option to revert back the changes pushed to the enviroment, then we can use the **rollback** option to rollback the deployment. This is possible as Kubernetes maintains a history of all deployment.
2) In kubernetes deployment, every deployment object rolled-out has a annotation. This is maintained as the history.
3) By default the revision history is maintained for 10 deployment revisions. This value can be changed in deployment configuration. The revision history number to be set to 0 as well, this will help to clean up all deployment revision info as soon the deployment is created.

4) To perform a rollback, we need to have information about revision history and this can be quried by running command,

```
$ kubectl rollout history deployment my-deployment
```

Once we get the information, we want to know details about which application is rolled out in a particular version, then we can get detailed info with below command.

```
$ kubectl rollout history deployment my-deployment --revision=1
```

Now to perform the rollback of the deployment to earlier revision we use command,

```
$ kubectl rollout undo deployment my-deployment
```

To perform rollback of deployment to a particular version, we use below command,

```
$ kubectl rollout undo deployment my-deployment --to-revision=1
```

In this case kubernetes **scale up** the old replicaset and **scale-down** the new replicaset for the revision selected

Restarting a deployment:

Incase we want to restart the PODs in a deployment object, may be due to some environment change etc., we can use the restarr option in the command as show below.

```
$ kubectl rollout restart deployment my-deployment
```

In this case the PODs are not actually restarted, but the restart option causes a new replicaset to create with same POD definition, and scale down old replica set.

Demonstration:

To demonstrate the deployment rollback, we can use a deployment manifest to rollout a particular version. Then we make changes to the manifest with a broken container image, causing the container to fail. To revover from this failed rollout we use rollback strategy as show above.

Create a file `deployment-probe.yaml`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-world
spec:
  replicas: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 10%
      maxSurge: 2
  revisionHistoryLimit: 20
  selector:
    matchLabels:
      app: hello-world
  template:
    metadata:
      labels:
        app: hello-world
    spec:
      containers:
      - name: hello-world
        image: httpd:latest
        ports:
        - containerPort: 8080
        readinessProbe:
          httpGet:
            path: /index.html
            port: 8080
          initialDelaySeconds: 10
          periodSeconds: 10
```

```
$ kubectl apply –f deployment-probe.yaml –record
```

Using record we can keep the revision history along with annotation and thus using the revisoin history we roolback the deployment rolledout.

https://plusforum.in

Scaling Deployments:

o   Manually… by specifying number of replicas to scale to.

```
$ kubectl scale deployment hello-deploy –replicas=10
```

Or,

```
$ kubectl apply —f deployment-yaml
```

o   Horizaontal Pod Autoscaler

```
$ kubectl autoscale deployment/hello-deploy --max=5 --min=2 --
cpu-percent=80
```

Demonstration:

```
$ kubectl create deployment hello-world --
image=ganeshhp/hello-world:latest
```

```
$ kubectl get deployment hello-world
```

```
$ kubectl scale deployment hello-world —-replicas=10
```

```
$ kubectl get deployment hello-world
```

Now, the same change can be achieved declaratively as well, by updating the manifest (yaml) file. In the spec section of deployment object, we have replicas set to 10, we change the number to 15. If we run this on environment by using,

```
$ kubectl apply —f deployment-replicas-15.yaml
```

After running the above command, we check the status by running `kubectl get deployments` command.

Also check the status by running, `describe` command on the deployment.

Deployment Tips – how to control:

1) Control the rollout with an update strategy appropriate for your application.
2) Use Readyness Probes for your application.
3) Use the `--record` option to leave a trail of work for others.

Some additional Controller types in Kubernetes:

Deployment and Maintaining application using

1) **DaemonSets**
2) **Jobs and CronJobs controllers.**
3) **StatefulSets**

What's a **DaemonSet**: A DaemonSet is an api in kuberetes that ensures a single POD always runs on a node or subset of Nodes. These are like kubernetes managed system daemons which is like **systemd** or **initd** in Linux.

A **Job** is a Kubernetes API that ensure to run a POD or a task once inside the cluster.

To schedule a **Job** in Kubernetes cluster we have **CrobJobs** as api object in kubernetes to run the job at a specified schedule.

StatefulSets is a api object in kubernetes that provides required infrastructure to run a statefull application. **StatefulSets** are valuable for applications that require one or more of the following.

- Stable, unique network identifiers.
- Stable, persistent storage.
- Ordered graceful deployment and scaling.
- Ordered, automated rolling updates.

1) Daemonsets:

This API ensures that all or some nodes run a POD, effectovely an init daemon inside your cluster.

Examples,

- Kube-proxy for network service
- Log collectors
- Metric servers
- Resource monitoring agents
- Storage daemons

Defining a daemonset:

DaemonSet.yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
    name: hello-world-ds
spec:
```

```
        selector:
          matchLabels:
             app: hello-world-app
        template:
          metadata:
            labels:
               app: hello-world-app
          spec:
            containers:
              -  name: hello-world
                 image: gcr.io/google-sample/hello-app:1.0
```

Defining a DaemonSet with a **node selector** so that the daemonset POD can run on a specific Node or set of nodes.

DaemonSet-node.yaml

```
      apiVersion: apps/v1
      kind: DaemonSet
      metadata:
         name: hello-world-ds
      spec:
         selector:
           matchLabels:
             app: hello-world-app
         template:
           metadata:
             labels:
               app: hello-world-app
           spec:
             nodeSelector:
               node: hello-world-ns
             containers:
               -  name: hello-world
                  image: gcr.io/google-sample/hello-app:1.0
```

Here the label needs to be assigned to the Nodes on which we want the daemonset POD to be running and same is used in nodeSelector statement.

DaemonSet Update strategy:

Default update strategy is **RollingUpdate** and default maxUnavailable integer is **1.**

OnDelete update strategy will cause the Daemonset to be updated to new template only on deletion of the earlier version. If we want tp ugrade the

Daemonset to a new version, update the manifest YAML with desired chnages and Daemonset will push the changes to the cluster allwoing the desired state configuration to be maintained. In this there is no option to pause the upgradation of daemonset like the one in deployment api abject.

Demo:

In this demo we will creata **DaemonSet** with one **POD** on each **Node** and then we will create one POD on selected Node.

First, let's get the list of available Nodes in the cluster.

```
$ kubectl get nodes
```

```
NAME             STATUS   ROLES                  AGE   VERSION
docker-desktop   Ready    control-plane,master   10d   v1.22.5
```

Now we will also check the default **DaemonSet**, i.e. the **kube-proxy** that runs in the cluster. This can be checked by running command,

```
$ kubectl get daemonsets --namespace kube-system kube-proxy
```

```
NAME        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-proxy  1         1         1       1            1           kubernetes.io/os=linux   10d
```

Here, we are quering the `kube-system` namespace for the `kube-proxy` daemonset.

Now let's deploy a Daemonset using yaml file.

```
$ kubectl create -f DaemonSet.yaml
```

Verify daemonset status

```
$ kubectl get daemonsets
```

Verify where the daeonset PODs are running.

```
$ kubectl get daemonsets -o wide
```

```
$ kubectl get pods -o wide
```

Now, to get information on Callout, labels, desired/currenmt state Nodes scheduled, Pod status, template, events.

```
$ kubectl describe daemonset  hello-world-ds
```

Now let's try to change the Label for a POD and observer if the Daemonset creates a new POD with matching Label.

```
$ kubectl label pods <podname> app=not-hello-world-ds --overwrite
```

After we run this command, you will notice one new POD been created in place of the changed label POD., so now there are more than one PODs running on the node.

Now, lets try creating **DaemonSet** on only a subset of nodes and not on all worker nodes.

First remove all Daemonsets from earlier run. And also remove the POD that was taken out of Daemonset.

```
$ kubectl delete –f DaemonSet.yaml
```

Creating DaemonSet on subset of Nodes:

```
$ kubectl apply –f DaemonSet-node.yaml
```

The file is updated with a NodeSelector statement. After running this DaemonSet you will notice that the POD is not scheduled yet. This means we need to Label the Nodes as well to match in the **nodeselector**.

In this case we will label the node docker-desktop with the matching Label as `app:hello-world-ns`

```
$ kubectl label node docker-desktop app:hello-world-ns
```

Now, let's chake the status of the daemonset.

```
$ kubectl get daemonsets
```

```
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR         AGE
hello-world-ds  1         1         1       1            1           node=hello-world-ns   60s
```

Here, we see the daemonset is running and available.

Now if we label the node back again with label not macthing to the daemonset label, let's see what happens.

```
$ kubectl label node docker-desktop node-
```

The POD will get deleted as the matching node is not found in the cluster. The same can be seen with the daemonset describe command.

```
$ kubectl describe daemonsets hello-world-ds
```

```
Events:
  Type    Reason           Age    From                 Message
  ----    ------           ---    ----                 -------
  Normal  SuccessfulCreate  67s   daemonset-controller  Created pod: hello-world-ds-qfd5r
  Normal  SuccessfulDelete  5s    daemonset-controller  Deleted pod: hello-world-ds-qfd5r
```

Now, let's see how we can **update a daemonset** using update strategy. In this case the yaml file has been updated with a new image imfo and deployong a new daemonset is done uwing running the yaml file.

```
$ kubectl apply –f DaemonSet-update.yaml
```

```
$ kubectl describe daemonsets hello-world.
```

In this file we haven't mentioned the update strategy. The default **daemonset** update strategy is **rollingupdate**. We can see that by running a command, check the events to see the details.

```
$ kubectl get Daemonsets hellow-world-ds –o yaml | more
```

After we complete all activities related to demonstrating Daemonset, clear the daemonsets using

```
$ kubectl delete –f daemonset-ns.yaml
```

2) JOBs:

Controller objects so far introduced `(ReplicaSet, Deployment, DaemonSet)` allow to start and run the PODS in the cluster continuously and stay in that state to maintain the Desired State Configuration (DSC).

But there are requirements when we do want to run a task just once or periodically, but not continuously like running a POD in cluster continuously, for this we have the `Job` or `CronJob` objects.

- **Jobs** create one or more Pods,
- Runs a program in a container to completion.
- Ensures that a specified number of Pods completes successfully.
- Workload exmaples.
    o Ad-hoc tasks

- o Batch job to be executed inside the cluster
- o Data oriented tasks like moving data from system**A** to System**B.**

The Job controller object helps to run the POD to completion. In case of a interrupted execution the POD, like incase of a node failure, then the JOB controller has the task to reschedule the POD to a different node. Or if the applicatin running inside the POD returns a Non-Zero exit code, then the Job controller kicks in the the POD **restartPolicy**. The default restartpolicy is **Always**, we should change itto '**OnFailure'.**

If the Job completes successfully, the status is set to 'Completed'.

https://plusforum.in

Let's write a yaml file`, job.yaml`

```
apiVersion: batch/v1
kind: Job
metadata:
    name: hello-world-job
spec:
    template:
      spec:
        containers:
        - name: ubuntu
          image: ubuntu
          command:
          - "/bin/bash"
          - "-c"
          - "/bin/echo Hello From Pod $(hostname) at $(date)"
          restartPolicy: Never
```

Controlling the JOB execution:

1) backOffLimit – number of job retries before it's marked as 'failed'.
2) activeDealineSeconds: max execution time for the job
3) parallelism – max number of runnning PODs in a job at a point fo time.
4) Completion - number of PODs that need to finish successfull.

Demonstration:

Let's first set the '`restartPolict` to `OnFailure`'

`$ kubectl apply –f job.yaml`

Follow the job status with a `watch`

```
$ kubectl get job --watch
```

```
NAME              COMPLETIONS   DURATION   AGE
hello-world-job   0/1           9s         9s
hello-world-job   1/1           22s        22s
```

```
$ kubectl get pods
```

```
NAME                         READY   STATUS      RESTARTS   AGE
hello-world-job--1-rj9tw     0/1     Completed   0          3m28s
```

This indicates, the Pods started by the task completed successfully.
Try to run the describe command on the pod created by the Job.

```
$ kubectl describe job hello-world-job
```

```
Events:
  Type    Reason          Age   From            Message
  ----    ------          ----  ----            -------
  Normal  SuccessfulCreate 61s  job-controller  Created pod: hello-world-job--1-rj9tw
  Normal  Completed        39s  job-controller  Job completed
```

```
$ kubectl logs hello-world-job--1-rj9tw
```

```
PS C:\Users\Lenovo> kubectl logs hello-world-job--1-rj9tw
Hello from POD hello-world-job--1-rj9tw at Sun Feb 27 14:29:05 UTC 2022
```

So from this demonstration we are able to see that the job helped us to get a
POD started with a continer running specific commands and after the container
completed execution of the commands, the container stops in other words the
Pod completes its lifecycle.
As we have set the restartPolicy value to 'Never', the POD doesn't restart on
completion. If we set this to OnFailure, we can expect the POD to restart
automatically on failed POD in the JOB.

A Job can also be executed with parallel option allowing multiple PODs to run in
parallel.

paralleljob.yaml

```
apiVersion: batch/v1
kind: Job
metadata:
      name: hello-world-job
```

```
spec:
      completions: 50    # allows us to run 50 number of PODs in all as part of the JOB
      parallelism: 10    # allows to run 10 PODs in parallel
      template:
        spec:
          containers:
          - name: ubuntu
            image: ubuntu
            command:
            - "/bin/bash"
            - "-c"
            - "/bin/echo Hello From Pod $(hostname) at $(date)"
          restartPolicy: Never
```

$ kubectl apply –f paralleljob.yaml # this will start creating 10 PODs as part of the job.

CronJobs:

- CronJob will run a Job on a given time based schedule
    - o Concentually CronJob is cimilar to Linux / Unix Cron utility.
    - o It used the standard cron format.

Example workloads –

1) Periodic workload or scheduled tasks
2) CronJob resource is ceated when the object is submtted to the API server
3) When it's time, a Job is crated via the Job template from the CronJob Object.

Controlling CronJob Execution:

- Schedule – a cron formatted schedule
- Suspend – suspecnds the cron job
- startingDealineSeconds – the Job that hans't started on this amount of time is marked as filed.
- ConcurrencyPolicy – handled concurrent execution of Job, allowed values are 'Allow', 'Forbid' and 'Replace'

cronjob.yaml

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
      name: hello-world-cron
spec:
      schedule: "*/1 * * * *"     # here the job will get scheduled every minute
      jonTemplate:
```

```
spec:
  template:
    sepc:
        containers:
         - name: ubuntu
           image: ubuntu
           command:
            - "/bin/bash"
            - "-c"
            - "/bin/echo Hello From Pod $(hostname) at
              $(date)"
           restartPolicy: Never
```

Demostration:

```
$ kubectl apply -f cronjob.yaml
```

 Here we will be to see job getting scheduled in turn starting POD every minute and thus PODs starting → completing → creating new

StatefulSet:

This api object helpts to maintain a statefull application to run inside a POD, i.e. we will be able to run a statefull application with help persistent storage, persistent Naming, or Headless Service.
Service / application like databse service need persistent access to the datafile.